CHPSIM

Version 2.0

A Simulator and Debugger for the CHP Language

—User Manual—

Written by Marcel van der Goot and Chris Moore

Contents

1 Getting Started **5**
- 1.1 Running the simulator . . . 5
- 1.2 Example program . . . 5
  - 1.2.1 Modules . . . 7
  - 1.2.2 Functions . . . 7
  - 1.2.3 Types . . . 7
  - 1.2.4 Meta processes . . . 7
  - 1.2.5 CHP . . . 8

2 Language **9**
- 2.1 Source file . . . 9
  - 2.1.1 Importing modules . . . 10
- 2.2 Types . . . 11
  - 2.2.1 Boolean type . . . 11
  - 2.2.2 Integer types . . . 11
    - 2.2.2.1 Integer fields . . . 12
  - 2.2.3 Symbol types . . . 12
  - 2.2.4 Array types . . . 12
  - 2.2.5 Record types . . . 13
  - 2.2.6 Union types . . . 13
  - 2.2.7 Wired types . . . 14
  - 2.2.8 Templated types . . . 14
- 2.3 Constants . . . 14
- 2.4 Routines . . . 14
  - 2.4.1 Parameter passing . . . 14
  - 2.4.2 Functions . . . 15
  - 2.4.3 Procedures . . . 15
  - 2.4.4 Processes . . . 16
  - 2.4.5 Routine body . . . 16
  - 2.4.6 Scope and order of definition . . . 17
- 2.5 Statements . . . 17
  - 2.5.1 Assignment . . . 17
  - 2.5.2 Communication . . . 18
  - 2.5.3 Repetition . . . 18
  - 2.5.4 Selection . . . 19
  - 2.5.5 Procedure call . . . 19
- 2.6 Expressions . . . 20
  - 2.6.1 Binary expressions . . . 20
    - 2.6.1.1 Arithmetic operators . . . 20
    - 2.6.1.2 Comparison . . . 21

2.6.1.3 Logical and bitwise operators . . . 21
2.6.1.4 Concatenation . . . 21
2.6.1.5 Replicated expressions . . . 21
2.6.2 Postfix expressions . . . 21
2.6.2.1 Indexing of arrays . . . 21
2.6.2.2 Indexing of integers . . . 22
2.6.2.3 Accessing fields of records . . . 22
2.6.2.4 Accessing fields of unions . . . 22
2.6.2.5 Accessing wires . . . 23
2.6.3 Prefix expressions . . . 23
2.6.3.1 Arithmetic operators . . . 23
2.6.3.2 Logical and bitwise operator . . . 23
2.6.3.3 Probe . . . 24
2.6.4 Atoms . . . 24
2.6.4.1 Array constructor . . . 24
2.6.4.2 Record constructor . . . 24
2.6.4.3 Function call . . . 25
2.6.5 Literals . . . 25
2.7 Meta processes . . . 25
2.7.1 Process instances . . . 26
2.7.2 Meta parameters . . . 26
2.7.3 Connecting processes . . . 27
2.8 Lexical tokens . . . 27
2.8.1 Comments . . . 28
2.8.2 Integers . . . 28
2.8.3 Identifiers . . . 28
2.8.4 Keywords . . . 28
2.8.5 Characters and strings . . . 28

3 Simulation **31**
3.1 Command line arguments . . . 31
3.2 Execution . . . 33
3.2.1 The current statement . . . 33
3.2.2 Steps . . . 34
3.2.2.1 Execution of functions . . . 34
3.2.3 Breakpoints . . . 34
3.2.4 Tracing . . . 35
3.2.5 Inspecting the state . . . 35
3.2.6 Other commands . . . 36
3.2.7 Instantiation . . . 36
3.3 Built-in procedures . . . 37
3.4 Standard I/O . . . 38

Chapter 1

Getting Started

In this chapter we give an example program that illustrates several features of the CHP language. Although the example is no substitute for reading the rest of the manual, it is a good idea to try the example first, as a means of getting started.

1.1 Running the simulator

To make sure that the simulator is installed, run

```
chpsim -help
```

Assuming that `chpsim` is installed, this will print a version number, followed by a list of command line options. These options are described in Chapter 3, but for now, ignore them. Make sure that the version of chpsim is at least 2.0, as earlier versions have significant differences.

1.2 Example program

Create a file `demo1.chp` with the following contents (without the line numbers).

```
1   requires "stdio.chp"; // for reading and writing files
2
3   export function is_odd(x: int): bool
4   CHP
5     { [   x[0] -> is_odd := true
6       [] ~x[0] -> is_odd := false
7       ]
8     }
9
10  type command = { Push, Pop, Top, Show };
11
12  type ubyte = {0 .. 255};
13
14  process stack(N: int)(I?: command; P?: ubyte; Q!: ubyte)
15  CHP
16    { var s: array [0..N-1] of ubyte;
17      var c: command;
18      var top: int = -1;
19      *[ I?c;
```

```
20          [ c = ‘Push -> top := top + 1;
21                          P?s[top];
22          [] c = ‘Pop  -> Q!s[top];
23                          top := top - 1;
24          [] c = ‘Top  -> [  is_odd(s[top]) -> print(s[top], "is odd")
25                         [] ~is_odd(s[top]) -> print(s[top], "is even")
26                          ]
27          [] c = ‘Show -> show(s, top)
28           ]
29        ]
30     }
31
32   /* note: tester is a terminating process */
33   process tester()(cmnd!: command; p!, q?: ubyte)
34   CHP
35    { var x: ubyte;
36      cmnd!‘Push, p!5;
37      p!6, cmnd!‘Push;
38      cmnd!‘Pop, q?x;
39      show(x);
40      cmnd!‘Top;
41      cmnd!‘Push; p!x+2;
42      cmnd!‘Show;
43    }
44
45   process main()()
46   META
47    { instance ps: stack;
48      instance ts: tester;
49      ps(5);
50      connect ps.P, ts.p;
51      connect ps.Q, ts.q;
52      connect ps.I, ts.cmnd;
53    }
```

To simulate the example, run

```
chpsim demo1.chp
```

At several points during simulation, the simulator will stop with a ‘`(cmnd?)`’ prompt. To continue the simulation, hit the enter key. Eventually, the simulation will terminate with an error message about deadlock, at which point you can hit enter to exit chpsim.

To better understand the program, you may want to step through it. You do this by entering commands at the `(cmnd?)` prompt. The `help` command will list the available commands. For example, try

```
(cmnd?) next /ts
```

then hit the enter key at subsequent `(cmnd?)` prompts. `/ts` is the instance name of the *tester* process (line 48); note the ‘/’: process instance names always start with a slash. With this command, the simulation will stop every time before executing a statement of the *tester* process. It will identify the line number and the statement where it is stopped. For instance

```
(next) /ts at demo1.chp[36:14]
        p!5
```

says that the simulator is stopped at `p!5`, at line 36 of `demo1.chp`, to be executed by `/ts`. The `:14` identifies the character position in the line.

Another good command to try is `print`. For instance, `print x` will print the value of variable *x*. `print /ts` will print the port connections of process */ts*.

Below we give a very brief description of the example code; consult Chapter 2 for the details.

1.2.1 Modules

Line `1`. A program can consist of multiple modules. A module is imported with a `requires` clause. Module `stdio.chp` is a standard module that provides routines for reading and writing files; the example does not actually use any of the `stdio.chp` routines. A module can be in the current working directory or in the module search path. The `-v` option of `chpsim` will report which files are actually read. There is no default extension (such as `.chp`).

To make a name visible outside a module (when that module is imported), it must be exported with `export`. The only name exported by `demo1.chp` is function *is_odd* at line `3`.

1.2.2 Functions

Lines `3`–`8`. Function *is_odd* determines whether an integer is odd or even. The expression `x[0]`, when *x* is an integer, returns bit 0 of the integer. For purpose of bit manipulation, integers always use a 2's complement notation. Integers have infinite precision.

The function uses a selection statement to compute its result. In this case, the selection could be replaced by a single assignment:

```
is_odd := x[0]
```

Note that the body of the function starts with the `chp` keyword. An alternative is seen at line `46`. Not demonstrated here are procedures, which are very similar to functions but do not return a result (but they can assign result parameters).

1.2.3 Types

Lines `10`–`12`. These lines define two new types. *command* is a symbol type with four possible values. Note that these values are symbols and not strings — strings are contained in double quotes and can contain any characters, while symbols (outside of the type declaration) are preceded by a single back tick and must be alphanumeric characters. Variable *c* at line `17` has this type.

ubyte is a range of integers corresponding to an unsigned byte. Values of type *ubyte* and type `int` can be mixed in expressions. However, whenever a value is assigned to a *ubyte* variable, it must be within the range of that type (there is no automatic truncation). See Section 2.2 for more details.

There are also array types (line `16`) and record types, as well as the boolean type `bool`.

1.2.4 Meta processes

Lines `45`–`53`. There are two types of processes, meta processes, and CHP processes. Meta processes only execute statements to create a graph of CHP processes; the actual computation is performed by the CHP processes. The simulator identifies two different phases, 'instantiation', when only meta processes are executed, and 'CHP execution', when the actual computation takes place. By default, the simulator starts by executing a process called *main*.

Line `47` creates an instance of the *stack* process. Its parameter *N* is assigned at line `49`, and its ports connected at lines `50`–`52`. The *tester* process has no meta parameters; therefore, the equivalent of line `49` would be a statement `ts();`. Since this statement has no effect (*ts* is already created at line `48`), it has been omitted.

1.2.5 CHP

Lines 14–43. Processes have two parameter lists, one for meta parameters (like N at line 14), the other for ports. Input ports are identified by a '?', output ports by a '!' (there are also synchronization ports, which are just identifiers).

Output ports must be connected to input ports of similar type. As with variables, a run-time check compares actual values with the valid range. The *stack* process and the *tester* process are connected to each other. However, the *stack* process has an infinite loop (*[...]), whereas the *tester* process terminates. Therefore, at some point the *stack* will attempt I?c (line 19), when I is no longer connected to another process. The simulator detects this error, then terminates. (When the error occurs, try (cmnd?) print /ps.)

Note that the *stack* process lacks any protection against overflow and underflow of the stack. You are encouraged to use the simulator to see what happens when these errors occur.

The *print* and *show* procedures (e.g., lines 24 and 27) are built-in to the simulator.

Chapter 2

Language

Words in *italics* denote non-terminals or non-literal terminals; words and symbols in `typewriter font` denote keywords and literal symbols. Choices are separated by bar symbols, '|'; braces are used for grouping ('{' is for grouping, '`{`' is a literal brace). The following subscripts are used:

$item_{opt}$	—	an optional item
$item_{series}$	—	one or more items
$item_{list}$	—	one or more items separated by commas
$item_{seq}$	—	one or more items separated by semi-colons
$item_{tseq}$	—	sequence with optional terminating semi-colon

There are also $item_{series\text{-}opt}$ etc., denoting zero or more items.

2.1 Source file

---

source_file:
    required_module$_{series\text{-}opt}$ *global_definition*$_{series\text{-}opt}$

global_definition:
    `export`$_{opt}$ *definition*

definition:
    type_definition
| *const_definition*
| *function_definition*
| *procedure_definition*
| *process_definition*
| *field_definition*

The source can be distributed over multiple source files, called modules.

A definition defines the meaning of a name. If the definition is marked with `export`, the name is visible outside its own module, if that module is imported. Exporting a name is necessary if you want to refer to that name in a different module; you do not need to export objects that you do not refer to by name. E.g.,

```
export type abc = array [0..10] of pqr;
```

exports the name *abc*, but not *pqr*. It is not necessary to export *pqr* in order to access elements of an array of type *abc*.

2.1.1 Importing modules

required_module:
`requires` *string_literal*$_{list}$ `;`

The *string_literal* is a file name, denoting a module that should be imported. There is no default file name extension, but there is a search path (see `-I` in Section 3.1).

Importing a module makes the exported names of that module visible. However, if multiple imported modules export the same name, that name will not be visible. Imported names are overridden by names defined in the *source_file* itself. Importing is not recursive.

It is allowed to have circular dependencies between modules. E.g., *m1* may require *m2* while *m2* requires *m1*. (It makes sense to allow this, since routines can be defined in any order; Section 2.4.6.) However, there is an important restriction on such circular dependencies:

> If module *m1* has a circular dependency with module *m2*, then the top-level declarations of *m1* must not depend on *m2*.

Top-level declarations are the definitions in the syntax of *source_file*, but for routines the restriction only applies to the parameters and return type, not to the body. The following example is correct code.

```
// file m1
requires "m2";

export type color = {red, green, blue};

export function f1(x: color): {0..255}
CHP
  { var y: byte;
    y := f2(0);
    f1 := y + 1
  }
```

```
// file m2
requires "m1";

export type byte = {0..255};

export function f2(x: byte): byte
CHP
  { f2 := x * 2 }

export function g2(x: byte): byte
CHP
  { g2 := f1(blue) }
```

However, the restriction prevents us from changing the return type of *f1* to *byte*, because the return type is part of *m1*'s top-level declarations. If we want to share the definition of *byte*, we should put it in a third module that is imported by both *m1* and *m2*.

Although importing is not recursive, the notion of dependency is transitive. Hence, there can be circular dependencies involving more than two files. The restriction applies to any two modules in the same cycle.

A module should not import itself.

2.2 Types

---

type_definition:
`type` *identifier* = *type* `;`

type:
integer_type
| *symbol_type*
| *array_type*
| *record_type*
| *union_type*
| *wired_type*
| *generic_type*
| *identifier*

generic_type:
`bool` | `int` | `symbol`

An *identifier* used as *type* must have been defined with a *type_definition*.

As a special case, if the *type* in an exported *type_definition* is a *symbol_type* (Section 2.2.3), the symbol literals are exported as well.

The language distinguishes between generic types and specific types; each specific type can be reduced to a generic type. Typically, variables have specific types, whereas expressions have only generic types. When types need to match, such as for an assignment, it is sufficient if the generic types are equal; hence, the typing is weak. Matching of types can be verified at compile-time. However, each variable may only hold values that belong to its specific type; the simulator uses run-time checks to verify this condition. We use the word 'type' when the context makes clear whether we are referring to generic or specific types.

2.2.1 Boolean type

The generic `bool` type is an exception, in that there are no specific types. Values of `bool` type are usually produced by comparisons, but you may declare variables of type `bool` as well. The `bool` type has only two values, `true` and `false`.

2.2.2 Integer types

---

integer_type:
`{` *const_range* `}`

const_range:
const_expr `..` *const_expr*

---

integer_type describes a specific type. The two expressions denote lower and upper bounds (the lower bound must come first). The corresponding generic type is the arbitrary-precision type `int`, which is used for all integer expression evaluation. Any integer expression can be assigned to any integer variable, provided that the value lies within the bounds of the variable's specific type. Note that there is no automatic truncation of values.

It is possible to declare variables or parameters of the generic `int` type. This is not recommended for code that describes hardware, but may be useful when defining generic functions (such as *is_even()*).

2.2.2.1 Integer fields

---

field_definition:
`field` *identifier* `=` `[` *const_range* `]` `;`

This syntax gives a name to the specified range of bits. In this case, the higher bound may come first. Note that the field name is not tied to a particular integer type: it can be used with any integer value. See Sections 2.6.2.3 and 2.6.2.2.

2.2.3 Symbol types

---

symbol_type:
`{` $identifier_{list}$ `}`

A symbol type consists of a set of symbol literals, which are just names. For instance

```
type color = { red, orange, yellow, green, blue, purple };
```

Unlike enumeration types in some languages, there is no ordering among symbol literals, nor is there an implied mapping to integers. The generic type for all symbol types is `symbol`. Since all symbol values belong to the generic type, they are identified only by their name: if types *color* and *fruit* both have an *orange* value, these values are identical. The first occurrence of a name in a *symbol_type* declares the name as a symbol literal; subsequent occurrences (in the same scope) do not count as declarations. Within a single *symbol_type*, each name should occur only once.

To export symbol literals, the *symbol_type* must be given a name with an exported *type_definition* (Section 2.2).

2.2.4 Array types

---

array_type:
`array` `[` $const\_range_{list}$ `]` `of` *type*

Having multiple ranges is merely a short-hand for nested array types: the two types

```
array [1..10, 0..5] of byte
array [1..10] of array [0..5] of byte
```

describe exactly the same specific type. Internally, the simulator always uses the latter form.

The generic type of a specific array type is obtained by omitting the bounds, and replacing the element type by its generic type. For example,

```
var a: array [1..10] of byte;
var b: array [0..5] of {-100..100};
var c: array [1..10] of {red, green, blue};
```

Arrays a and b have the same generic type, namely `array of int`, but c does not. When an array value is assigned to an array variable, the number of elements must be equal, and the assignment of each individual element must be valid. Hence, $a := b$ can never be correct, but $b := a[i..j]$ may be correct.

2.2.5 Record types

---

record_type:
`record {` *record_field*$_{tseq}$ `}`

record_field:
identifier$_{list}$ `:` *type*

The field names are local to each specific *record_type*. The generic type of a specific record type is obtained by omitting the field names, and replacing each field type by its generic type. For example,

```
var a: record { x, y: byte };
var b: record { p, q: {-100..100} };
var c: record { x, y, z: byte };
```

Both *a* and *b* have the same generic type, namely

```
record { int; int }
```

but *c* does not. Assignment of record types requires that each of the field assignments is valid.

The field names are always accessed through an object of the record type; hence, they need not be exported.

2.2.6 Union types

---

union_type:
`union {` *union_field*$_{tseq}$ `}`

union_field:
identifier `{` *identifier* `,` *identifier* `}` `:` *type*
`default :` *type*

Again, the field names are local to each specific *union_type*, and need not themselves be exported.

A union type should contain exactly one union field with the `default` keyword. This default type should not be an array or a record, nor should it be another union. An expression of union type has the same generic type as its default type, and is effectively equivalent to this type. The exception to this is when postfix union access is applied to the expression (Section 2.6.2.4).

Every other field must contain two function references after the field name. The first should take one argument of the default type and return a value of the field's type. The second should take one argument of the field's type and return a value of the default type.

An example of a union type is:

```
type byte = union {
    default : {0..255};
    signed{ui2si,si2ui} : {-128..127};
}
```

2.2.7 Wired types

---

wired_type:
`(` *wire_field*$_{list}$ `;` *wire_field*$_{list}$ `)`

wire_field:
identifier {`[` *const_range*$_{list}$ `]`}$_{opt}$ *wire_initializer*$_{opt}$

wire_initalizer:
`+` [] `-`

Variables and constants may not have a wired type. Only a port may have a wired type. The wired type is similar to the record type, except that all elements are booleans, or arrays of booleans.

2.2.8 Templated types

The syntax for a templated type is:

---

identifier `:` `type`

The syntax is similar to a declaration, but in this case identifier is defined as a named type, not an expression. The expression must appear as a meta parameter (Section 2.7.2) for a process. With regards to type checking, the identifier is regarded as a generic type unique unto itself (within this process).

2.3 Constants

---

const_definition:
`const` *identifier* {`:` *type*}$_{opt}$ *initializer* `;`

initializer:
`=` *const_expr*

If the *type* is specified, the *initializer* must be a value of that type; otherwise, the (generic) type of the constant is determined directly from the initializer.

2.4 Routines

The term 'routine' refers to functions, procedures, and processes.

2.4.1 Parameter passing

---

value_parameter:
`const`$_{opt}$ `val`$_{opt}$ *identifier*$_{list}$ `:` *type*

result_parameter:
`res` *identifier*$_{list}$ `:` *type*
[] `valres` *identifier*$_{list}$ `:` *type*

The term 'parameter' refers to formal parameters; 'argument' refers to actual parameters.

The parameter passing mechanism for functions and procedures is value-result (a.k.a. copy-restore) passing: `val` and `valres` parameters get their initial value assigned from the corresponding arguments at the very beginning of the call. At the very end of the call, the final values of `res` and `valres` parameters are copied to the corresponding arguments of the call. In a hardware implementation, this mechanism easily translates to receiving and sending initial and final values. During the call, the parameters are local variables of the routine (note that `res` parameters are not initialized).

```
procedure g(val p: int; valres q: int; res r: int)
CHP { q := q + p;
      p := q;
      r := p + 1;
    }
```

Suppose we have $x = 3$ and $i = 1$, then call $g(x, i, a[i])$. Afterward $x = 3$, $i = 4$, and $a[1] = 5$. Note that the location of $a[i]$ was determined before the call, so that $a[1]$ is modified rather than $a[4]$.

If a `val` parameter is declared constant, it cannot be written to, but it may be used in places such as the bounds of an array that require a *const_expr*.

The type requirements for parameter passing are the same as for assignment. In addition, the arguments for `res` and `valres` parameters must be ℓ-values (writable locations). It is an error to pass the same location for two result parameters in the same call. E.g., with the above example, $g(x, i, i)$ would be wrong.

2.4.2 Functions

---

function_definition:
`function` *identifier* `(` *value_parameter*$_{seq}$ `)` `:` *type chp_body*

A function has a return type and is called as part of an expression. The parameters of functions are always `val` parameters. Inside the function, the function name acts like a `res` parameter. The value you assign to it corresponds to the function's return value.

A function may contain nested functions and procedures, but no processes.

Because there are no global variables, and because function parameters are value parameters, functions are free of side-effects. Also, functions do not have a persistent state.

2.4.3 Procedures

---

procedure_definition:
`procedure` *identifier* `(`
{*value_parameter* `[]` *result_parameter*}$_{seq\text{-}opt}$ `)` *chp_body*

A procedure has no return type; its call is a statement.

A procedure may contain nested functions and procedures, but no processes.

A procedure can modify its environment by modifying result parameters, but otherwise has no side-effects, nor a persistent state.

2.4.4 Processes

---

process_definition:
`process` *identifier* `(` *meta_parameter*$_{seq\text{-}opt}$ `)`
`(` *port_parameter*$_{seq\text{-}opt}$ `)` *process_body*

port_parameter:
{*identifier* `[` *const_range*$_{list}$ `]`$_{opt}$ *direction*}$_{list}$ `:` *type*
[] {*identifier* `[` *const_range*$_{list}$ `]`$_{opt}$ }$_{list}$

direction:
`?` [] `!`
process_body:
chp_body
[] *meta_body*
[] *chp_body meta_body*

Processes have no value or result parameters; instead they have meta parameters and ports. Port parameters of the first form are data ports, used for sending and receiving values. A '`?`' indicates an input port, a '`!`' indicates an output port. The *type* is the data type of the port.

Port parameters of the second form are synchronization ports. Both forms of ports can be arrayed by specifying the array range within brackets, and multidimensional arrays of ports can be created with a comma separated list. For data ports, the following two port paramters are equivalent:

```
X[0..N-1]? : T;
X? : array [0..N-1] of T;
```

Processes are not called, but instantiated. Within a process, the meta parameters act as constants (e.g., they can be used in type definitions).

There are two types of processes, meta processes and CHP processes. Meta processes serve to instantiate other processes, eventually resulting in a process graph with only CHP processes. Meta processes cannot contain communications. Only when the complete process graph has been created, does the execution of the CHP program start. Meta processes are explained in Section 2.7.

Often a meta process is writen to replace a sequential chp process with a collection of processes that perform in parallel the same function. When this happens, the meta body should be appended to the chp body rather than replacing it. By default the parallel version is chosen, but the sequential chp still plays a functional role (see Section 3.2.7).

2.4.5 Routine body

---

chp_body:
`chp {` {*definition* [] *declaration*}$_{series\text{-}opt}$
parallel_statement$_{tseq\text{-}opt}$ `}`

declaration:
`var` *identifier*$_{list}$ `:` *type initializer*$_{opt}$ `;`

Definitions inside a routine body are local to that routine; unlike global definitions, they cannot be exported. Variables are always local to a routine.

If a variable has an initial value, the value must be of the variable's type. Unlike for constants, a variable's type cannot be omitted.

2.4.6 Scope and order of definition

All names, including symbol literals, are in the same name space. However, there are multiple nested scopes, due to the nesting of definitions. Scope is always static, i.e., the meaning of a name can be determined at compile time. The outermost scope level contains the names exported by imported modules. The next scope level is the module (*source_file*) itself. Definitions and declarations inside a body are local to that body. Parameters of routines are also local to that routine's body.

A name can always be redefined in a nested scope, hiding the original meaning of the name. Each name may be defined or declared only once in a particular scope, except for the names exported by imported modules. In case of the latter, if different modules export the same name, neither meaning of the name is visible. As explained in Section 2.2.3, all symbol literals belong to the same generic type. Symbol literals are only declared the first time they are encountered (in a scope).

Most names are visible in their own scope, and in nested scopes if they have not been redefined. However, to avoid shared variables, variables must always be local to the routine that uses them.

Routines can be defined in any order, but all other names must be defined or declared before they are used. (However, if a process definition and instantiation occur at the same scope level, then the definition must precede the instantiation.)

2.5 Statements

---

parallel_statement:
    statement$_{list}$

statement:
    `skip`
[] *assignment*
[] *communication*
[] *loop_statement*
[] *selection_statement*
[] *procedure_call*
[] `{` *parallel_statement*$_{tseq}$ `}`
[] *replicated_statement*

replicated_statement:
    `<<` { `;` [] `,` } *identifier* `:` *const_range* `:` *parallel_statement*$_{tseq}$ `>>`

Statements separated by commas are executed in parallel. A variable that is only read may be read by multiple statements in parallel. However, a variable that is modified may be accessed by only one of a group of parallel statements. With respect to this rule, communications count as modification of the port. If this exclusion rule is violated, the effect is undefined.

Statements separated by semi-colons are executed in sequence. From the syntax it follows that commas bind tighter than semi-colons. Braces can be used to alter the binding.

The replicated statement executes the set of statements after the second colon once for each value of the identifier within the given range. If the replicator symbol is a comma, all of the statement sets are executed in parallel. If it is a semi-colon the statements sets are executed sequentially, starting with the lowest value in the replication range and ending with the highest.

A `skip` statement has no effect.

2.5.1 Assignment

---

assignment:

lvalue `:=` *expr*
[] *lvalue* `+`
[] *lvalue* `-`

lvalue:
postfix_expr

An expression is an ℓ-value if it corresponds to (part of) a variable. In Section 2.6 we indicate which expressions are ℓ-values.

An assignment with an `:=` symbol requires that both expressions have the same generic type, and that the value belongs to the type of the assigned variable.

The other two forms of assignment require that the ℓ-value has type `bool`; a `+` sets the boolean to `true`, a `-` sets it to `false`.

2.5.2 Communication

---

communication:
port_expr
[] *port_expr* `!` *expr*
[] *port_expr* `?` *lvalue*
[] *port_expr* `!` *port_expr* `?`
[] *port_expr* `#?` *lvalue*

port_expr:
expr

The different forms of *communication* are referred to as sync, send, receive, pass, and peek, respectively.

Note that a port of array or record type may either be connected to a single channel that passes the data as a whole, or as a group of channels that passes each field of data seperately (Section 2.7.3). If the former is the case, an attempt to access a field of the port in one of the above communications will result in an error during simulation. To avoid this, it is a good idea to make note of which ports need to connect to the latter type of channel at the beginning of a process definition.

For a sync, the port must be a synchronization port; for all other communications it must be a data port.

For a send, the port must be an output port, and the expression must be valid for the data type of the port.

For a receive and a peek, the port must be an input port, and the value received is assigned to the ℓ-value. The peek receives a value without removing it from the port; hence, a subsequent receive or peek will receive the same value.

The first port of a pass must be an output port, the second must be an input port. The pass receives a value from the input port, and simultaneously (without introducing slack) sends it via the output port. The value received must be valid for the data type of the output port.

All communication actions, including the peek, suspend until they can complete. None of these communications can be used on a port of wired type.

2.5.3 Repetition

---

loop_statement:
`*[` *guarded_command_list* `]`
[] `*[` *arbited_guarded_command_list* `]`
[] `*[` $parallel\_statement_{tseq}$ `]`

guarded_command_list:
    guarded_command
[] *guarded_command_list* { [] *guarded_command_list* }$_{series\text{-}opt}$
[] `<<` [] *identifier* `:` *const_range* `:` *guarded_command_list* `>>`

arbited_guarded_command_list:
    guarded_command
[] *arbited_guarded_command_list* { `[:]` *arbited_guarded_command_list* }$_{series\text{-}opt}$
[] `<<` `[:]` *identifier* `:` *const_range* `:` *arbited_guarded_command_list* `>>`

guarded_command:
    bool_expr `->` *parallel_statement*$_{tseq}$

bool_expr:
    expr

The guard of a guarded command, i.e., the expression before the arrow, must have `bool` type. Execution of a guarded command consists of execution of the statement sequence.

The first two forms of the *loop_statement* choose one guarded command from the guarded command list that has a true guard, and execute it. For the replicated guard list, a value of the identifier within the given range which produces a true guard is chosen. This process is repeated until no guard is true. It is an error if any (non arbited) guarded command list contains more than one true guard (at the time the choice is made). For an arbitrated guarded command list, an arbitrary choice is made among the guarded commands with a true guard.

The last form of the *loop_statement* executes the statement sequence repeatedly, forever.

2.5.4 Selection

---

selection_statement:
    `[` *guarded_command_list* `]`
[] `[` *arbited_guarded_command_list* `]`
[] `[` *bool_expr* `]`

The first two forms of the *selection_statement* wait until one of the guarded commands has a true guard, then execute one of the guarded commands that has a true guard. The distinction between the first two forms is the same as for the *loop_statement*: with the [] separator it is an error if more than one guard is true.

The last form of selection statement simply waits until the expression (which must have `bool` type) is true.

2.5.5 Procedure call

---

procedure_call:
    identifier { `(` *expr*$_{list\text{-}opt}$ `)` }$_{opt}$

The *identifier* must be the name of a procedure (not of a function). The argument expressions match the parameters of the procedure, in order. Let expression x match parameter p. If p is a `val` or `valres` parameter, the assignment $p{:=}x$ is performed at the start of the procedure call. If p is a `res` or `valres` parameter, the assignment $x{:=}p$ is performed at the end of the procedure call. In all cases, the standard rules for the assignment apply. In addition, the same ℓ-value may not be used for two different result parameters.

The parentheses are optional if the argument list is empty.

2.6 Expressions

const_expr:
expr

expr:
binary_expr

An expression is a constant expression if all its constituent expressions are constants.

2.6.1 Binary expressions

binary_expr:
prefix_expr
binary_expr binary_operator binary_expr
replicated_expr

binary_operator:

^				
*	/	%	mod	
+	-	xor		
<	<=	>	>=	
=	!=			
&	|			
++				

replicated_expr:
`<<` *binary_operator identifier* `:` *const_range* `:` *binary_expr* `>>`

The ambiguity in the syntax of binary expressions is resolved by using operator precedence. Each line in the definition of *binary_operator* corresponds to a precedence level: the first line has the highest level, i.e., '`^`' binds the tightest. Operators listed on the same line have equal precedence. Among operators with equal precedence the order is left-to-right, i.e., all operators are left-associative.

A binary expression is not an ℓ-value.

2.6.1.1 Arithmetic operators

The following operators require that their operands have generic type `int`; the result of the operation has type `int`:

`^ * / % mod + -`

'`^`' denotes exponentiation: `2^N` means 2^N. Since the result is an integer, the second operand must be ≥ 0 ($x^0 = 1$ always).

'`/`' is integer division (rounding towards 0). Consider a/b. If a and b have the same sign, the result is ≥ 0; otherwise the result is ≤ 0. '`%`' is the remainder of division. The sign of $a\%b$ is the sign of a.

`mod`, on the other hand, is the standard modulo operation, which always yields a non-negative result ($a \bmod b = a \mod |b|$). The operations are summarized by the following example.

$$\begin{array}{rrrrrrrrrrrr}
10/\ 3 = & 3 & & 10\%\ 3 = & 1 & & 10 & \bmod & 3 & = & 1 \\
-10/\ 3 = & -3 & & -10\%\ 3 = & -1 & & -10 & \bmod & 3 & = & 2 \\
10/-3 = & -3 & & 10\%-3 = & 1 & & 10 & \bmod & -3 = & & 1 \\
-10/-3 = & 3 & & -10\%-3 = & -1 & & -10 & \bmod & -3 = & & 2
\end{array}$$

The `/`, `%`, and `mod` operators require that their second operand is not 0.

2.6.1.2 Comparison

The `<`, `<=`, `>`, and `>=` operators all yield a `bool` result. Their operands must either both have generic type `int`, or both have type `bool` (where `false<true`).

The `=` and `!=` operators also yield a `bool` result. They require that both operands have the same generic type.

2.6.1.3 Logical and bitwise operators

The `&`, `|`, and `xor` operators require that either both operands have type `bool`, or both operands have generic type `int`.

If the operands have type `bool`, the result also has type `bool`.

If the operands have generic type `int`, the result also has type `int`; in this case, the operation is a bitwise operation. Note that precedence of the operators is what is usually expected for the logical operations. Due to the arbitrary precision of integer expressions, care must be taken when bitwise operations are applied; see Section 2.6.3.2.

2.6.1.4 Concatenation

The `++` operator denotes array concatenation. It requires that both operands have the same generic array type; the type of the result is that generic array type.

2.6.1.5 Replicated expressions

As with other forms of replication, `<<` **op** `i : n..m : expr(i) >>` is equivalent to: `expr(n)` **op** `expr(n+1)` **op** $\cdots$ **op** `expr(m)` However, to avoid ambiguity in the meaning of this expression, only associative binary operators may be used in a replicated expression.

2.6.2 Postfix expressions

---

postfix_expr:
- *atom*
- *array_access*
- *record_access*
- *union_access*
- *wire_access*

2.6.2.1 Indexing of arrays

---

array_access:
- *postfix_expr* `[` *expr* $_{list}$ `]`
- *postfix_expr* `[` *expr* `..` *expr* `]`

In both cases the *postfix_expr* must have a specific array type, or have generic type `int` (see Section 2.6.2.2). The whole expression is an ℓ-value if the *postfix_expr* is an ℓ-value.

The expression $x[3, 4, 5]$ is exactly the same as $x[3][4][5]$; internally, the simulator uses the latter form.

If the *postfix_expr* has a specific array type with element type T, then the first form of *array_access* has result type T. The second form (sometimes called a *slice*) has as result type the generic type of the

postfix_expr. Note that you can only index an array with a specific type, i.e., an array with known bounds. Hence $x[1..4][2]$ is not a valid expression. The indices must be within the bounds of the array. For a slice, the smallest index must come first.

2.6.2.2 Indexing of integers

The same syntax used to index arrays can be used to access bits of an integer. In this case, the indices must be ≥ 0. The first form, with a single index, has result type `bool`. The second form (the slice) has result type `int`.

As explained in Section 2.6.3.2, some care is necessary when treating integers as arrays of bits. In particular, you should never set the sign bit directly. E.g.,

```
var x: {-128..127};
x[7] := true
```

If x was > 0, then the assignment makes $x > 128$, not negative, because the infinite sequence of 0 sign bits has not been changed. The following procedure sets the sign bit correctly.

```
// set sign bit x[pos] to b
procedure set_sign(valres x: int; pos: int; b: bool)
CHP
  { [   b -> x := x | -2^pos
    [] ~b -> x := x & (2^pos - 1)
    ]
  }
```

The value of a slice is obtained by treating the specified bits as an unsigned integer. Hence, with the above x, $x := x[0..7]$ is unsafe, because the result has range 0..255. When taking a slice of an integer, it is allowed to put the largest index first.

Indexing of integers may only be applied to contants and variables, not to ports.

2.6.2.3 Accessing fields of records

---

record_access:
postfix_expr `.` *identifier*

The *postfix_expr* must have a specific record type or have generic type `int`. The expression is an ℓ-value if the *postfix_expr* is an ℓ-value.

In case of a record type, the identifier must be one of the record's field names. The result type is the type of the field.

If the *postfix_expr* is an integer, the identifier must be a field name defined with a *field_definition* (Section 2.2.2.1). This specifies a slice of the integer, and is completely equivalent to the array notation for slices explained in Section 2.6.2.2.

2.6.2.4 Accessing fields of unions

---

union_access:
postfix_expr `.` *identifier*

The *postfix_expr* must be a port with a specific union type. This expression may only appear in a connection statement.

The identifier must be one of the union's field names. This allows us to pipe each value on a channel through the field's specified function. For example, consider the example union type in Section 2.2.6:

```
type byte = union {
    default : {0..255};
    signed{ui2si,si2ui} : {-128..127};
}

...

connect A.O.signed, B.I;
```

Process A can send unsigned values on O, while process B receives signed values. This is possible because the union access sets up the channel to pipe all of its data through the function ui2si.

2.6.2.5 Accessing wires

---

wire_access:
postfix_expr `.` *identifier*

The *postfix_expr* must be a port of wired type.

This allows access to the wire as a boolean variable. An array of wires will be accessed as an array of boolean variables.

2.6.3 Prefix expressions

---

prefix_expr:
postfix_expr
`#` `{` $port\_expr_{list}$ `:` *bool_expr* `}`
prefix_operator prefix_expr

prefix_operator:
`+` `-` `~` `#`

A prefix expression is not an ℓ-value.

2.6.3.1 Arithmetic operators

The `+` and `-` operators require that their operand has generic type `int`, which is also the result type.

2.6.3.2 Logical and bitwise operator

The `~` operator can be applied to a `bool` operand, in which case it denotes negation and yields a `bool` result. It may also be applied to an operand with generic type `int`, in which case it is applied bitwise, resulting in the one's complement operation (the result type is `int`).

Because `int` has arbitrary precision, care must be taken when bitwise operations are applied. For the purpose of bitwise operations, an integer is considered as an infinite array of 0s and 1s, using the standard 2's complement notation. Consequently, there is always an infinite number of equal sign bits. This means that the `~` operator always changes the sign of an integer. While this is sound (since, by definition, $-x = \sim x + 1$), it may be unexpected if you intended a number to be an unsigned integer. For example, `~0 = -1`.

The binary bitwise operators likewise require care with the sign bits. Consider

```
var x: {-128..127}
```

In this case, bit 7 of x is the sign bit. However, if $x > 0$, then `x|2^7` is larger than 128, not negative, because the infinite sequence of 0 bits has not been changed. To change the sign, you would need to do `x|-2^7`.

2.6.3.3 Probe

The operand of the probe ('`#`') prefix operator must be a port. The expression returns a `bool` result, `true` if a subsequent communication action on the port will complete. Specifically, the probe will return `true` only when the process at the other end of the channel is suspended on a sync, send, receive or pass operation. In the case of the pass, the process on the opposite side of the pass must meet this criterion as well. Executing a peek or evaluating a probe on the other side of the channel does not qualify.

The value probe, the form of *prefix_expr* with braces, takes a list of ports. The *bool_expr* that follows can refer to the input ports in the list as if they were variables: each input port stands for the value that a subsequent communication will receive. The value probe is true if the individual probes of the each of the ports is true, and the expression evaluates to true.

Neither form of probe suspends.

2.6.4 Atoms

---

atom:
- *identifier*
- *literal*
- *array_constructor*
- *record_constructor*
- *function_call*
- `(` *expr* `)`

An identifier used as *atom* is an ℓ-value if it is the name of a variable or parameter; the name of a function is also an ℓ-value inside that function. (As mentioned in Section 2.4.6, variables, parameters, and function names used as variables, can only be accessed in their own scope, not in nested routines.) Other identifiers used as *atom* should be the names of constants or symbol literals. Parentheses do not affect whether an expression is an ℓ-value.

2.6.4.1 Array constructor

---

array_constructor:
`[` $expr_{list}$ `]`

The expression must all have the same generic type T. The result type is a generic `array of` T. An *array_constructor* is not an ℓ-value.

2.6.4.2 Record constructor

---

record_constructor:
`{` $expr_{list}$ `}`

The result type is a generic record where each field type is the generic type of the respective expression. A *record_constructor* is not an ℓ-value.

2.6.4.3 Function call

function_call:
identifier `(` $expr_{list}$ `)`

The identifier must be the name of a function. The requirements for the arguments are the same as for procedure calls (Section 2.5.5), but only value parameters are allowed. Note that a function must have at least one parameter. The type of the expression is the return type of the function.

A function call is not an ℓ-value. If all arguments are constants, the function call is also a constant.

2.6.5 Literals

literal:
integer_literal
[] *character_literal*
[] *string_literal*
[] *symbol_literal*
[] *boolean_literal*

symbol_literal:
identifier

boolean_literal:
`false` [] `true`

Literals are never ℓ-values. Integer literals have type `int`. Character literals have type

```
type char = {0..127}
```

Their values correspond to ASCII codes. A string literal is a 0-terminated generic `array of int`, similar to an *array_constructor*.

A *symbol_literal* must have been declared as part of a *symbol_type*.

2.7 Meta processes

meta_body:
`meta {` $\{$*definition* [] *meta_declaration*$\}_{series\text{-}opt}$
parallel_statement$*_{tseq\text{-}opt}$ `}`

A meta process is a process with a *meta_body*. The form of a meta process is nearly the same as that of a CHP process, but there are some statements and declarations that can only occur in one type of process and not in the other. The main difference is with respect to execution: The simulator starts by executing meta processes, which have as goal to create the graph of CHP processes; this is the instantiation phase. If meta process P instantiates process Q (which may be a meta process itself), then Q does not start executing until P has terminated. If Q is a CHP process, then Q does not start executing until all meta processes have terminated.

The instantiation phase ends when all meta processes have terminated. Once that happens, the execution phase starts by executing all instantiated CHP processes in parallel. The CHP processes model the actual hardware; the meta processes just serve to describe the hardware configuration.

The '*parallel_statement**' in a *meta_body* has the same form as a regular *parallel_statement*, except that '*statement*' excludes communications and is extended as follows.

statement:
[] *meta_binding*
[] *connection*

Furthermore, a meta process may not contain any probes.

2.7.1 Process instances

---

meta_declaration:
`instance` *identifier*$_{list}$ `:` *process_type* `;`
[] *declaration*

process_type:
identifier
[] `array [` *const_range*$_{list}$ `] of` *process_type*

An `instance` declaration instantiates one or more processes. The *identifier* in *process_type* must be the name of a process (as defined with `process`); this may be a meta process or a CHP process. Normally identifiers are declared immediately when they are encountered, but as a special case, instance names are only declared after parsing the *process_type*: this means the instance name may be identical to the process name (assuming they are declared at different scope levels).

Although an `instance` declaration creates the process instance, it does not specify the values of the instance's meta parameters, nor its connections to other processes. These must be specified with subsequent statements.

2.7.2 Meta parameters

---

meta_parameter:
identifier$_{list}$ `:` *meta_type*

meta_type:
type
`type`

Meta parameters get their value during the instantiation phase. Although their values are not known at compile-time, nevertheless they are considered constants. Meta parameters may be expressions of a given type, or may themselves be a templated type using the `type` keyword (Section 2.2.8).

Meta parameters are given a value with a *meta_binding*.

---

meta_binding:
instance_expr `(` *meta_expr*$_{list}$ `)`

instance_expr:
postfix_expr

meta_expr:
expr
`<` *type* `>`

The *instance_expr* must be one of the process instances declared with `instance`. The argument expressions must match the meta parameters of the corresponding process, just as with function parameters. Types enclosed in `<>` must match a templated type in the meta parameters.

There must be a *meta_binding* for each process instance that has meta parameters.

2.7.3 Connecting processes

---

connection:
`connect` *connection_list*

connection_list:
connection_point `,` *connection_point*
[] `all` *identifier* `:` *const_range* `:` *connection_list*

connection_point:
port_expr
[] *instance_expr* `.` *identifier*

The first form of *connection_list* creates a single channel to connect the two points listed. The second forms connects its sublist for each value of the identifier in the given range, just like a replicator.

There are two different ways to connect two ports of array type:

```
connect A.X, B.Y;
connect all i : 0..N-1 : A.X[i], B.Y[i];
```

The former creates a single channel between the ports, while the latter creates N channels to connect each element of the arrays. The former of these may cause a simulation error if the chp process being connected attepmts to access one of the N channels individually. The latter method is always safe, but it slows the simulation and can make debugging harder.

A similar choice exists for connecting ports of record and union type. However, because of the restrictions on union field access (Section 2.6.2.4), all methods of connecting unions are safe. This also means that arrays and records within a union type are safe.

The first form of *connection_point* identifies a port of the current process. The second form identifies a port of an instantiated process.

For data ports, if both *connection_point*s have the same form, they must be ports with opposite directions (one input, one output). However, if one *connection_point* has the first form and the other the second form, both must have the same direction. The data types of connected ports must be compatible.

Using a port of the current process indicates a pass-through: the port is not used for direct communication, but instead is merely a 'wire' between two other ports. The current implementation does not perform (run-time) range checks on the data type of a pass-through port, only on ports that are actually used in communication actions.

2.8 Lexical tokens

The description of lexical tokens is slightly informal. A token may not contain white space, unless explicitly allowed. In some cases tokens must be separated by white space to avoid ambiguity.

2.8.1 Comments

comment:
`/*` *any*$_{series\text{-}opt}$ `*/`
| `//` *any-except-linebreak*$_{series\text{-}opt}$ *linebreak*

A comment of the first form cannot contain '`*/`'.

2.8.2 Integers

integer_literal:
decimal
| {`0x` | `0X`} {*hex_digit* | `_` }$_{series}$
| {`0b` | `0B`} {*binary_digit* | `_` }$_{series}$
| *decimal* `#` {*based_digit* | `_` }$_{series}$

decimal:
decimal_digit {*decimal_digit* | `_` }$_{series\text{-}opt}$

Digits > 9 are `a..z` (case-insensitive). Integers can contain underscores, but not as first character. The '`#`' notation allows for integers in any base, $1 < base \leq 26$.

2.8.3 Identifiers

identifier:
{*letter* | `_` } {*letter* | *digit* | `_` }$_{series\text{-}opt}$

Identifiers may not be keywords; they are case-sensitive.

2.8.4 Keywords

Keywords are case-insensitive.

2.8.5 Characters and strings

character_literal:
`'` {*char_character* | `\` *printable_character*} `'`

char_character:
printable_character except

A backslash is used as escape character. The following escapes are recognized:

`\a`	BELL	`0x7`
`\b`	BS	`0x8`
`\t`	TAB	`0x9`
`\n`	LF	`0xA`
`\v`	VT	`0xB`
`\f`	FF	`0xC`

`\r`	CR	`0xD`
`\q`	XON	`0x11`
`\s`	XOFF	`0x13`
`\"`	`"`	`0x22`
`\'`	`'`	`0x27`
`\\`	`\`	`0x5C`

A *printable_character* is any character in the ASCII range `0x20` through `0x7E`, i.e., a space character or a character that involves ink when printing.

---

string_literal:
" {*string_character* [] \ *printable_character*}$_{series}$ "

string_character:
printable_character except " *and* \

Escapes for strings are the same as for characters.

Chapter 3

Simulation

3.1 Command line arguments

The `chpsim` program takes the following command line arguments.

source_file
: The top-level module. Only one source file is specified on the command line; other modules are read based on `requires` clauses.

`-main` *process*
: The execution starts with one instance of this process. This must be a process without meta parameters and port parameters. If no `-main` option is specified, the initial process is *main*.

`-I` *directory*
: Appends the directory to the module search path. If module *A* has a `requires "`*B*`"` clause, the simulator first looks for *B* in the directory where it found *A*. If *B* is not found in that directory, the simulator checks each of the directories in the search path, in order. The search path does not apply to the top-level module, nor does it apply if *B* is an absolute path name or starts with '`.`' or '`..`'. See also the `-v` option.

`-I-`
: Clears the search path.

`-C` *command* [*args* **...**]
: Execute the given command before user interaction. Multiple `-C`'s can be specified, which will execute the commands in the order listed. All arguments following the `-C`, until the next argument starting with `-`, are treated as the command and its argument, so it is a good idea to specify the *source_file* before any such commands. This option is most useful in conjunction with the source command (Section 3.2.6).

`-batch`
: By default an interactive session is started, as described in Section 3.2. With this option, the simulator executes the whole program without stopping for user commands. Any `-C` arguments present will still be executed.

`-q`
: This option also disables the interactive session, but rather than executing the program, `chpsim` will terminate. Any `-C` arguments present will still be executed (this option would be rather pointless otherwise).

`-log` *file*
: A copy of all user commands and simulator output is written to this file. When `-batch` is used, the default log file is *stderr*. (The log file only applies to simulator output, not to output generated by the CHP program; see the `-stdout` option.)

`-stdout` *file*, `-o` *file*
: Specifies the file to use for the *print* and *show* procedures described in Section 3.3, and for *stdout* defined in Section 3.4. The default is the standard output.

`-v`
: Prints version information, the search path, and the name of each file that is read.

`-trace` *instance*
: Trace this instance, in the same way as the `trace` command of Section 3.2.4. This is useful with the `-batch` option (you probably also want to use `-log`). Instance names are described in Section 3.2; they always start with a '/'.

`-traceall`
: Trace all process instances. This will generate a lot of output for anything but the shortest programs. In most cases, judiciously placed breakpoints are more helpful.

`-watchall`
: Watch all wires. This is equivalent to executing the `watch` command in Section 3.2.4 for every single wire in the program. Again, this will usually produce too much data to be helpful.

`-timed`
: Usually `chpsim` executes commands in a random order to ensure that the design does not contain any race conditions. This option will instead run the simulation with a very simple timing model that can be used to get a rough idea of the circuit's performance. Note that by default, this timing model will execute as many CHP threads as possible before simulating a single production rule, which can lead to "livelock".

`-seed`
: Specify a specific seed for the random number generator that controls the thread execution order and the `random` function (Section 3.3). By default this seed is set to zero, so that multiple runs of the same program will produce identical results.

`-timeseed`
: Set the random number generator seed to the value of the current system clock. The seed used will be reported at the beginning of operation so that the run can be repeated if necessary.

`-critical`
: This flag turns on tracking of critical timing paths, enabling the use of the `critical` command described in Section 3.2.6. This option implies the `-timed` option.

`-nohide`
: Normally, when `chpsim` uses a value union to connect two ports of different type, and the conversion between these types is specfied via a process instead of a function, `chpsim` will hide the instance of the process that converts between the two types from the various debugging commands. This option will simply treat them like any other instance.

`-strict`
: This turns on strict checking for illegal variable sharing. The syntax of CHP itself makes sharing a variable between seperate processes impossible, but it doesn't prevent sharing a variable between threads of a single process. Sometimes `chpsim` catches these errors, sometimes the error will pass unnoticed, and sometimes `chpsim` will end up in a very odd state that makes it hard to track down

the source of the error. Strict checking will report all illegal variable sharing, and no program should be considered correct if it cannot pass strict checking. However, strict checking can significantly slow down large simulations, so it is not on by default.

3.2 Execution

At any time, the simulator executes a number of threads in parallel. Typically, there is one thread per process instance, but if a process executes a parallel statement (statements separated by commas), it will temporarily have multiple threads. Each thread can be active (ready) or suspended. The simulator's `print` will, for each instance, list the current number of active and suspended threads:

```
(cmnd?) print
  /q[0]/r: 1 active threads
  /tgt: 1 active threads
  /src: 1 suspended threads
```

The example shows that the three process instances of this example have a total of three threads. The names starting with a slash are hierarchical instance names, constructed from the identifiers in instance declarations (Section 2.7.1). The initial process has as name merely a slash. Hence, `/q[0]/r` is instance `r` created by `/q[0]`, which itself was instantiated by the initial process. An indication like `example.chp[42:2]` gives, respectively, the source file, the line number, and the position on the line, of the statement that will be executed next by the thread.

The simulator executes one (non-composite) CHP statement at a time, chosen from among the active threads. Execution is fair: an active statement will eventually be executed; a suspended statement will eventually be checked to determine whether it can be made active again.

The simulation consists of two phases, the instantiation phase and the execution phase. The simulation starts by creating a single instance of the initial process. Typically, this is a meta process that instantiates other (child) processes. As explained in Section 2.7, child processes only start executing when their parent process has terminated. Furthermore, CHP processes only start executing when all meta processes have terminated, which is the end of the instantiation phase. (The initial process may be a CHP process, in which case the execution phase starts immediately.)

3.2.1 The current statement

When the simulator stops to allow input of a command, it will print something similar to the following two lines:

```
(break) /q[0]/r at example.chp[42:2]
x[i] := true
```

The first line starts with the reason that the simulator has stopped for input (in this case, a breakpoint was reached). Next on the line is the instance to which the current thread belongs, followed by the exact position in the code of the statement that the thread is on. This position is specified first by the name of the file it is in, then the line number of the statement, and finally, the exact line position of the first character of the statement. The second line printed is a copy of the statement itself.

The statement that is printed here is known as the current statement. Aside from when simulation is stopped due to an error or warning, this is the statement that will be executed next by thread. The current statement also has the "focus", meaning that it is used as a reference point for most of the simulator commands. When no process instance is mentioned, most commands apply to the current statement or the current instance. Some commands allow you to select an instance explicitly. For convenience, several of the simulator commands (such as `trace`) allow you to reference process instances that have not yet been created.

The `view`, `up`, and `down` commands let you change the instance and statement that have the focus. This is mainly useful when you want to print variable values (because variables must be in scope), but also changes the default focus for the other simulator commands. However, these commands never affect the actual order of execution: the next statement is the original current statement. See Section 3.2.5 for more details.

Commands may be abbreviated, usually to a single letter.

3.2.2 Steps

The simulator repeatedly executes statements, or steps, until there is a reason to ask for command input. The simulator stops at the beginning of the instantiation phase, and at the beginning of the execution phase. When the simulator is stopped, you can enter commands to inspect variables, to set breakpoints, etc.. To make the simulation continue, you must enter one of the three commands `step`, `next`, or `continue`. An empty command is equivalent to one of these three, depending on what made the simulator stop: if the simulator stopped because of a `step` or `next` command, that is also the default; in all other cases the default is `continue`.

The `step` command tells the simulator to execute one step, i.e., one statement, of the current process. This is the statement printed before the command prompt. For instance:

```
(step) /src at example.chp[63:17]
        p!n
(cmnd?)
```

The current process is `/src`, which is about to execute `p!n`. Entering `step` (or, in this case, an empty command) allows execution of `p!n`, and forces a stop at the next statement of `/src`. The `step` command refers to the current process only: between the current step and the next stop, there may be multiple execution steps of other processes.

The `next` command is nearly the same as the `step` command, but it treats function and procedure calls as a single step. I.e., if the current statement is a procedure call, `next` will stop at the statement after the call, whereas `step` will stop at the first statement of the called procedure.

The `step` and `next` commands can take an instance name as argument. In that case the simulator will stop when it reaches the next statement of that process instance.

The `continue` command continues execution until there is reason to stop again, such as a breakpoint.

3.2.2.1 Execution of functions

Since expressions are always part of a statement, execution of a statement as a single step implies that expression evaluation is atomic. However, this poses a problem for function evaluation: a function call is part of an expression, hence atomic, yet its evaluation consists of multiple statements. The simulator solves this by executing the function in isolation: while executing the function, it ignores all other threads until the function has finished. Since functions do not interact with other threads, their atomicity is actually a non-issue; it is mentioned only because it can be observed in the simulator.

Function calls with constant arguments are themselves constant (Section 2.6.4.3). You may observe that such a call is indeed executed only once.

Procedure calls are statements, and are therefore treated like other composite statements: they are not atomic.

3.2.3 Breakpoints

The `break` command sets a breakpoint. Whenever execution reaches a breakpoint, the simulator stops to allow command input. Breakpoints may be associated with a just a statement in the code, or with a particular instance as well as a statement. However, each statement may have at most one breakpoint attached to it.

By itself, `break` sets a breakpoint at the current statement. This will not be tied to the current instance.

The following form of `break` sets a breakpoint at the statement at or near the indicated position.

```
break "filename" linenumber : position
```

Only the line number is required. If a file name is specified, it must be between quotes. If a position on the line is specified, it must be preceded by a colon.

This third form of `break` sets a breakpoint associated with one particular instance.

```
break instance linenumber : position
```

The given line number and position must refer to a statement that is part of the process definition that is connected to the given instance.

All three of the above commands can be followed by "`if` *condition*", where *condition* can be any expression that evalutes to a boolean and is valid in the reference frame of the breakpoint. Adding this construct will only allow the breakpoint to be tripped when *condition* evaluates to `true`.

A fourth form of `break` sets a breakpoint at the beginning of the specified routine (a process, procedure, or function).

```
break "filename" routine
```

Again, the file name is optional. If the routine is not a top-level routine, you can use a dot to construct a hierarchical name: use `break g.f` to stop at function `f` which is declared inside procedure `g`. A file name need only be specified if otherwise the top-level routine name is not unique.

To remove a breakpoint, use the `clear` command instead of the `break` command with any of the above forms. The `clear` command will clear whatever breakpoint is at the specified statement, regardless of whether it has a condition or is tied to a given instance, and regardless of whether the `clear` command specfifes a single instance (`clear` is not allowed to specify a condition). Once a statement has a breakpoint set to it, the only way to modify the breakpoint is to clear it and set it again.

3.2.4 Tracing

When a process instance is traced, the simulator prints every statement executed by that instance, without stopping (unless there is another reason to stop).

`trace` by itself starts tracing the current process instance. If an instance name is specified, that instance will be traced.

A process that has production rules cannot be traced. Instead, such a process can have a watch placed on any or all of its wires. This will result in the simulator printing out a notfication any time that a watched wire changes value. The command to set a watch is `watch` *expression*, where *expression* is a standard expression that is valid in the current frame of reference and evaluates to a wire (i.e. would be valid as the target of a production rule). Watching the wires of a given instance requires first setting the focus to that instance, and each wire must be listed in a seperate `watch` command.

To stop watching a wire, use `clear watch` *expression*, with the restrictions on *expression* as above. To stop tracing, use `clear trace`, optionally followed by an instance name. `clear step` *instance* can also be used to cancel a pending `step` or `next` for a given process (i.e., execution will not stop when the next statement of that process is reached). The `clear` command cannot undo the effects of the command line arguments `-traceall` and `-watchall`.

3.2.5 Inspecting the state

The `where` command prints the current call stack, i.e., the sequence of nested procedure and function calls that led to the current statement. You can move the focus up or down this stack with the `up` and `down` commands (both take an optional number of steps as argument). The 'up' direction is towards the caller. Moving the focus is useful when you want to print variables, because only variables of the current routine are visible.

You can shift the focus to a different process with the `view` command. (Without argument this simply prints the current statement.) Again, this is useful if you want to print variables of that process. As mentioned before, changing the focus changes the default process for commands like `step`; however, it does not change the actual scheduling of statements.

The `print` command by itself shows all existing threads.

`print` followed by an instance name prints the values of the instance's meta parameters, the other processes the instance is connected to, and the current position in the code for each thread of the process.

`print` followed by a variable or constant name prints the value of that variable or constant. The name must be in scope at the current focus.

3.2.6 Other commands

The `help` command prints a command summary.

The `source` command takes a filename as an argument, and executes the list of commands contained in the file.

The `batch` command switches to non-interactive execution, without stopping for more user input, just like the `-batch` option described in Section 3.1.

Interrupting the program with `ctrl-C` forces the simulator to stop at the next statement, regardless of the instance. This is useful if the program gets stuck in an infinite or very long loop. (In batch mode, `ctrl-C` simply terminates the simulation.)

The simulator also stops immediately after a statement that caused a warning or error message, so that you can inspect variables etc.. The computation cannot continue following an error; instead, the simulator will terminate.

The `quit` command also terminates the simulation.

The commands `fanin` and `fanout` allow for the inspection of which production rules are associated with a given wire. Each requires an expression as an argument that must evaluate to a single wire. `fanin` lists in full all production rules which target a given wire (there will be at most two). `fanout` lists an abbreviated form of each production rule where the given node is part of the rule's guard. Any threads which are waiting for the wire value to change are also listed.

The `critical` command will list the critical timing path of the most recent transition of a given wire. A critical timing path goes from one production rule firing to whatever event caused the guard of the production rule to become true, and repeats this step as long as the event is also a production rule firing. Typically, the critical timing paths of all wires will converge to the same path, and this path will usually repeat itself if the simulation goes on for long enough. The repeating portion of such a path is called the critical cycle.

3.2.7 Instantiation

During instantiation additional commands are available. The purpose of these commands is to replace the large collection of processes produced by a meta process with a single sequential chp process. Of course, this can only be done if the meta process has the optional chp body specified.

The `instantiate` command takes an instance name as an arguments, and completes the instantiation of only this process. If no argument is given, the current process is chosen. In order to function correctly, the instantiate command must specify a process that is listed via the `print` command, in other words one whose parent has completed instantiation. Thus, to instantiate the meta process */cpu/fetch*, we run:

```
instantiate /
instantiate /cpu
instantiate /cpu/fetch
```

The command `continue sequential` is a special command that completes the instantiation phase. However, only meta processes that have already begun instatiation, or that do not have an optional chp body specified will be instantiated. In other words, after specifying which processes need to be instantiated as meta bodies, `chpsim` completes instantition using as few additional processes as possible.

3.3 Built-in procedures

The simulator has a few built-in procedures that can be called from your CHP program to help with testing and debugging. These are always available, unless you have redefined the identifiers.

`procedure step()`

This routine executes the simulator's `step` command. The effect is that the simulator will stop at the next statement of the process that made the call. (Hence, this acts more or less as a permanent breakpoint.)

`procedure print(...)`

This procedure can be passed any number of arguments. It prints its argument values on the standard output, followed by a newline. E.g.,

```
x := 5; y := red;
print("x and y are", x, y);
```

results in

```
/q[0]/r> x and y are 5 red
```

where process instance `/q[0]/r` executed the `print` call.

`procedure show(...)`

This procedure prints its arguments with their values:

```
show(x+1, y)
```

prints

```
/q[0]/r> example.chp[7:3]
        x + 1 = 6
        y = red
```

Note that it also indicates the location of the call.

```
procedure warning(...)
procedure error(...)
```

These cause a warning and error, respectively, with the argument values as message (in the same way as `print()`).

`procedure assert(cond: bool)`

Causes an error if the condition is `false`.

`function random(N: int)`

Return a random integer between 0 and $N - 1$. This is for debugging only — if you want to implement a pseudo random number generator as part of your design, you need to choose an algorithm and implement it in stantard CHP.

`function time()`

Return the current elapsed simulation time. Again, this is for debugging only and is not the solution to timing assumption problems in your code. Even for debugging this should be used with care, since it can easily lead to unintentional deadlock or livelock. If random timing is enabled, the numbers returned by `time()` will be pretty much meaningless.

There is also a built-in type `string`, which is an array of `{0..127}`, i.e., an array of ascii values. However, since no array bounds are specified, you cannot index variables of this type. It is useful for some debugging routines.

3.4 Standard I/O

There is a standard module `stdio.chp` that defines procedures for reading and writing files. This module must be imported with `requires` to use these procedures. Obviously, this module is intended for testing and debugging only.

```
type file
const stdin: file
const stdout: file
type file_err = { ok, eof, no_int };
```

Opening and closing of files is done with

```
procedure fopen(res f: file; name: string; mode: string)
procedure fclose(f: file)
```

The `mode` argument is the same as for the `fopen()` function in C. In particular, `"r"` opens the file for reading, `"w"` opens the file for writing (erasing the existing contents).

Reading is done with

```
procedure read_byte(f: file; res x: {0..255}; res err: file_err)
procedure read_int(f: file; res x: int; res err: file_err)
```

`read_byte` reads a single byte (character). If successful it sets `err` to `ok`; otherwise it sets `err` to `eof` and `x` to 0.

`read_int` skips white space, then reads an integer written in the standard CHP notation (Section 2.8.2). The integer may be immediately preceded by a single minus sign. If no digit is found, `err` is set to `no_int` (and `x` to 0). At the end of the file, `err` is set to `eof`.

Writing is done with

```
procedure write_byte(f: file; x: {0..255})
procedure write_string(f: file; x: string)
procedure write_int(f: file; x: int; base: {2..36})
procedure write(f: file; ...)
```

`write_byte` writes a single byte/character. `write_string` writes a sequence of characters, stopping if a 0 character is encountered (the 0 is optional).

`write_int` writes an integer in the specified base. Decimal integers are written without base, hexadecimal is written with `0x`, and all others are written with the *base*`#` notation.

Finally, `write` writes its arguments in the same way as the `print` procedure of Section 3.3, but without the instance name.