Helena 2.3
User's guide

Sami Evangelista - (Sami [dot] Evangelista [at] lipn.univ-paris13 [dot] fr)

August 29, 2017

Abstract

This manual describes Helena, a High LEvel Nets Analyzer. Helena verifies properties of high level nets by exploring all these possible configurations and reports to the user either a success, i.e. the property holds, either a faulty execution invalidating the specified property. This technique is called model checking, or state space analysis. Helena can also perform more basic tasks like state space exploration in order to report statistics like, e.g., the number of reachable statistics, the structure of the reachability graph.
Helena is a command line oriented tool freely available under the terms of the GNU General Public License. Basic knowledges on Petri nets, high-level Petri nets and model checking are welcome to understand this manual.
The installation on a Linux platform is quite simple and should not raise any problem. The procedure is detailed in file `helena/README`.
This manual is organized as follows. The specification language of Helena is presented in Chapter 1. Chapter 2 is devoted to the use of Helena. Some examples of the distribution are described in Chapter 3. The possibility of interfacing Helena with C code is described in Chapter 4 together with a tutorial illustrating this feature. At last, Chapter 5 is intended to provide some help to the users and some indications on how to use Helena efficiently.
Appendixes contain the syntax summary and the second version of the GNU general public license. An index that references all the construction of the specification language of the tool can be found at the end of the document.

Contents

Chapter 1

Helena specification language

We introduce in this chapter the two specification languages of Helena used to describe high level nets and properties.

1.1 Lexical and syntaxic conventions

We give now some lexical conventions. First of all, it must be noticed that Helena specification language is case sensitive.

1.1.1 Lexical tokens

1.1.1.1 Reserved words

The following words are reserved and can not be used as identifiers:

accept and assert capacity card case constant deadlock default description dom else empty enum epsilon exists false for forall function guard if import in init inhibit let list ltl max min mod mult not of or out pick place pred priority product property proposition range reject return safe set state struct subtype succ sum transition true type until vector while with

Reserved words will systematically appear in a bold font in this document.

1.1.1.2 Identifiers

Places, data types and transitions are examples of Helena constructions that are identified by textual names, or identifiers. Identifiers must start with an alphabetic character and must only contain alpha-numeric characters or the '_' character. They also must, of course, not belong to the list of reserved words.

1.1.1.3 Numerical constants

A valid numeric constant has the form $[0-9][0-9]*$. Octal or hexadecimal notations are not allowed. The maximal constant allowed is system dependant but is 2^{31} on most systems.

1.1.1.4 Comments

Comments are indicated as in the C++ language. Two slashes // start a comment that will end with the current line. /* start a comment which is explicitely ended by */. Comments can not be nested

1.1.2 Preprocessor directives

Helena features some preprocessor directives taken from the C language. However, it must be noticed that preprocessor symbols are not macros. Thus, no expansion takes place. A directive starts with a # at the first column, followed by a list of blanks (possibly empty), and followed by the directive name.

1.1.2.1 Defining and undefining symbols

Symbols can be defined and undefined by directives `define` and `undefine`. The directive must be followed by the symbol name. A symbol name must be a valid identifier (possibly a reserved word).

1.1.2.2 Conditional compilation

Directives `ifdef`, `ifndef`, `else`, and `endif` can be used for conditional compilation. Directive `ifdef` checks if the symbol placed just after it is defined. If it is defined and the `ifdef` has a corresponding `else`, all the lexical tokens between the corresponding `else` and `endif` are ignored. Otherwise, all the lexical tokens are skipped until the corresponding `else` or `endif` is found. Directive `ifndef` has a symmetric behavior.

1.1.3 Conventions

We first start with some conventions that are used in the remainder of this section.

- Terminal symbols of the grammar, i.e., tokens, are placed between quotes, e.g., '**`if`**', '**`transition`**'.
- Non terminal symbols appear in italic between the two characters ⟨ and ⟩, e.g., ⟨*expression*⟩, ⟨*statement*⟩.
- ε denotes the empty list of lexical tokens.
- The non terminal ⟨*name*⟩ stands for any identifier.
- The non terminal ⟨*string*⟩ stands for any string delimited by two characters ¨.
- The non terminal ⟨*number*⟩ stands for any numeric constant.
- Symbols placed between brackets are optional, e.g., [⟨*item*⟩].
- (⟨*item*⟩)* is a sequence (possibly empty) of non terminals ⟨*item*⟩.
- (⟨*item*⟩)$^+$ is a non empty sequence of non terminals ⟨*item*⟩.

1.2 Net specification language

1.2.1 Nets

A net is described by a list of definitions that are the components of the net. Elements that may be defined in a net are data types, constants, functions, places, transitions and state propositions.

⟨*net*⟩ ::= ⟨*net name*⟩
[' **(**' ⟨*net parameter list*⟩ '**)**']
'**{**' (⟨*definition*⟩)* '**}**'
⟨*net name*⟩ ::= ⟨*name*⟩
⟨*definition*⟩ ::= ⟨*type*⟩
| ⟨*constant*⟩
| ⟨*function*⟩
| ⟨*place*⟩
| ⟨*transition*⟩
| ⟨*state proposition*⟩

1.2.2 Net parameters

A net may have parameters such as, e.g., a number of processes. They are interpreted as constants (see Sect. 1.2.4) of the predefined `int` type. The advantage of using parameters is that their values can be changed via the command line when helena is invoked, i.e., without changing the model file. Chapter 2 details how this can be done.

Below is an example of net parameterized by constants `Clients` et `Servers` having default values of 5 and 2 respectively.

```
myNet (Clients := 5, Servers := 2) { ... }
```

⟨*net parameter list*⟩ ::= ⟨*net parameter*⟩
| ⟨*net parameter*⟩ ’**,**’ ⟨*net parameter list*⟩
⟨*net parameter*⟩ ::= ⟨*net parameter name*⟩ ’**:=**’ ⟨*number*⟩
⟨*net parameter name*⟩ ::= ⟨*name*⟩

1.2.3 Types and subtypes

Helena allows the definition of different kinds of data types: integer types (range or modulo types), enumeration types, structured types, vector types, and container types (list or set types). Enumeration and integer types form the family of discrete types whereas other types are said to be composite. This type hierarchy is summarized below.

- Discrete types
  - Integer types
    - Range types
    - Modulo types
  - Enumeration types
- Composite types
  - Structured types
  - Vector types
  - Container types
    - Set types
    - List types

Some data types are predefined. They will be described in the corresponding type description.

⟨*type*⟩ ::= ⟨*type name*⟩ ’**:**’ ⟨*type definition*⟩ ’**;**’
| ⟨*subtype*⟩
⟨*type name*⟩ ::= ⟨*name*⟩
⟨*type definition*⟩ ::= ⟨*range type*⟩
| ⟨*modulo type*⟩
| ⟨*enumeration type*⟩
| ⟨*vector type*⟩
| ⟨*struct type*⟩
| ⟨*list type*⟩
| ⟨*set type*⟩

1.2.3.1 Range type

A range type is an integer type which values belong to a specified range. A range type is defined by specifying the lower bound and the upper bound of the range. Bounds must be numerical expressions, statically evaluable (see Section 1.2.9 for more precisions on statically evaluable expressions). Additionally, the upper bound of the type must be greater or equal to the lower bound. The integer type `int` is a predefined range type. Its definition is system dependant, but on most systems it is defined as follows:
type int : **range** −2147483648 .. 2147483647;

⟨*range type*⟩ ::= ⟨*range*⟩
⟨*range*⟩ ::= ’**range**’ ⟨*expression*⟩ ’**..**’ ⟨*expression*⟩

1.2.3.2 Modulo type

A modulo type is an integer type which values can range from 0 to $m-1$ where m is a specified value called the modulo value. This one must be a numerical expression, statically evaluable, and strictly positive.

⟨*modular type*⟩ ::= ’**mod**’ ⟨*expression*⟩

1.2.3.3 Enumeration type

An enumeration type consists in a non empty collection of distinct enumeration constants. The boolean type bool is a predefined type which is defined as:
type bool: **enum** (**false**, **true**);
It will be referred in the remainder as the boolean type.

⟨*enumeration type*⟩ ::= '**enum**' '**(**' ⟨*enumeration constant*⟩ ('**,**' ⟨*enumeration constant*⟩)* '**)**'
⟨*enumeration constant*⟩ ::= ⟨*name*⟩

1.2.3.4 Vector type

Elements of a vector type (or array type) consist in a set of contiguous elements of the same type, called the element type, that may be accessed by specifying an index, or more precisely a list of indexes. These indexes must be of discrete types. The number of elements in the type is equal to the product of the cardinals of the types which form the index.

⟨*vector type*⟩ ::= '**vector**' '**[**' ⟨*index type list*⟩ '**]**' '**of**' ⟨*type name*⟩
⟨*index type list*⟩ ::= ⟨*type name*⟩ ('**,**' ⟨*type name*⟩)*

1.2.3.5 Structured type

Elements of a structured type consist in contiguous elements called components which may be of different types. Each component is identified by a name. The number of elements in the type is equal to the number of components in the declaration.

⟨*struct type*⟩ ::= '**struct**' '**{**' (⟨*component*⟩)+ '**}**'
⟨*component*⟩ ::= ⟨*type name*⟩ ⟨*component name*⟩ '**;**'
⟨*component name*⟩ ::= ⟨*name*⟩

1.2.3.6 List type

An element of a list type is a list which is defined as a finite sequence of elements of the same type. The same item may appear several times in a list. The following line declares a list type called bool_list .

```
type bool_list: list[nat] of bool with capacity 10;
```

An element of a list of type bool_list has the boolean type. The index type of bool_list is the type between brackets, i.e., nat. Let us note that this type must be discrete. Indeed, we will see later that the elements in a list can be directly accessed via indexes. For example, let us consider a list l of type bool_list . The expression l[0] will denote the first element of the list, l[1] the second one and so on.

The capacity of a list type is the maximal length of any list of this type. The expression provided must be statically evaluable and strictly positive. Here the capacity is 10. This means that a list of type bool_list cannot contain more than 10 booleans.

⟨*list type*⟩ ::= '**list**' '**[**' ⟨*type name*⟩ '**]**' '**of**' ⟨*type name*⟩ '**with**' '**capacity**' ⟨*expression*⟩

1.2.3.7 Set type

Sets are similar to lists except that the same item may not appear several times in a set. Sets are not indexed. Therefore no index type may be provided. As for list types a capacity must be specified.

⟨*set type*⟩ ::= '**set**' '**of**' ⟨*type name*⟩ '**with**' '**capacity**' ⟨*expression*⟩

1.2.3.8 Subtype

Discrete types can be subtyped. Each subtype has a parent (which can also be a subtype) and is defined by a constraint which limit the set of values which belong to the subtype.

A constraint simply consists of a range which must be statically evaluable and which bounds must belong to the parent of the subtype.

Here are some examples of subtypes definitions:

```
type small : range 0..255;

subtype very_small : small range 0..15;
//  subtype very_small is equivalent to small without the values from 16 to 255

type color : enum (chestnut, dark_blue, green, blue, pink, yellow);

subtype light_color : color range green .. yellow;
//  subtype light_color contains values green, blue, pink and yellow

subtype very_light_color : light_color range pink .. yellow;
//  subtype very_light_color contains values pink and yellow
```

The subtypes nat (natural numbers), short (short numbers), and ushort (unsigned short numbers) are predefined subtypes defined as follows:

```
subtype nat    : int range 0 .. int'last;
subtype short  : int range - 32768 .. 32767;
subtype ushort : int range 0 .. 65535;
```

where int'last is the last value of type int (see Section 1.2.9.15).

⟨*subtype*⟩ ::= ⟨*subtype name*⟩ ':' ⟨*parent name*⟩ [⟨*constraint*⟩]
⟨*subtype name*⟩ ::= ⟨*type name*⟩
⟨*parent name*⟩ ::= ⟨*type name*⟩
⟨*constraint*⟩ ::= ⟨*range*⟩

1.2.4 Constants

As in programming languages, constants may be defined at the net level. A constant is defined by using the keyword **constant**. It must necessarily be assigned a value which must have the type of the constant.

⟨*constant*⟩ ::= '**constant**' ⟨*type name*⟩ ⟨*constant name*⟩ ':=' ⟨*expression*⟩ ';'
⟨*constant name*⟩ ::= ⟨*name*⟩

1.2.5 Places

The state of a system modeled by a Petri net is given by the distribution (or marking) of items called tokens upon the places of the net. In high level nets, these tokens are typed. This type is given by the domain of the place. In the class of high level nets of Helena, domains of places are products of basic data types. Tokens are denoted by lists of expressions placed between the two symbols <(and)>. The same token may appear several times in a place. We call multiplicity of a token the number of repetitions of this token in the place. The marking of a place will be noted as the linear combination of the tokens in the place. For instance the marking

```
2*<(1,false)> + 4*<(2,true)>
```

of the place p defined as

```
place p { dom: int * bool; }
```

is the marking which contains 2 occurrences of token <(1, **false**)> and 4 occurrences of token <(2,**true**)>. In others words, the multiplicity of token <(1, **false**)> is 2, the multiplicity of token <(2,**true**)> is 4, and the multiplicity of all the others tokens is 0.

The domain is the only attribute that must be specified by the user. Several optional attributes of the place can also be defined.

⟨*place*⟩ ::= '**place**' ⟨*place name*⟩ '{' ⟨*place domain*⟩ (⟨*place attribute*⟩)* '}'
⟨*place attribute*⟩ ::= ⟨*initial marking*⟩
| ⟨*capacity*⟩
| ⟨*place type*⟩

1.2.5.1 Domain

Domains of places are products of basic data types. The keyword **epsilon** is used to denote the empty product. In this case, the place is equivalent to an ordinary Petri net place. Each time a place is declared a type is implicitly declared which correspond to the domain of the place. This type belongs to a special family of types called token types. A token type is some kind of structured type which elements are given by the domain of the place. A token type is hidden from the user, and can thus not be used. Token types are only used in iterators (see Section 1.2.9.16).

⟨*domain*⟩ ::= '**dom**' '**:**' ⟨*domain definition*⟩ '**;**'
⟨*domain definition*⟩ ::= '**epsilon**'
| ⟨*types product*⟩
⟨*types product*⟩ ::= ⟨*type name*⟩ ('*****' ⟨*type name*⟩)*

1.2.5.2 Initial marking

The initial marking, i.e., before the firing of any transition, of a place can be defined in the place description. Any valid arc label can be used to initialise the marking of a place.

⟨*initial marking*⟩ ::= '**init**' '**:**' ⟨*marking*⟩ '**;**'
⟨*marking*⟩ ::= ⟨*arc label*⟩

1.2.5.3 Capacity

We call the capacity of a place, the maximal multiplicity of any item in this place. In the formal definition of Petri nets, this capacity is infinite. However, the amount of available memory being finite, an implementation must fix this one. The capacity specified must be a numerical expression, statically evaluable and strictly positive. Errors can be raised at the run time if the supplied capacity is not sufficient.

⟨*capacity*⟩ ::= '**capacity**' '**:**' ⟨*expression*⟩ '**;**'

1.2.5.4 Type

A type can be associated to each place of the net. This type specifies the kind of information which is modeled by the place. Several types are allowed:

Process places model the control flow of processes.

Local places model resources local to a process, e.g., a local variable.

Shared places model resources shared by several processes of the system, e.g., a global variable.

Protected places model shared resources which can not concurrently be accessed by the processes, e.g., a global variable which is protected by a lock.

Buffer places model communication buffers between processes.

Ack places are special buffer places. An ack place models an acknowledgment of a synchronous exchange between two processes.

Chapter 5, Section 5.3 gives more details on the use of this feature.

Remark. Since it is usual to name places or transitions process, local, ..., buffer we decided not to include these in the list of reserved words.

⟨*place type*⟩ ::= '**type**' '**:**' ⟨*place type name*⟩ '**;**'
⟨*place type name*⟩ ::= '**process**'
| '**local**'
| '**shared**'
| '**protected**'
| '**buffer**'
| '**ack**'

1.2.6 Transitions

Transitions of a Petri net are active nodes that may change the state of the system, that is, the distribution of tokens in the places. Transitions need some tokens in their input places to be firable and produce tokens in their output places. To further restrain the firability of a transition, inhbitor arcs may be used to specify that some tokens must not be present in a specific place. In high level Petri nets, arcs between places and transitions are labeled by expressions in which variables appear. Thus, a transition is firable for a given instantiation (or binding) of these variables. In Helena, these variables are not explicitely given in the definition of the transition. A variable is implicitly declared if it appears in an arc between the transition and one of its input places, at the top level (i.e., not in a sub expression) and in a non guarded tuple (see Section 1.2.10.2 for more details on tuples). The user may also let Helena bind a variable by picking its value in a specific domain. We call these variables the *free variables* of the transition and they appear in the **pick** section of the transition. Rather than repeating the same expression it is also possible to assign this expression to a *bound variable* declared in the **let** section and then replace the expression by the bound variable wherever it occurs.

The reader may find in Section 5.1 a brief description of the algorithm used by Helena to evaluate transitions and the conditions under which a transition is evaluable.

Transitions in Helena are identified by a name. The description of a transition must specify the input and output places of the transition followed by inhibitor arcs (if any), free variables (if any), bound variables (if any) and finally its attributes: a guard, a priority, a description and a safe attribute.

⟨*transition*⟩ ::= '**transition**' ⟨*transition name*⟩
'**{**' ⟨*transition inputs*⟩
⟨*transition outputs*⟩
[⟨*transition inhibitors*⟩]
[⟨*transition free variables*⟩]
[⟨*transition bound variables*⟩]
(⟨*transition attribute*⟩)* '**}**'
⟨*transition name*⟩ ::= ⟨*name*⟩
⟨*transition inputs*⟩ ::= '**in**' '**{**' (⟨*arc*⟩)* '**}**'
⟨*transition outputs*⟩ ::= '**out**' '**{**' (⟨*arc*⟩)* '**}**'
⟨*transition inhibitors*⟩ ::= '**inhibit**' '**{**' (⟨*arc*⟩)* '**}**'
⟨*transition attribute*⟩ ::= ⟨*transition guard*⟩
| ⟨*transition priority*⟩
| ⟨*transition description*⟩
| ⟨*safe*⟩

1.2.6.1 Arcs

An arc is characterized by the place from which we remove, add or check tokens and by an expression specifying for a given instantiation of the variable of the transition considered tokens.

The description of arc labels appear later in this section.

⟨*arc*⟩ ::= ⟨*place name*⟩ '**:**' ⟨*arc label*⟩ '**;**'

1.2.6.2 Free variables

Free variables must appear in the **pick** section of the transition. The value of a free variable can be picked within

- a discrete type. A range can be specified to avoid considering all possible values of the type.
- or, a container, i.e., a set, a list.

Note that variables of the transitions may appear in the definition of free variables.

When computing enabled bindings at some marking, Helena considers all possible values that can be picked for free variables. For example, the following pick section:

```
pick {
    i in int range 1..5;
    b in bool;
}
```

will potentially multiply by 10 the number of enabled bindings of the corresponding transition since we will generate all the possible values of $(i,b) \in \{1,2,3,4,5\} \times \{\textbf{false}, \textbf{true}\}$.

⟨*transition free variables*⟩ ::= '**pick**' '**{**' (⟨*free variable*⟩)* '**}**'
⟨*free variable*⟩ ::= ⟨*free variable name*⟩ '**in**' ⟨*free variable domain*⟩
⟨*free variable domain*⟩ ::= ⟨*type name*⟩ [⟨*range*⟩]
| ⟨*expression*⟩

1.2.6.3 Bound variables

The **let** section of a transition declaration is provided to declare bound variables that are used to avoid repeating the same expression in the transition. Note that bound variables are always evaluated after input arcs and free variables. Hence, they may not appear in these arcs or in the definition of free variables but can appear at all other places within the transition declaration: in the output or inhibitor arcs, in the guard and in the priority.

For instance, the following declaration:

```
transition t {
  in { q: <( x )>; } out { r: <( f(x) )>; }
  guard: f(x) > 0;
}
```

if equivalent to:

```
transition t {
  in { q: <( x )>; } out { r: <( y )>; }
  let { int y := f(x); }
  guard: y > 0;
}
```

⟨*transition bound variables*⟩ ::= '**let**' '**{**' (⟨*transition bound variable*⟩)* '**}**'
⟨*transition bound variable*⟩ ::= ⟨*type name*⟩ ⟨*variable name*⟩ '**:=**' ⟨*expression*⟩ '**;**'

1.2.6.4 Guard

Transitions can be guarded by a boolean expression. This guard is an additional condition that the variables of the transition must fulfill for a binding to be firable.

⟨*transition guard*⟩ ::= '**guard**' '**:**' ⟨*guard definition*⟩ '**;**'
⟨*guard definition*⟩ ::= ⟨*expression*⟩

1.2.6.5 Safe attribute

Transitions can be declared as safe. A transition binding is safe if it can not be disabled by the firing of any other binding. If a transition is safe all its bindings are considered by Helena as safe. Please report to Chapter 5, Section 5.3 for further details on this feature.

⟨*safe*⟩ ::= '**safe**' '**;**'

1.2.6.6 Priority

Transitions can be prioritized. A valid priority is any expression of type `int`. A transition may not fire for a given binding if another binding (of the same or any other transition) with a greater priority is also enabled. By default, the priority of any binding is 0. It is allowed to refer in a priority expression to all variables of the transition. Moreover the priority system of Helena is dynamic in the sense that the content of a place may also be used to define a priority using e.g., iterators (see Section 1.2.9.16). Hence, the priority of a transition depends on the current system state.

Let us consider the following definitions.

```
transition t {
  in { q: <( x )>; }
  out { r: <( x )>; }
  priority: (x = 0 and p'card > 0) ? 1 : 0;
```

}

Then it follows, that transition t has priority 1 for binding x=0 and if place p is not empty. Otherwise it has priority 0.

⟨*transition priority*⟩ ::= '**priority**' '**:**' ⟨*expression*⟩ '**;**'

1.2.6.7 Description

The default string printed by Helena to describe a transition may be replaced by providing a description. This description consists of a formatting string (following the C conventions) followed by the expressions that may appear in this string. All these expressions must be of discrete types.

Here is an example of transition description:

```
transition t {
   in  { q: <( x )>; }
   out { r: <( x, b )>; }
   pick { b in bool; }
   description: "move␣%d␣from␣q␣to␣r", x;
}
```

⟨*transition description*⟩ ::= '**description**' '**:**' ⟨*string*⟩ ['**,**' ⟨*non empty expression list*⟩] '**;**'

1.2.7 Functions

The user is allowed to define functions which may then appear in arc expressions or in the property to verify. Functions can not have any side effect. They are functions in the mathematical sense: they take some parameters, compute a value and return it. Two alternatives are possible to write the body, i.e., the effect, of a function. First, it can be written in the language provided by Helena that is described bellow. Second, it is possible to import it from a C function, that is, to write it directly in C and then to invoke Helena with option `-L` in order to link the appropriate object files. This second alternative is described in Chapter 4. To allow the definition of mutually recursive functions, the prototype, i.e., name, parameters and return type of the function, must be specified before its own body. The prototype and the body must naturally match. A function becomes visible as soon as its prototype or body is declared.

⟨*function*⟩ ::= ⟨*function declaration*⟩
| ⟨*function body*⟩

⟨*function prototype*⟩ ::= '**function**' ⟨*function name*⟩
'**(**' ⟨*parameters specification*⟩ '**)**' '**->**' ⟨*type name*⟩

⟨*function declaration*⟩ ::= ⟨*function prototype*⟩ '**;**'

⟨*function body*⟩ ::= '**import**' ⟨*function prototype*⟩ '**;**'
| ⟨*function prototype*⟩ ⟨*statement*⟩

⟨*parameters specification*⟩ ::= [⟨*parameter specification*⟩ ('**,**' ⟨*parameter specification*⟩)*]

⟨*parameter specification*⟩ ::= ⟨*type name*⟩ ⟨*parameter name*⟩

⟨*function name*⟩ ::= ⟨*name*⟩

⟨*parameter name*⟩ ::= ⟨*name*⟩

1.2.8 Statements

Helena allows rich possibilities to write functions: conditional statements (if, case), loop statements (for, while), sequence of statements (block), assertions return and assignments statement. Except for the for statement, each has the same semantic as C's corresponding statement.

⟨*statement*⟩ ::= ⟨*assignment*⟩
| ⟨*if statement*⟩
| ⟨*case statement*⟩
| ⟨*while statement*⟩
| ⟨*for statement*⟩
| ⟨*return statement*⟩
| ⟨*assert statement*⟩
| ⟨*block*⟩

1.2.8.1 Assignment

The assignment statement evaluates an expression and assigns its value to a variable. The variable assigned can be a simple variable a structure component a vector component or a list component. Assigned expression must naturally have the same type as the variable.

⟨*assignment*⟩ ::= ⟨*variable*⟩ '**:=**' ⟨*expression*⟩ '**;**'

1.2.8.2 If-then-else

An if statement evaluates a boolean expression, and according to its value executes either the ⟨*true statement*⟩ either the ⟨*false statement*⟩ if it exists.

⟨*if statement*⟩ ::= '**if**' '**(**' ⟨*expression*⟩ '**)**' ⟨*true statement*⟩ ['**else**' ⟨*false statement*⟩]
⟨*true statement*⟩ ::= ⟨*statement*⟩
⟨*false statement*⟩ ::= ⟨*statement*⟩

1.2.8.3 Case

A case statement evaluates an expression and according to the value of this expression chooses an appropriate alternative. A default alternative can be defined. Expressions which appear in the alternatives must have the same type as the evaluated expression and must be statically evaluable. In addition, two different alternatives can not have the same expression.
All possibilities may not be covered. If an alternative is not covered, and the evaluated expression falls into this alternative, the case statement has no effect.

⟨*case statement*⟩ ::= '**case**' '**(**' ⟨*expression*⟩ '**)**' '**{**' (⟨*case alternative*⟩)* [⟨*default alternative*⟩] '**}**'
⟨*case alternative*⟩ ::= ⟨*expression*⟩ '**:**' ⟨*statement*⟩
⟨*default alternative*⟩ ::= '**default**' '**:**' ⟨*statement*⟩

1.2.8.4 Return

A function returns a value by a return statement. An expression is evaluated which correspond to the value returned by the function. The type of this expression must be the return type of the function in which the return statement appear. All statements appearing after the return statement are ignored.

⟨*return statement*⟩ ::= '**return**' ⟨*expression*⟩ '**;**'

1.2.8.5 Block

A block is a list of variables or constants declarations followed by a sequence of statements. A block start with the token { and terminates with the token }. Each variable declared in the block is naturally visible as soon it is declared. Its visibility terminates with the end of the block. A variable declared in the block hides previously declared variables with the same name.

⟨*block*⟩ ::= '**{**' (⟨*declaration*⟩)* (⟨*statement*⟩)$^+$ '**}**'
⟨*declaration*⟩ ::= ⟨*constant declaration*⟩
| ⟨*variable declaration*⟩
⟨*variable declaration*⟩ ::= ⟨*type name*⟩ ⟨*variable name*⟩ ['**:=**' ⟨*expression*⟩] '**;**'
⟨*variable name*⟩ ::= ⟨*name*⟩

1.2.8.6 While

Helena's while statement has exactly the same semantic as C's while statement: as long as the boolean expression is evaluated to **true**, the enclosed statement is executed.

⟨*while statement*⟩ ::= '**while**' '**(**' ⟨*expression*⟩ '**)**' ⟨*statement*⟩

1.2.8.7 For loop

A for statement iterates on all the possible values of some variables called the iteration variables. Iteration variables are implicitly declared with the for statement. Thus, if another variable with the same name has been previously declared, it is hidden in the for statement. In addition, iteration variables are not visible outside the for statement and they are considered as constant in the enclosed statement. The domain of an iteration variable is evaluated once, before entering the loop. Thus, even if the bounds depend on some variable which value is changed in the for, it will have no consequence on the iteration.

We will call iteration scheme a list of iteration variables. These schemes appear in for loops, in iterators (see Section 1.2.9.16) or in front of tuples that label the arcs (see Section 1.2.10.2) of the net. The domain of an iteration variable v can be:

- a **discrete type** t. The iteration variable successively takes all the values of t from the first one to the last one. The iteration may be limited to a specific range. The type of the iteration variable is t.
- a **place** p. In this case, all the tokens present in the place at the current state will be considered. The type of the iteration variable is the token type of p, the different components of the token may then be accessed using the syntax v−>1.
- a **container** c, that is, any expression which has a set or a list type. In this case, all the items in the container will be considered. If the container is a list, it will be traversed from the first element to the last. If it is a set, no assumption can be made on the order of traversal. The type of the iteration variable is the element type of the type of c.

In the case of for loops, place iteration variables are not allowed.

In the following example we define a function compute_sum which computes the sum of some integers contained in a set.

```
type int_set: set of int with capacity 10;
function compute_sum (int_set s) -> int {
    int result := 0;
    for(item in s) result := result + item;
    return result;
}
```

⟨*for statement*⟩ ::= '**for**' '**(**' ⟨*iteration scheme*⟩ '**)**' ⟨*statement*⟩
⟨*iteration scheme*⟩ ::= ⟨*iteration variable*⟩ ['**,**' ⟨*iteration variable*⟩]
⟨*iteration variable*⟩ ::= ⟨*variable name*⟩ '**in**' ⟨*type name*⟩ [⟨*range*⟩]
| ⟨*variable name*⟩ '**in**' ⟨*place name*⟩
| ⟨*variable name*⟩ '**in**' ⟨*expression*⟩

1.2.8.8 Assertion

Assertions can also be placed in functions. The boolean expression associated to an assertion is checked, and if this expression is evaluated to **false** the search is immediately stopped.

⟨*assert statement*⟩ ::= '**assert**' '**:**' ⟨*expression*⟩ '**;**'

1.2.9 Expressions

Helena has been primarily designed to manage the verification of realistic software systems. Thus, we naturally decided to include in Helena a wide range of possibilities concerning expressions.

Some remarks:

- Each expression has a single type that Helena tries to guess at the parsing stage. When Helena has to choose between several types for a given expression, the widest possible type is chosen. For instance, 0 > 2 will be considered as a comparison between two constants of type int. If Helena can not choose between different types, an ambiguity error will be raised.
- Iterators and token components are special kinds of expressions that can only appear in a property specification. These form the family of complex expressions. Attributes related to a place also belong to this family.
- An expression is said to be statically evaluable if no variable and no function call appear in all its sub expressions.

- Expressions may raise errors at the run time such as division by 0, cast error, ... Out of range errors form a special type of error since they are not detected at the evaluation of sub expressions but at the whole expression evaluation. Let us consider for instance the following type declaration:

```
type my_type: range 1 .. 10;
my_type i;
```

The expression i − 1 > 1 will never raise an error even if i has value 1 since the whole expression is correct. However, the statement i := i − 1 will raise an error if i = 1, since the final expression is out of the range of type my_type.

⟨*expression*⟩	::=	'**(**' ⟨*expression*⟩ '**)**'	|	⟨*numerical constant*⟩
	|	⟨*enumeration constant*⟩	|	⟨*variable*⟩
	|	⟨*predecessor-successor operation*⟩	|	⟨*integer operation*⟩
	|	⟨*comparison operation*⟩	|	⟨*boolean operation*⟩
	|	⟨*function call*⟩	|	⟨*cast*⟩
	|	⟨*if-then-else*⟩	|	⟨*structure*⟩
	|	⟨*structure component*⟩	|	⟨*structure assignment*⟩
	|	⟨*vector*⟩	|	⟨*vector component*⟩
	|	⟨*vector assignment*⟩	|	⟨*empty list*⟩
	|	⟨*list*⟩	|	⟨*list component*⟩
	|	⟨*list assignment*⟩	|	⟨*list slice*⟩
	|	⟨*list concatenation*⟩	|	⟨*list membership*⟩
	|	⟨*empty set*⟩	|	⟨*set*⟩
	|	⟨*set membership*⟩	|	⟨*set operation*⟩
	|	⟨*token component*⟩	|	⟨*attribute*⟩
	|	⟨*iterator*⟩		
⟨*expression list*⟩	::=	ε		
	|	⟨*non empty expression list*⟩		
⟨*non empty expression list*⟩	::=	⟨*expression*⟩ ('**,**' ⟨*expression*⟩)*		

1.2.9.1 Numerical and enumeration constants

Constants are the most basic expressions. The resulting expression has the value of the constant. For a numerical constant, the resulting expression is of any integer type (range or modular) which greatest bound (in absolute value) is greater than the constant. For an enumeration constant, the resulting expression is of any enumeration type which includes the constant.

⟨*numerical constant*⟩	::=	⟨*number*⟩
⟨*enumeration constant*⟩	::=	⟨*name*⟩

1.2.9.2 Variable

We call variable any expression which can be assigned a value. It is either a simple variable, either a component of a structure, a vector, or a list. The type of the expression depends on the declaration of the variable. Structure, vector, and list components will be described later in the section.

⟨*variable*⟩	::=	⟨*variable name*⟩
	|	⟨*structure component*⟩
	|	⟨*vector component*⟩
	|	⟨*list component*⟩

1.2.9.3 Predecessor and successor operators

The **succ** and **pred** operators allows to pick the successor and predecessor of a discrete value. Let t be a discrete type and e be an expression of this type.

- If t is a numerical type **pred** e is equivalent to e − 1 and **succ** e is equivalent to e + 1. An error may be raised if t is a range type and the **pred** (resp. **succ**) operator is applied on the first (last) value of the type.
- If t is an enumeration type, the evaluation directly depends on the declaration of the t. For instance, if we consider the following declaration

```
type color: enum(red, green, blue);
```

then we have **succ** red = green = **pred** blue, and **succ** blue = red.

⟨*predecessor-successor operation*⟩ ::= '**pred**' ⟨*expression*⟩
| '**succ**' ⟨*expression*⟩

1.2.9.4 Integer arithmetic

All the classical operators are allowed to perform integer arithmetic: the binary +, −, /, ∗ and % (modulo) operators, and the unary + and − operators. In the case of a binary operator both operands must be of the same type.

Let t be the type of the operand(s). The type of the resulting expression will also be t. Its value depends on t.

- If t is a range type the expression has the conventional meaning. A division by 0 will raise an error.
- If t is a modulo type, the operation is done as for a range type. The resulting value is then normalized as follows. Let m be the modulo value of the type and r be the result of the operation. If r is positive or zero, the value of the resulting expression is $r \mod m$. If it is negative, the result is $r + m \cdot (1 + ((-r)/m)) \mod m$. A division by 0 also raises a run time error.

⟨*integer operation*⟩ ::= ⟨*expression*⟩ '**+**' ⟨*expression*⟩
| ⟨*expression*⟩ '**−**' ⟨*expression*⟩
| ⟨*expression*⟩ '*****' ⟨*expression*⟩
| ⟨*expression*⟩ '**/**' ⟨*expression*⟩
| ⟨*expression*⟩ '**%**' ⟨*expression*⟩
| '**+**' ⟨*expression*⟩
| '**−**' ⟨*expression*⟩

1.2.9.5 Comparison operators

The comparison operators = and != are defined for any type.

- For discrete types, the equality test is straightforward.
- Two structured expressions are equal if all their corresponding components are equal.
- Two vectors are equal if they contain the same elements at the same indexes.
- Two lists are equal if (1) they have the same length and (2) they contain the same elements at the same indexes.
- Two sets are equal if they contain the same elements.

The operators >, >=, < and <= are only defined for discrete and set types.

- For integer types, the comparison is straightforward.
- Enumeration types are ordered according to the way the type has been declared. For instance, let us consider the following declaration: **type** color: **enum**(red, green, blue); It follows from the declaration that red < green < blue.
- For set types, it holds that s1 > s2 if and only if the set s2 is a (strict) subset of the set s1.

A comparison operation has the boolean type.

⟨*comparison operation*⟩ ::= ⟨*expression*⟩ '**=**' ⟨*expression*⟩
| ⟨*expression*⟩ '**!=**' ⟨*expression*⟩
| ⟨*expression*⟩ '**>**' ⟨*expression*⟩
| ⟨*expression*⟩ '**>=**' ⟨*expression*⟩
| ⟨*expression*⟩ '**<**' ⟨*expression*⟩
| ⟨*expression*⟩ '**<=**' ⟨*expression*⟩

1.2.9.6 Boolean logic

Boolean connectors are essential to express complex boolean expressions used, for example, in transition guards. The language includes the classical **or**, **and** and **not** operators. The operand(s) of these operators must have the boolean type, which is also the type of the resulting expression. We thus forbid expression such as 1 **or** 0 which are allowed by the C language.

⟨*boolean operation*⟩ ::= ⟨*expression*⟩ '**or**' ⟨*expression*⟩
| ⟨*expression*⟩ '**and**' ⟨*expression*⟩
| '**not**' ⟨*expression*⟩

1.2.9.7 Function call

Functions previously declared can be called. The syntax of a function call is the same as in the C language. If the function does not take any parameter, () must follow the function name. The type of the expression is the return type of the function. The parameters passed to the function must fit with the function declaration: the number of parameters must be the same in the declaration and in the call, and the type of each parameter must be the same type as in the declaration. The value of the expression is the value returned by the function for the parameters specified.

⟨*function call*⟩ ::= ⟨*function name*⟩ '**(**' ⟨*expression list*⟩ '**)**'

1.2.9.8 Cast

Type casting allows to convert a value of any discrete type to another type. The "source" and "target" types must have at least one value in common. Errors will be raised at the run time if the cast fails, i.e., the value of the casted expression does not belong to the type in which the expression is converted.

⟨*cast*⟩ ::= ⟨*type name*⟩ '**(**' ⟨*expression*⟩ '**)**'

1.2.9.9 If-then-else

The if-then-else expression taken from the C language is allowed in Helena. An if-then-else consists in a boolean condition, and two expressions which must have the same type. The condition is evaluated. If it is evaluated to **true**, the resulting expression is the first expression. Else, it is the second one. The resulting expression has the type of the true and false expressions.

⟨*if-then-else*⟩ ::= ⟨*condition*⟩ '**?**' ⟨*true expression*⟩ '**:**' ⟨*false expression*⟩
⟨*condition*⟩ ::= ⟨*expression*⟩
⟨*true expression*⟩ ::= ⟨*expression*⟩
⟨*false expression*⟩ ::= ⟨*expression*⟩

1.2.9.10 Structures

Structures can be handled in Helena in three different manners.

Firstly, a structure can be constructed by placing all its elements between characters { and }. In this case, the expression is of any structured type which have the same number of components as the structure. In addition, each expression of the structure has to be of the same type as the component declared at the same position in the structured type. For instance if we consider the following type declaration

```
type t:
    struct {
        int i;
        bool b;
    };
```

the expression {−35, **false**} has (at least) type t. Its first component i has value −35, and its second component b has value **false**.

A second possibility to manipulate structures is to access a component of the structure. If we let s be an expression of type t, s.i denotes the value of the component i of s. This expression is only valid if t is a structured type which has a component

named i. If these conditions are met, the type of s.i is the type of component i in the declaration of t, and its value is the value of the component i of s. Please note that we do not allow constructions such as {10, **false** }.i. The structure accessed must be a variable.

At last, structures can be manipulated by using the :: operator. This construction is a shortcut to "assign" an expression to a component of a structure. Let us consider for instance the expression s :: (b := **not** s.b) where the type of s is the type t previously defined. This expression has the same type of s, i.e., t, and has the same value of s except that its component b is replaced by the expression at the right of symbol :=. To be correct such an expression must respect three rules:

1. Expression s must have a structured type t.
2. The replaced component, i.e., before symbol := must be a component of type t.
3. The expression after symbol := must have the same type as the the replaced component in the structured type declaration.

⟨*structure*⟩ ::= '**{**' ⟨*non empty expression list*⟩ '**}**'
⟨*structure component*⟩ ::= ⟨*variable*⟩ '**.**' ⟨*component name*⟩
⟨*structure assignment*⟩ ::= ⟨*expression*⟩ '**::**' '**(**' ⟨*component name*⟩ '**:=**' ⟨*expression*⟩ '**)**'

1.2.9.11 Vectors

The handling of vectors is very similar to the handling of structures. Three basic constructions allow to do this.

Firstly, vectors can be constructed by placing the list of elements in the vector between characters [and]. All the elements in the vector must have the same type. The vector can be of any type which fulfills the two following conditions:

1. The element type of the vector type must be the same as the type of the expressions in the vector expression.
2. The number of elements of the vector type must be greater or equal to the length of the list.

The order of values in a vector is determined from left to right. If the size of the vector, i.e., the length of the expression list in the vector, is less than the number of elements of the vector type, the last expression in the vector is used for the non specified elements.

Let us consider for instance the following vector type declaration:

```
type bool_matrix: vector [bool, bool] of bool;
```

and the following variable declaration

```
constant bool_matrix m1 := [false, false, true];
constant bool_matrix m2 := [false];
```

The vector m1 and m2 will be defined by:

```
m1[false,false] = m1[false,true] = false
m1[true,false]  = m1[true,true]  = true

m2[false,false] = m2[false,true] = m2[true,false] = m2[true,true] = false
```

Let us recall that the predefined type bool is defined as: **type** bool : **enum** (**false**, **true**);

Secondly, a specific element of a vector can be accessed. Let us consider the type bool_matrix previously defined and m a variable of this type. The expression m[**false**, **false**] is a boolean expression, i.e., the type of the elements of vector type bool_matrix, and its value is the value of the element of m which index is [**false** , **false**].

At last, an element of a vector can be assigned an expression. The syntax is close to the syntax of a structure assignment. Instead of specifying the name of a component, an index of the vector is supplied. For instance, the expression m :: ([**true**, **false**] := m[**false**,**true**]) has the same type as m. Its value is the vector m in which the element at index [**true**, **false**] has been replaced by the element at index [**false** ,**true**].

⟨*vector*⟩ ::= '**[**' ⟨*non empty expression list*⟩ '**]**'
⟨*vector component*⟩ ::= ⟨*variable*⟩ '**[**' ⟨*non empty expression list*⟩ '**]**'
⟨*vector assignment*⟩ ::= ⟨*expression*⟩ '**::**' '**(**' '**[**' ⟨*non empty expression list*⟩ '**]**' '**:=**' ⟨*expression*⟩ '**)**'

1.2.9.12 Lists

Lists can be handled in the same way as vectors and structures. We will illustrate the different possibilities with the help of the type int_list defined below.

```
type int_list: list [nat] of int with capacity 10;
```

First, we can construct the empty list, i.e., that does not contain any element, with the help of the keyword **empty** as in the example below.

```
constant int_list empty_list := empty;
```

A list can also be constructed by placing all its elements between two '|' characters. For example:

```
constant int_list l := |1, 2, 3, 4, 5|;
```

The list l is a list of five integers. Its element at the first index is the constant 1. At the second index there is the constant 2, and so on.

The elements of a list can be accessed via their indexes, as for vectors. For example, if we consider the list l previously defined, l[0] will be the first element of the list, i.e., 1, l[1] the second element of the list, and so on. An error will be raised if we attempt to access an element at an index that does not exist, e.g., l[5].

Let us note that the classification starts at 0 since the index type of int_list is nat. If this index type was, for example, the type short the first element would naturally have index -32768.

It is possible to "assign" a value to an element of a list at a specified index. The syntax is the same as for vectors except that only one index can be specified. If we consider the list l previously defined then the expression l :: ([2] := 10) has type int_list . Its value is the list l in which the element at index 2 has been replaced by the constant 10. In other words, l :: ([2] := 10) = |1, 2, 10, 4, 5|.

An error is raised if an attempt is made to assign a value to an element at an index which does not exist, e.g, l :: ([5] := 10) is an error.

Another possibility is to extract a slice, i.e., a sub-list, from a list. To do so it is necessary to provide the indexes of the first and the last elements desired. The resulting slice consists of the sub-list which contains all the elements of the original one from the first index to the last index. As an example let us considered the list l previously defined. Then l[1 .. 3] is equivalent to the list |l[1], l[2], l[3]|. It is important to notice that in the resulting list the index of the first element will still be 0, i.e., the first value of the index type nat and not 1.

To be correct a slice must be such that both the first and the last index must be less than (or equal to) the index of the last element of the list. If the index of the last element is less than the index of the first one the resulting list is the empty list.

Two lists may be concatenated using the binary & operator. One of its operands must have a list type l which is also the type of the resulting expression. The other operand must have the type l or the element type of l. Here are some examples of concatenations with the value of the resulting expression.

```
|1, 2, 3, 4| & 5 = |1, 2, 3, 4, 5|
1 & |1, 2, 3| & 5 = |1, 1, 2, 3, 5|
|1, 2, 3| & |4, 5, 6| = |1, 2, 3, 4, 5, 6|
```

The concatenation of two lists may raise an error if the number of elements of the resulting list exceeds the capacity of the corresponding list type.

It is possible to check if an item belongs to a list by using the **in** operator. For instance:

```
5 in |1, 2, 5| = true
2 in |1, 5, 2, 5| = true
3 in |1, 5| = false
```

The resulting expression has the bool type. The expression e **in** l is evaluated to **true**, if there is an index i such that l[i] = e or **false** otherwise.

Lastly, list have some attribute that may be useful. We can for example extract the prefix or the suffix of a list or select the first element of a list. List attributes are described later in Section 1.2.9.15.

⟨*empty list*⟩ ::= '**empty**'
⟨*list*⟩ ::= '**|**' ⟨*non empty expression list*⟩ '**|**'
⟨*list component*⟩ ::= ⟨*variable*⟩ '**[**' ⟨*expression*⟩ '**]**'
⟨*list assignment*⟩ ::= ⟨*expression*⟩ '**::**' '**(**' '**[**' ⟨*expression*⟩ '**]**' '**:=**' ⟨*expression*⟩ '**)**'
⟨*list slice*⟩ ::= ⟨*expression*⟩ '**[**' ⟨*expression*⟩ '**..**' ⟨*expression*⟩ '**]**'
⟨*list concatenation*⟩ ::= ⟨*expression*⟩ '**&**' ⟨*expression*⟩
⟨*list membership*⟩ ::= ⟨*expression*⟩ '**in**' ⟨*expression*⟩

1.2.9.13 Sets

Some constructions are common for sets and lists. For example it is possible to construct an empty set with the help of the **empty** keyword. A set can also be defined by placing all its elements between two '|' characters.

Set membership is realized through the **in** operator as for lists.

The **or**, **and** and − operators may be used to compute the union, intersection, and differences of two sets. One of the operands of these operators must be a set while the other can be a a set of the same type or an expression of the element type of the set. A run time error will naturally be raised if the cardinal of the resulting set exceeds the capacity of its type.

At last lists and sets have many attributes in common. You will find a complete list of these attributes at Section 1.2.9.15.

Let us examine some examples that illustrate the use of sets.

```
type int_set: set of int with capacity 10;
constant int_set s1 := |1, 2, 3|;
constant int_set s2 := |1, 2, 3| or |2, 3, 4|;   //  s2 = |1, 2, 3, 4|
constant int_set s3 := |1, 2, 3| and |2, 3, 4|;  //  s3 = |2, 3|
constant int_set s4 := 0 or |1, 2| or 5;         //  s4 = |0, 1, 2, 5|
constant int_set s5 := |1, 2, 3| - |1, 2|;       //  s5 = |3|
constant int_set s6 := |1, 2, 3| - 1;            //  s6 = |2, 3|
constant bool b := 3 in s1;                      //  b = true
constant bool c := 4 in s1;                      //  c = false
```

⟨*empty set*⟩ ::= '**empty**'
⟨*set*⟩ ::= '**|**' ⟨*non empty expression list*⟩ '**|**'
⟨*set membership*⟩ ::= ⟨*expression*⟩ '**in**' ⟨*expression*⟩
⟨*set operation*⟩ ::= ⟨*expression*⟩ '**or**' ⟨*expression*⟩
| ⟨*expression*⟩ '**and**' ⟨*expression*⟩
| ⟨*expression*⟩ '**-**' ⟨*expression*⟩

1.2.9.14 Token component

Iterator expressions can be used to iterate on the tokens present in a place at the current marking. To check complex conditions on these tokens, components of tokens can be accessed by specifying the name of the token variable followed by symbol −> and the number of the accessed component of the token type.
For instance, let us consider the following place definition:
place p { **dom**: int ∗ bool ∗ int ; }
If the type of variable t is the token type of place p, then t−>1, t−>2 and t−>3 are three valid expressions. t−>1, and t−>3 have both type int, while t−>2 has type bool. t−>4 is not a valid expression since the domain of place p is a product of three items.

⟨*token component*⟩ ::= ⟨*token*⟩ '**->**' ⟨*component number*⟩
⟨*token*⟩ ::= ⟨*variable name*⟩
⟨*component number*⟩ ::= ⟨*number*⟩

1.2.9.15 Attributes

Some elements have attributes that can be used in expressions. The syntax of an attribute is inherited from the Ada syntax: it consists of the element (e.g., a type name, a place name) followed by the character ' and the name of the attribute. Table 1.1 summarizes the possible attributes and their meaning.

There are several categories of attributes.

Type attributes Let t be a discrete type of the net.

t'first and t'last correspond to the first and the last value of t. Their value depends on the definition of t:

- If t is a range type, defined as **range** low..up, then t'first =low and t'last =up.
- If t is a mod type, defined as **mod** N, then t'first =0, and t'last =N−1
- If t is an enumeration type, t'first it is the first element in the list which defines t, and t'last it is the last element of the list.

t'**card** is the cardinal of type t. This expression can have any numerical type. Its value depends on the definition of t.

- If t is a range type defined as **range** low..up, then t'**card**=1+up−low.
- If t is a mod type defined as **mod** N, it is N.
- If t is an enumeration type, it is the length of the list which defines t.

Some examples:

- bool'first =**false**
- bool'last =**true**
- bool'**card**=2

Place attributes Let p be a place of the net.

p'**card** is the number of tokens in place p at the current state. This expression can have any numerical type.

p'**mult** is the cumulated multiplicities of the tokens in place p at the current state. This expression can have any numerical type.

For instance, if place p contains the following tokens at the current state:

2*<(2,**true**)> + <(3,**false**)> + 4*<(5,**false**)>

Then we have p'**card** = 3 and p'**mult** = 7. Indeed, there are three different tokens in place p at the current state and the sum of the multiplicities of these three tokens is 7 (2 + 1 + 4).

Container attributes Let c be a container, i.e., an expression of which the type is a list type or a set type.

c'size is the size of c, i.e., the number of elements in this container. This expression can have any numerical type.

c'**capacity** is the value of the capacity of the type of c. This expression can have any numerical type.

c'space is the remaining space in container c, i.e., the capacity of the type of c minus the number of elements in c. This expression can have any numerical type.

c'full is a boolean expression which value is **true** if the container is full (i.e., the number of elements in it is equal to the capacity of the type of c), or false otherwise.

c'**empty** is a boolean expression which value is **true** if the container is empty (i.e., it does not contain any element), or false otherwise.

The following declarations illustrate the use of these attributes.

```
type int_set: set of int with capacity 10;
constant int_set s1 := empty;
constant int_set s2 := |1, 5, 12, −5|;
constant int_set s3 := |1, 2, 4, 8, 16, 32, 64, 128, 256, 512|;

constant int i1 := s2'size;        //  i1 = 4
constant int i2 := s1'capacity;    //  i2 = 10
constant int i3 := s2'space;       //  i3 = 10 − 4 = 6
constant bool b1 := s3'full;       //  b1 = true
constant bool b2 := s1'empty;      //  b2 = true
```

Table 1.1: Summary of the possible attributes

Expression	Valid if	Interpretation
e'**capacity**	e is a container	the capacity of e
e'**card**	e is a discrete type e is a place	the cardinal of e the number of distinct tokens in e
e'**empty**	e is a container	e is empty
e' first	e is a discrete type e is a list	the first value of e the first element of e
e' first_index	e is a list	the index of the first element of e
e' prefix	e is a list	the first elements of e
e' full	e is a container	the capacity of e is reached
e' last	e is a discrete type e is a list	the last value of e the last element e
e' last_index	e is a list	the index of the last element of e
e' suffix	e is a list	the last elements of e
e' length	e is a list	the length of e
e'**mult**	e is a place	the cumulated multiplicities of the tokens in e
e'space	e is a container	the remaining space in e

List attributes Let l be a list, i.e., an expression of which the type is a list type t defined as

```
type t: list[index_type] of element_type with capacity N;
```

l' first is the first element of l. The type of this expression is element_type. A run-time error is raised if l is empty.

l' last is the last element of l. The type of this expression is element_type. A run-time error is raised if l is empty.

l' prefix is the list which consists of the first elements of l, i.e., the list l from which we remove the last element. The type of this expression is t. A run-time error is raised if l is empty.

l' suffix is the list which consists of the last elements of l, i.e., the list l from which we remove the first element. The type of this expression is t. A run-time error is raised if l is empty.

l' first_index is the index of the first element of the list. It always hold that l' first_index = index_type' first . The type of this expression is index_type.

l' last_index is the index of the last element of the list. The type of this expression is index_type. A run-time error is raised if l is empty.

Let us have a look at some examples.

```
type int_list: list[nat] of int with capacity 10;
constant int_list l := |3, 5, 12, -5, 10|;

constant int i1 := l'first;          //  i1 = 3
constant int i2 := l'last;           //  i2 = 10
constant int_list l1 := l'prefix;    //  l1 = |3, 5, 12, -5|
constant int_list l2 := l'suffix;    //  l2 = |5, 12, -5, 10|
constant nat n1 := l'first_index;    //  n1 = 0
constant nat n2 := l'last_index;     //  n2 = 4
```

⟨*attribute*⟩	::=	⟨*type name*⟩ ''' ⟨*type attribute*⟩		
	|	⟨*place name*⟩ ''' ⟨*place attribute*⟩		
	|	⟨*expression*⟩ ''' ⟨*container attribute*⟩		
	|	⟨*expression*⟩ ''' ⟨*list attribute*⟩		
⟨*type attribute*⟩	::=	**'first'**	**'last'**	**'card'**
⟨*place attribute*⟩	::=	**'card'**	**'mult'**	
⟨*container attribute*⟩	::=	**'full'**	**'empty'**	**'capacity'**
	|	**'size'**	**'space'**	
⟨*list attribute*⟩	::=	**'first'**	**'first_index'**	**'prefix'**
	|	**'last'**	**'last_index'**	**'suffix'**

1.2.9.16 Iterator

Iterators are provided to express properties that must be verified by the net. The general syntax of an iterator is the following:

```
iterator(iteration-scheme | condition : expression)
```

For more precisions on the notion of iteration scheme, please refer to Section 1.2.8.7.

An iterator considers all the tokens present in a place at the current state (if the iteration domain is a place) or all the values of a discrete type (if the iteration domain is a type) or all the items present in a container (if the iteration domain is a container) and computes a value. A condition, i.e., a boolean expression, can be specified to limit the iteration to the values which satisfy the condition. The evaluation of an iterator depends on its type. Different types iterators are provided.

- Iterator **forall** checks that the expression is evaluated to **true** for all the possible iterations. The expression provided must have type bool and so is the type of the resulting expression.
- Iterator **exists** checks that the expression is evaluated to **true** for at least one iteration. No expression must be provided in the iterator. The resulting expression has type bool.
- Iterators **min** and **max** compute respectively a minimal and a maximal value. The expression inside the iterator can have any discrete type. This type is also the type of the resulting expression. If the set over which the variable iterates is empty, the resulting expression has an undefined value.
- Iterators **sum** and **product** compute respectively a sum and a product. The expression inside the iterator can have any numerical type which is also the type of the resulting expression.
- Iterator **card** computes the number of iterations that fulfill a condition. No expression must be provided in the iterator. The resulting expression can have any numerical type.
- Iterator **mult** is only valid if a single iteration variable is provided and if its domain is a place. It computes the sum of the multiplicities of the tokens in this place which fulfill a condition. No expression must be provided in the iterator. The resulting expression can have any numerical type.

We illustrate the use of these iterators on several examples. Let t be the type and p be the place defined by:

```
type t: range 1..10;
place p { dom: t * bool; }
```

The marking of place p at the current state is given by the following tokens distribution:

```
<(1,true)> + 2*<(2,false)> + <(2,true)> + 3*<(4,true)> + 4*<(8,false)>
```

Let us detail the evaluation of some iterators.

- **exists**(t **in** p) = **true**. Indeed, there are five tokens in place p.
- **forall**(t **in** p | t->2 : t->1 < 5) = **true**. All the tokens in place p which have their second component equal to **true** have their first component strictly less than 5.
- **card**(t **in** p | **not** t->2) = 2. There are 2 tokens in place p which have their second component equal to **false**.
- **mult**(t **in** p | t->2) = 5. The cumulated multiplicities of the tokens in p which have their second component equal to **true** is 5. These tokens are <(1,**true**)>, <(2,**true**)> and <(4,**true**)>.

- **min**(t **in** p : t−>1) = 1, **max**(t **in** p : t−>1) = 8. The minimal and maximal values for the first component of all the tokens in place p are 1 (for token <(1,**true**)>) and 8 (for token <(8, **false**)>).
- **sum**(t **in** p | t−>2 : t−>1) = 7. The sum of the first components of the tokens which have their second component equal to **true** is 7 (1 + 2 + 4).
- **product**(t **in** p | **not** t−>2 : t−>1) = 16. The product of the first components of the tokens which have their second component equal to **false** is 16 (8 · 2).
- **exists** (i **in** t | **forall** (b **in** bool : **card**(t **in** p | t−>1=i **and** t−>2=b) = 1)). This expression can be read as follows: there is an element i of type t which is such that the tokens <(i , **false**)> and <(i ,**true**)> are present in place p. This holds for i=2.

⟨*iterator*⟩ ::= ⟨*iterator type*⟩ '**(**' ⟨*iteration scheme*⟩ [⟨*iterator condition*⟩] [⟨*iterator expression*⟩] '**)**'
⟨*iterator type*⟩ ::= '**forall**' | '**exists**' | '**card**' | '**mult**' | '**min**' | '**max**' | '**sum**' | '**product**'
⟨*iterator condition*⟩ ::= '**|**' ⟨*expression*⟩
⟨*iterator expression*⟩ ::= '**:**' ⟨*expression*⟩

1.2.10 Arc labels

In high level Petri nets, arcs between places and transitions are labeled by expressions indicating, for a given instantiation of the variables of the transition, the tokens consumed or produced by the firing. In the high level nets supported by Helena, these expressions are linear combinations of simpler ones called tuples of expressions or, more simply, tuples. Tuples are lists of expressions placed between two tokens <(and)>. For instance given two variables x,y, the expression

```
2*<(x,y)> + <(y,0)>
```

produces two tokens of type <(0,1)> and one token of type <(1,0)> for the instantiation x=0, y=1.

1.2.10.1 Arc expression

As stated previously, an expression labeling an arc of the net is a linear combination of tuples, or a sum of complex tuples. Both are defined just afterwards.

⟨*arc label*⟩ ::= ⟨*complex tuple*⟩ ('**+**' ⟨*complex tuple*⟩)*

1.2.10.2 Tuples

Tuples are basic components of arc label. They may be guarded by a boolean expression. If this expression is evaluated to **true** for the firing instantiation, the corresponding tokens are normally produced. Otherwise, if the condition does not hold, tokens are not produced. A condition can be specified by placing a construction **if** (cond) before the tuple. For instance, let us consider the tuple **if** (x > 0) <(x)>. If x <= 0, the tuple does not produce any token, else it produces a single token <(x)>.

Helena also provides the following syntactical facility: instead of writing

```
<(x,false,1)> + <(x,false,2)> + <(x,true,1)> + <(x,true,2)>
```

one can prefix the tuple with some iteration scheme (see Section 1.2.8.7):

```
for(b in bool, i in int range 1..2) <(x,b,i)>
```

Note that we do not allow the iteration variable to loop over places or containers. Only discrete iteration variables are allowed. In addition, if a range is specified for an iteration variable then it must necessarily be evaluable statically, i.e., the two bounds must be evaluable statically.

The two possibilities can also be combined. For instance:

```
for(b in bool, i in int range 1..2) if(x != i or b) <(x,b,i)>
```

is a valid tuple.

If the domain of the corresponding place is the empty product **epsilon**, the only possible tuple is **epsilon**. The expressions list in the tuple must correspond to the domain of the corresponding place.

A factor may appear before the tuple to denote the number of tokens produced by this tuple. This one must be a numeric expression, statically evaluable and positive.

⟨*complex tuple*⟩ ::= [⟨*tuple for*⟩] [⟨*tuple guard*⟩] [⟨*tuple factor*⟩] ⟨*tuple*⟩
⟨*tuple for*⟩ ::= '**for**' '**(**' ⟨*iteration scheme*⟩ '**)**'
⟨*tuple guard*⟩ ::= '**if**' '**(**' ⟨*expression*⟩ '**)**'
⟨*tuple factor*⟩ ::= ⟨*expression*⟩ '*****'
⟨*tuple*⟩ ::= '**<(**' ⟨*non empty expression list*⟩ '**)>**'
| '**epsilon**'

1.2.11 State propositions

Most properties are expressed by means of state propositions. A state proposition is a boolean expression that usually refers to the current state using iterators (see Section 1.2.9.16). A proposition has a name meant to be used in properties and simply consists of an expression.

⟨*state proposition*⟩ ::= '**proposition**' ⟨*state proposition name*⟩ '**:**' ⟨*expression*⟩ '**;**'
⟨*state proposition name*⟩ ::= ⟨*name*⟩

1.3 Property specification language

The property specification simply consists of a list of properties. Helena currently supports two types of property: state properties and temporal properties expressed in the linear time temporal logic (LTL).

⟨*property specification*⟩ ::= (⟨*property*⟩)*
⟨*property*⟩ ::= ⟨*state property*⟩
| ⟨*temporal property*⟩

1.3.1 State properties

State properties form the most basic type of property Helena can analyse. A state property must hold in all the reachable states of the system.

A state property consists of the keyword **reject** followed by the description of the states that are rejected during the search. When a state is rejected by Helena the search stops and Helena displays the trace, i.e., the sequence of transition bindings, which leads from the initial state to the rejected, i.e., faulty, state. The keyword **reject** can be followed by:

1. the keyword **deadlock**. In this case, Helena rejects states in which no transition is enabled.
2. a state proposition name. Helena rejects any state in which the proposition holds.

Accept clauses can be used to limit the rejection of states which do satisfy the **reject** predicate: if a state verifies at least one of the accept clauses, then it is considered that the state property holds at this state.

For example, to express that no deadlock can occur we can write:

```
state property not_dead:
  reject deadlock;
```

Now to specify that the termination state is a valid "deadlock state", we can write:

```
state property not_dead2:
  reject deadlock;
  accept valid_termination;
```

where state proposition valid_termination has been defined in the net specification as:

```
proposition valid_termination: termination'card > 0;
```

⟨*state property*⟩ ::= '**state**' '**property**' ⟨*property name*⟩ '**:**' ⟨*state property definition*⟩
⟨*property name*⟩ ::= ⟨*name*⟩
⟨*state property definition*⟩ ::= ⟨*reject clause*⟩ (⟨*accept clause*⟩)*
⟨*reject clause*⟩ ::= '**reject**' ⟨*predicate*⟩ '**;**'
⟨*accept clause*⟩ ::= '**accept**' ⟨*predicate*⟩ '**;**'
⟨*predicate*⟩ ::= '**deadlock**'
| ⟨*state proposition name*⟩

1.3.2 Temporal properties

Helena can also analyse LTL properties. An LTL property is defined by a name and a temporal expression. To be verified, all maximal executable sequences must match the expression specified. A temporal expression is built using state propositions. Besides usual boolean operators, a temporal expression can also include the following temporal operators: [] ("globally"), <> ("finally") and **until**.

For instance the following temporal property expresses that once state proposition p holds it holds in all subsequent states of the sequence.

```
ltl property prop: [] (not P or [] P);
```

⟨*temporal property*⟩ ::= '**ltl**' '**property**' ⟨*property name*⟩ '**:**' ⟨*temporal expression*⟩ '**;**'
⟨*temporal expression*⟩ ::= '**(**' ⟨*temporal expression*⟩ '**)**'
| '**true**'
| '**false**'
| ⟨*state proposition name*⟩
| '**not**' ⟨*temporal expression*⟩
| ⟨*temporal expression*⟩ '**or**' ⟨*temporal expression*⟩
| ⟨*temporal expression*⟩ '**and**' ⟨*temporal expression*⟩
| '**[]**' ⟨*temporal expression*⟩
| '**<>**' ⟨*temporal expression*⟩
| ⟨*temporal expression*⟩ '**until**' ⟨*temporal expression*⟩

Chapter 2

Using Helena

2.1 Invoking Helena

Using Helena consists of writing the description of the high level net in a file (called the net file thereafter), e.g., `my-net.lna`, and the properties expressed on this net in a second file (called the property file thereafter), e.g., `my-net.prop.lna`, and to invoke Helena on this file. The command line of Helena has the following form:

```
helena [option ... option] my-net.lna
```

When invoked, Helena proceeds as follows:

1. If the net described in file `my-net.lna` is `my-net`, the directory `~/.helena/models/lna/my-net` is created.
2. A set of C source files and a Makefile are put in directory `~/.helena/models/lna/my-net/src`.
3. These files are compiled and an executable is created which corresponds to the actual model checker for the specific net.
4. The compiled executable is launched.
5. Once the search is finished, a report is displayed on the standard output. If a property was checked, this report indicates whether the desired property is verified or not. In the second case, a path leading from the initial marking to the faulty marking is displayed.

Each option has a short form preceded by a single `-` and a long form preceded by `--`. Most options take an argument. To invoke Helena on a file `my-net.lna` with an option `opt` having argument `arg`, simply type `helena -opt=arg my-net.lna`. In addition, let us note that the options are interpreted in the order in which they are found. Hence, in case of conflicting options, the last ones will prevail.

2.1.1 General options

`-h`, `--help`
: Prints a help message and exit.

`-V`, `--version`
: Prints the version number and exit.

`-v`, `--verbose`
: This option activates the verbose mode. Helena prints a message at each step.

`-N`, `--action=ACTION`
: This indicates the action performed by Helena on the model. `ACTION` must have one of the following values:

  - `EXPLORE` — Helena explores the state space of the model and then prints some statistics. This is the default.

- SIMULATE — Starts helena in interactive simulation mode. You can then navigate through the reachability graph of the model by executing transitions, undoing transitions,... A simple command language is provided. Once the simulation is started, type help to see the list of commands.
- BUILD-GRAPH — Helena builds the reachability graph of the model using a search algorithm DELTA-DDD (see option algo) and store it on disk. This graph can then be analyzed using the helena-graph tool (see Section 2.3.2).
- CHECK-prop — Helena checks whether or not property prop (which must be a property defined in the model file) is verified.

-p, --property-file=FILE-NAME
File FILE-NAME contains the definition of the property to check specified with option --action=CHECK-prop. By default, if the input file of the model is model.lna, Helena will try opening file model.prop.lna to look for the property definition.

-md, --model-directory=DIRECTORY
All generated files such as source files are put in directory DIRECTORY instead of ~/.helena/models/lna/my-net.

2.1.2 Search options

-A=ALGO-TYPE, --algo=ALGO-TYPE
This option is used to indicate to Helena the type of search that must be used to explore the state space. Five search modes are available. They correspond to the three following possible values for parameter ALGO-TYPE:

- DFS — The state space is explored using a depth-first search. This is the default.
- BFS — The state space is explored using a breadth-first search.
- FRONTIER — The state space is explored using a breadth-first search but only the states of the current level are kept in memory. This algorithm will not terminate if the state space contains cycles.
- DELTA-DDD — The state space is explored using a parallel breadth-first search based on state compression.
- RWALK — Helena explores the state space using a random walk. The principle is to randomly select at each state an enabled transition, execute it and reiterate this process. The walk is reinitiated each time a deadlock state is met. If no limit is specified (see Section 2.1.4) the search will last forever.

-t=N, --hash-size=N
Set the size of the hash table which stores the set of reachable markings to 2^N. The default value is 22.

-W=N, --workers=N
Specify the number of working threads that will perform the search. This option is only usable if the search algorithm is DELTA-DDD or RWALK.

-cs=N, --candidate-set-size=N
Set the cache size of algorithm DELTA-DDD. 100 000 is the default value. Increasing it may consume more memory but can fasten the search.

2.1.3 Reduction techniques

-H, --hash-compaction
The hash compaction storage method is used. Its principle is to only store a hash signature of each visited state. In case of hash conflict, Helena will not necessarily explore the whole state space and may report that no error has been found whereas one could exist.

-P, --partial-order
Specify if partial order reductions are applied during the search. Partial order methods try to alleviate the state explosion problem by limiting the exploration of multiple paths that are redundant with respect to the desired property. This causes some states to be never explored during the search. The reductions done depend on the property verified. If there is no property checked, the reduction done only preserves the existence of deadlock states.

-D, --delta
This option can activate the delta markings storage method to store the state space. The basic idea is store a large set of states compactly by only storing differences with respect to others states of the state space. This technique can greatly reduce the memory needed to store the state space, but will also slow the search. Option -K can be set to influence this.

`-K=N`, `--k-delta=N`
: This is the parameter of the delta markings storage method. Consequently, this has no effect if `D=0` is passed to Helena. The higher this parameter is, the less memory will be needed to store the state space, and the slower the search will be.

`-S`, `--state-caching`
: The principle of state caching is to only store a subset of visited states in such a way that termination is still guaranteed. Other visited states are forgotten. This technique can be very efficient but can also considerably increase the execution time by revisiting forgotten states.

`-s`, `--cache-size=N`
: When using state caching Helena can keep some visited states in a cache in order to limit redundant state revisits. This option is used to specify the size of this cache.

2.1.4 Search limits

`-ml=N`, `--memory-limit=N`
: The memory used by Helena is limited to `N`% of the available RAM. When this limit is reached the search stops as soon as possible.

`-tl=N`, `--time-limit=N`
: The search time is limited to `N` seconds. When this limit is reached the search stops as soon as possible.

`-sl=N`, `--state-limit=N`
: As soon as `N` states have been visited the search is stopped as soon as possible.

2.1.5 Model options

`-d=SYMBOL-NAME`, `--define=SYMBOL-NAME`
: Define preprocessor symbol `SYMBOL-NAME`.

`-a=N`, `--capacity=N`
: The default capacity of places is set to `N`.

`-r`, `--run-time-checks`
: Activate run time checks such as: division by 0, expressions out of range, capacity of places exceeded, ... If this option is not activated, and such an error occurs during the analysis, Helena may either crash, either produce wrong results.

`-L`, `--link=OBJECT-FILE`
: Add file `OBJECT-FILE` to the files linked by Helena when compiling the net. Please consult Chapter 4 for further help on this option.

`-m=p=i`, `--parameter=p=i`
: This gives value `i` (an integer) to net parameter `p`.

2.1.6 Output

`-o=FILE-NAME`, `--report-file=FILE-NAME`
: An XML report file is created by Helena once the search terminated. It contains some informations such as the result of the search, or some statistics. Please report to Section 2.2 for further indications on this report.

`-tr`, `--trace-type`
: Specify the type of trace displayed:

  - `FULL` — The full trace is displayed.
  - `EVENTS` — Only the sequence of events, the initial and the final faulty states are displayed. Intermediary states are not displayed.
  - `STATE` — Only the faulty state reached is displayed. No information on how this state can be reached is therefore available.

2.2 The output report

Once the search terminated, Helena prints a report to the standard output. This section details the structure of this report.

2.2.1 Structure of the report

This report is composed of four distinct parts: the information report, the search report, the trace report and the statistics report.

2.2.1.1 The information report

This first report contain general informations, such as the name of the net verified and the date of analysis.

2.2.1.2 The search report

This report contains various informations on the search such as the termination state of the search or the options enabled, e.g., partial order. The search can terminate in different ways: the property is verified, the property does not hold, the search ran out of memory, ...

2.2.1.3 The trace report

When the property specified does not hold, Helena reports in a trace report the faulty execution discovered. In the case of state property, this execution consists of a sequence of states $s_0, \ldots, s_n$ such that s_0 is the initial state and s_n is the faulty state reached by Helena. For LTL properties this consists of an infinite execution violating the temporal property.

2.2.1.4 The statistics report

The statistics report is the last part of the report printed by Helena after the search. This report simply consists of a set of statistics collected by Helena. The statistics reported gives various informations on the analyzed net, the size of the reachability graph, or the memory consumption. Let us point out that the type of statistics displayed depend on the option passed to Helena. For instance, no statistic concerning the reachability graph are available in a fast simulation mode.

2.2.2 Generating reports

Helena can produce an XML report file corresponding of the report displayed when it is invoked with option `--report-file`. The purpose of XML reports is to ease the interface between Helena and other tools.

2.3 Additional utilities

Together with Helena are installed several utilities that we briefly describe here.

2.3.1 The helena-report utility

The purpose of helena-report is to print an XML report that has been created by Helena. This utility is useful in the case where you have already invoked Helena on a net and you do not want to launch the search again. Here is an example of use of this utility:

```
helena my-net.lna
helena-report my-net
```

where `my-net` is the name of the net of file `my-net.lna`. The search report will then be printed to the standard output. Alternatively, you can directly pass to helena-report an xml report previously generated. For example the following sequence of commands is equivalent to the previous one:

```
helena --report-file=my_report.xml my-net.lna
helena-report my_report.xml
```

2.3.2 The helena-graph utility

Helena can build the reachability graph of net in order to display some statistics on, e.g., its strongly connected components. This is the purpose of the `--action=BUILD-GRAPH` option. This option is only meaningful if used in conjunction with the helena-graph utility. Let us assume that the file `my-net.lna` contains the description of net `my-net`. A typical use of this combination is

```
helena --action=BUILD-GRAPH my-net.lna
helena-graph my-net my_rg_report.pdf
```

- The first command explores the reachability graph of the net and stores it on disk in the model directory (in `~/.helena`, by default).
- The second command reads this file and produces a report containing various informations on the graph e.g., in-/out-degrees of nodes, shape of the BFS level graph, SCCs of the graph, dead markings, live transitions, ...

The output format of this report can be pdf or xml. In the case of a pdf report, you will need pdflatex as well as the Gnuplot python library on your system.

2.3.3 The helena-generate-interface utility

This tool is used to generate a C header file containing the translation of types, constants, and functions that can then be used in imported modules. Please consult Chapter 4 for further help on this tool.

Chapter 3

Examples

In this section we illustrate the possibilities of our description language. The first system studied is a distributed database system. Then we describe a load balancing system which makes use of more advanced features of our tool. The third example is the well-known puzzle of the towers of Hanoi. This one illustrates a use of high-level data types provided by Helena.

3.1 The distributed database system

We consider in this system a set of N database managers which communicate to maintain consistent replica of a database. It is a well-known and recurrent example of the colored Petri nets literature, initially presented by Genrich and later by Jensen.

When a manager updates its local copy of the database, he sends requests to other managers for updating their local copy (transition *Update*). As soon as a manager receives such a request (transition *Receive*) he starts the update of its copy. Its update finished, each manager acknowledges the initiating manager (transition *Send ack*). This process finishes when the initiating manager collects all the acknowledgments (transition *Receive acks*). Managers can be either *Inactive*, either *Waiting* for acknowledgments, either *Performing* an update. Places *Msgs*, *Received*, *Acks* and *Unused* model communication channels between sites. Thus, $N.(N-1)$ tokens are distributed upon these places at each marking. At last the correctness of the protocol is ensured by place Mutex which guarantees that two managers cannot concurrently update their local copy.

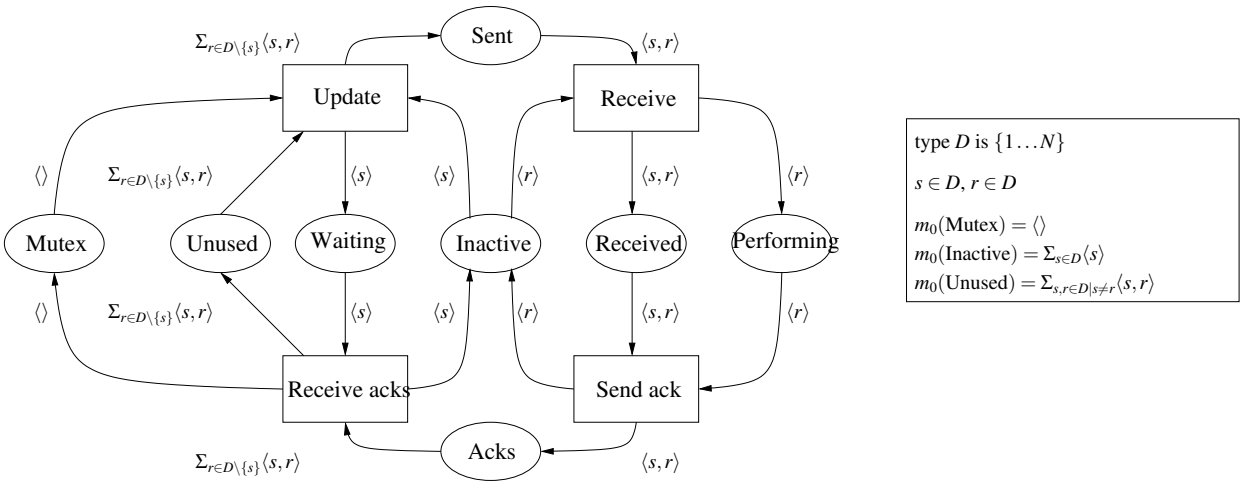


Figure 3.1: The distributed database system

Listing 3.1: Helena file of the distributed database system (file `examples/dbm.lna`)

```
1 /*************************************************************************
2 *
3 *  Example file of the Helena distribution
4 *
```

```
5  *  File   : dbm.lna
6  *  Author: Sami Evangelista
7  *  Date   : 27 oct. 2004
8  *  Source:
9  *     Coloured Petri Nets: A high level language for system design and analysis
10 *     In Application and Theory of Petri Nets, p.342--416, Springer, 1989
11 *     Kurt Jensen
12 *
13 *  If symbol UNUSED is defined, the model includes the place unused.
14 *
15 ******************************************************************************/
16
17 dbm (N := 10) {  /*  N = number of sites  */
18
19    type site_id : mod N;
20
21    /*
22     *  process places modelling the control flow of processes
23     */
24    place inactive {
25       dom : site_id;
26       init : for(s in site_id) <( s )>;
27       capacity : 1;
28       type: process;
29    }
30    place waiting {
31       dom : site_id;
32       capacity : 1;
33       type: process;
34    }
35    place performing {
36       dom : site_id;
37       capacity : 1;
38       type: process;
39    }
40
41    /*
42     *  places modelling communication channels
43     */
44    place sent {
45       dom : site_id * site_id;
46       capacity : 1;
47       type: buffer;
48    }
49    place received {
50       dom : site_id * site_id;
51       capacity : 1;
52       type: buffer;
53    }
54    place acks {
55       dom : site_id * site_id;
56       capacity : 1;
57       type: ack;
58    }
59 #ifdef UNUSED
60    place unused {
61       dom : site_id * site_id;
62       init : for(s in site_id, r in site_id) if(s != r) <( s, r )>;
63       capacity : 1;
64       type: buffer;
65    }
66 #endif
```

```
67    place mutex {
68       dom : epsilon;
69       init : epsilon;
70       capacity : 1;
71       type: shared;
72    }
73
74    transition update_and_send {
75       in {
76          inactive : <( s )>;
77          mutex    : epsilon;
78 #ifdef UNUSED
79          unused   : for(r in site_id) if(s != r) <( s, r )>;
80 #endif
81       }
82       out {
83          waiting : <( s )>;
84          sent    : for(r in site_id) if(s != r) <( s, r )>;
85       }
86    }
87    transition receive_acks {
88       in {
89          waiting : <( s )>;
90          acks    : for(r in site_id) if(s != r) <( s, r )>;
91       }
92       out {
93          inactive : <( s )>;
94          mutex    : epsilon;
95 #ifdef UNUSED
96          unused   : for(r in site_id) if(s != r) <( s, r )>;
97 #endif
98       }
99    }
100   transition receive_message {
101      in {
102         inactive : <( r )>;
103         sent     : <( s, r )>;
104      }
105      out {
106         performing : <( r )>;
107         received   : <( s, r )>;
108      }
109   }
110   transition send_ack {
111      in {
112         performing : <( r )>;
113         received   : <( s, r )>;
114      }
115      out {
116         inactive : <( r )>;
117         acks     : <( s, r )>;
118      }
119   }
120
121   /*
122    *  state propositions
123    */
124   proposition site_waiting: waiting'card > 0;
125 }
```

Listing 3.2: Helena file of the distributed database system properties (file `examples/dbm.prop.lna`)

```
1 /*
2  *  a site waiting for answer will eventually leave this state
3  */
4 ltl property bounded_wait:
5    ([] (site_waiting => <> (not site_waiting)));
```

3.2 The load balancing system

We propose to specify and verify a simple load balancing system with Helena. The full net is illustrated by Figure 3.2. Initial markings and transition guards have been omitted to clarify the figure.

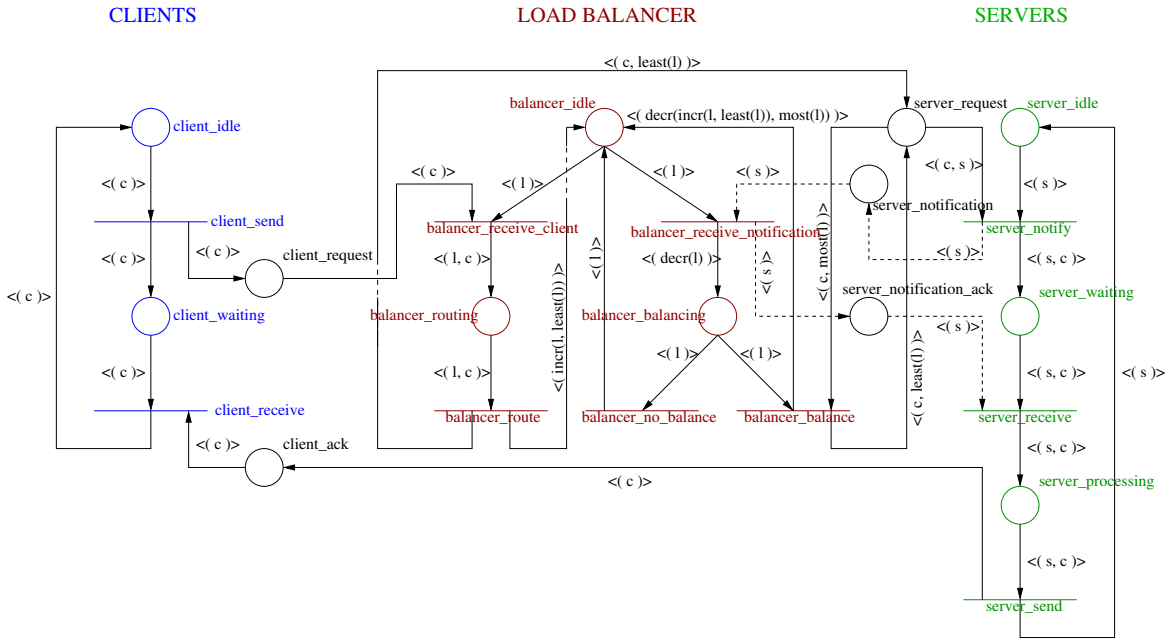


Figure 3.2: The whole load balancing system

In this system, we have two kinds of process: a set of clients and a set of servers. An additional process called the load balancer distribute requests of clients to servers. Its task is also to redistribute pending requests when servers accept requests in order to maintain the loads of servers balanced.

The clients We note C the number of clients considered. Clients are numbered from 1 to C. The behavior of the clients is quite simple. A client may want to send a request to a set of servers. Instead of asking a server directly, he sends the request to the load balancer which will route the request to the adequate server, i.e., the least loaded server. Once the request sent, the client waits for the answer. When this one arrives, the client comes back to the idle state.

The servers The number of servers is noted S. Servers are numbered from 1 to S. Servers receive requests from clients via the load balancer process. When a server accepts a request, he first has to notify this to the load balancer process, in order that this one rebalances the pending requests. Then he has to wait for an acknowledgment from the load balancer to start treating the request. Once the request treated, he directly sends the answer to the concerned client and goes back to the idle state.

The load balancer The load balancer can perform two kinds of task. The first one is to redirect each client request to the least loaded server. Secondly, when a server accepts a request from a client the load balancer has to rebalance the pending requests. If these are already balanced, the load balancer has nothing to perform and can come back to its idle state (transition balancer_no_balance). If the loads are not balanced, the load balancer takes a pending request of the most loaded server and redirects it to the least loaded server (transition balancer_balance). The load balancer has to maintain for each server the number of requests sent to this server.

Listing 3.3: Helena file of the load balancing system (file `examples/load_balancer.lna`)

```
/*************************************************************************
*
*  Example file of the Helena distribution
*
*  File  : load_balancer.lna
*  Author: Sami Evangelista
*  Date  : 27 oct. 2004
*
*  This file contains the description of a load balancing system.
*
*************************************************************************/


load_balancer (C := 7,    /*  number of clients  */
               S := 2) {  /*  number of servers  */

   /*  clients  */
   type client_id  : range 1 .. C;
   type clients_no : range 0 .. client_id'last;

   /*  servers  */
   type server_id : range 1 .. S;

   /*  load  */
   type servers_load : vector [server_id] of clients_no;
   constant servers_load empty_load := [0];


   /*  return the least loaded server  */
   function least (servers_load load) -> server_id {
      server_id result := server_id'first;
      for(i in server_id)
         if(load[i] < load[result])
            result := i;
      return result;
   }

   /*  return the most loaded server  */
   function most (servers_load load) -> server_id {
      server_id result := server_id'first;
      for(i in server_id)
         if(load[i] > load[result])
            result := i;
      return result;
   }

   /*  check if load is balanced  */
   function is_balanced (servers_load load) -> bool {
      clients_no max_no := 0;
      clients_no min_no := clients_no'last;
      for(i in server_id)
      {
         if(load[i] > max_no) max_no := load[i];
         if(load[i] < min_no) min_no := load[i];
      }
      return (max_no - min_no) <= 1;
   }

   /*  increment the load of server i  */
   function incr (servers_load l, server_id i) -> servers_load
```

```
61        return l :: ([i] := l[i] + 1);
62
63    /*  decrement the load of server i  */
64    function decr (servers_load l, server_id i) -> servers_load
65        return l :: ([i] := l[i] - 1);
66
67    /*  return the difference between the two loads  */
68    function diff (clients_no c1, clients_no c2) -> clients_no
69        return (c1 > c2) ? (c1 - c2) : (c2 - c1);
70
71
72    /*
73     *  clients
74     */
75    place client_idle {
76        dom : client_id;
77        init : for(c in client_id) <( c )>;
78        capacity : 1;
79    }
80    place client_waiting {
81        dom : client_id;
82        capacity : 1;
83    }
84    place client_request {
85        dom : client_id;
86        capacity : 1;
87    }
88    place client_ack {
89        dom : client_id;
90        capacity : 1;
91    }
92    transition client_send {
93        in  { client_idle    : <( c )>; }
94        out { client_waiting : <( c )>;
95              client_request : <( c )>; }
96        description: "client␣%d:␣send␣request", c;
97    }
98    transition client_receive {
99        in  { client_waiting : <( c )>;
100             client_ack     : <( c )>; }
101       out { client_idle    : <( c )>; }
102       description: "client␣%d:␣receives␣response", c;
103   }
104
105
106   /*
107    *  servers
108    */
109   place server_idle {
110       dom : server_id;
111       init : for(s in server_id) <( s )>;
112       capacity : 1;
113   }
114   place server_waiting {
115       dom : server_id * client_id;
116       capacity : 1;
117   }
118   place server_processing {
119       dom : server_id * client_id;
120       capacity : 1;
121   }
122   place server_notification {
```

```
123       dom : server_id;
124       capacity : 1;
125    }
126    place server_notification_ack {
127       dom : server_id;
128       capacity : 1;
129    }
130    place server_request {
131       dom : client_id * server_id;
132       capacity : 1;
133    }
134    transition server_notify {
135       in  { server_idle         : <( s )>;
136             server_request      : <( c, s )>; }
137       out { server_waiting      : <( s, c )>;
138             server_notification : <( s )>; }
139       description: "server␣%d:␣lb␣process␣notification", s;
140    }
141    transition server_receive {
142       in  {server_waiting          : <( s, c )>;
143            server_notification_ack : <( s )>; }
144       out {server_processing       : <( s, c )>; }
145       description: "server␣%d:␣reception␣of␣request␣from␣client␣%d", s, c;
146    }
147    transition server_send {
148       in  {server_processing : <( s, c )>; }
149       out {server_idle       : <( s )>;
150            client_ack        : <( c )>;}
151       description: "server␣%d:␣send␣response␣to␣client␣%d", s, c;
152    }
153
154
155    /*
156     *  load balancer process
157     */
158    place balancer_idle {
159       dom : servers_load;
160       init : <( empty_load )>;
161       capacity : 1;
162    }
163    place balancer_routing {
164       dom : servers_load * client_id;
165       capacity : 1;
166    }
167    place balancer_balancing {
168       dom : servers_load;
169       capacity : 1;
170    }
171    transition balancer_receive_client {
172       in  {balancer_idle    : <( l )>;
173            client_request   : <( c )>; }
174       out {balancer_routing : <( l, c )>; }
175       description: "lb:␣receive␣request␣of␣client␣%d", c;
176    }
177    transition balancer_route {
178       in  { balancer_routing : <( l, c )>; }
179       out { balancer_idle    : <( incr(l, ll) )>;
180             server_request   : <( c, ll )>; }
181       let { server_id ll := least(l); }
182       description: "lb:␣route␣request␣of␣client␣%d␣to␣server␣%d", c, ll;
183    }
184    transition balancer_receive_notification {
```

```
185       in  { balancer_idle           : <( 1 )>;
186             server_notification     : <( s )>; }
187       out { server_notification_ack : <( s )>;
188             balancer_balancing      : <( decr(l, s) )>; }
189       description: "lb: receive notification of server %d", s;
190    }
191    transition balancer_balance {
192       in  { balancer_balancing : <( 1 )>;
193             server_request     : <( c, most(l) )>; }
194       out { balancer_idle      : <( decr(incr(l, ll), ml) )>;
195             server_request     : <( c, ll )>; }
196       let { server_id ll := least(l);
197             server_id ml := most(l); }
198       guard: not is_balanced(l);
199       description: "lb: redirect request of client %d from server %d \
200 to server %d", c, ml, ll;
201    }
202    transition balancer_no_balance {
203       in  { balancer_balancing : <( 1 )>; }
204       out { balancer_idle      : <( 1 )>; }
205       guard: is_balanced(l);
206       description: "lb: no rebalance";
207    }
208
209
210    /*
211     *  state propositions
212     *
213     *  load_not_balanced: for each couple of servers (s1,s2) with s1 != s2,
214     *  the difference between the number of requests pending or accepted by
215     *  s1 and the number of requests pending or accepted by s2 is at most 1.
216     */
217    proposition load_not_balanced:
218       not forall (s1 in server_id, s2 in server_id | s1 != s2 :
219             diff (card (sr in server_request | sr->2 = s1) +
220                   card (sn in server_notification | sn->1 = s1),
221                   card (sr in server_request | sr->2 = s2) +
222                   card (sn in server_notification | sn->1 = s2)) <= 1);
223    proposition balancing:
224       balancer_balancing'card = 1;
225 }
```

Listing 3.4: Helena file of the load balancing system properties (file `examples/load_balancer.prop.lna`)

```
1  /*
2   *  reject any deadlock state
3   */
4  state property not_dead:
5     reject deadlock;
6
7  /*
8   *  the loads are balanced or are being rebalanced
9   */
10 state property balance_ok:
11    reject load_not_balanced;
12    accept balancing;
```

3.3 The towers of Hanoi

The towers of Hanoi is a well-known mathematical game[1]. It consists of three towers, and a number of disks of different sizes which can slide onto any tower. The puzzle starts with the disks neatly stacked in order of size on one tower, smallest at the top, thus making a conical shape.

The objective of the game is to move the entire stack to another tower, obeying the following rules:

- Only one disk may be moved at a time.
- Each move consists of taking the upper disk from one of the towers and sliding it onto another tower, on top of the other disks that may already be present on that tower.
- No disk may be placed on top of a smaller disk.

This example illustrates the use of lists in Helena.

Listing 3.5: Helena file of the towers of Hanoi (file `examples/hanoi.lna`)

```
1  /*************************************************************************
2  *
3  *  Example file of Helena distribution
4  *
5  *  File   : hanoi.lna
6  *  Author: Sami Evangelista
7  *  Date   : 15 feb. 2007
8  *
9  *  This file contains the description of the towers of Hanoi game.
10 *
11 *************************************************************************/
12
13 hanoi (N := 3,      /*  N = the number of disks  */
14        M := 3)  {   /*  M = number of towers  */
15
16    //  identifier of a disk
17    type disk: range 1..N;
18
19    //  identifier of a tower
20    type tower: range 1 .. M;
21
22    //  a list of disks
23    type disk_list: list[nat] of disk with capacity N;
24
25    //  construct the list of disks initially present on the first tower
26    function construct_tower1() -> disk_list {
27       disk_list result := empty;
28       for(i in disk)
29          result := i & result;
30       return result;
31    }
32
33    //  this unique place models the state of the towers.
34    //  in the initial marking the first tower contains the list |N, ..., 1|
35    //  and all others are empty
36    place towers {
37       dom : tower * disk_list;
38       init: <( tower'first, construct_tower1() )>
39           + for(t in tower range tower'first + 1 .. tower'last)
40                <( t, empty )>;
41    }
42
43    //  transition move_disk models the move of the disk on top
```

[1]The description is taken from http://en.wikipedia.org/wiki/Tower_of_Hanoi.

```
44      //  of tower src to the tower dest.  the src stack must not be
45      //  (not src_disks'empty).  if the dest tower is not empty, the
46      //  disk on top of src (src_disks'last) must be smaller than the disk
47      //  on top of the dest tower (dest_disks'last).
48      //  the move consists of deleting the last element from the src stack
49      //  (src_disks'prefix is the the list src_disks from which we remove
50      //  the last element) and pushing it onto the dest stack.
51      transition move_disk {
52         in  {
53            towers: <( src,  src_disks  )>
54                  + <( dest, dest_disks )>;
55         }
56         out {
57            towers: //  remove the last disk of tower t
58                    <( src,  src_disks'prefix )>
59
60                    //  add the removed disk on top of tower u
61                  + <( dest, dest_disks & src_disks'last )>;
62         }
63         guard: not src_disks'empty
64                and (dest_disks'empty or src_disks'last < dest_disks'last);
65         description: "move␣disk␣%d␣from␣tower␣%d␣to␣tower␣%d",
66                      src_disks'last, src, dest;
67      }
68
69      //  in the end state the last tower is full, i.e., it contains the list
70      //  |N, ..., 1|
71      proposition all_moved:
72         exists(t in towers | t->1 = tower'last and t->2'full);
73   }
```

Listing 3.6: Helena file of the towers of Hanoi properties (file `examples/hanoi.prop.lna`)

```
1   //  we reach a state in which all disks have been moved
2   //  to the last tower
3   state property end_state:
4      reject all_moved;
```

Chapter 4

Interfacing Helena with C code

Although the language provided by Helena for arc expressions is quite rich, it may not be sufficient and the user may, for instance, prefer to use its own C functions rather than writing these in the Helena specification. This is provided by the language through the **import** construct. As the code generated by Helena is written in C all imported components have to be written in this language. This chapter first starts with a tutorial illustrating the use of this feature.

4.1 Tutorial: Importing C Functions

Our goal is to simulate the quick-sort algorithm with Helena. The net we want to analyze consists of a single transition swap that takes a list of integers from a place myList, swaps two of its elements and put its back in place myList. Each occurence of this transition simulates a single step of the quick-sort algorithm. The list we want to sort, toSort, is a constant initialized via function initList . Below is the definition of this net.

Listing 4.1: Helena file of the sort net (file `example/sort.lna`)

```
1  /*************************************************************************
2  *
3  *  Example file of Helena distribution
4  *
5  *  File   : sort.lna
6  *  Author: Sami Evangelista
7  *  Date   : 3 mar. 2010
8  *
9  *  This simple example illustrates the use of imported functions.
10 *
11 *  Please read the user's guide in directory doc if you want more
12 *  informations on this model.
13 *
14 *  To analyse this net you must first compile the C imported functions and then
15 *  invoke helena as follows:
16 *  > helena-generate-interface sort.lna sort_interface.h
17 *  > gcc -c initList.c quickSort.c isSorted.c
18 *  > helena -L=initList.o -L=quickSort.o -L=isSorted.o sort.lna
19 *
20 *************************************************************************/
21
22 sort {
23
24 type intList: list[nat] of int with capacity 1000;
25
26 function initList ()                    -> intList;
27 function quickSort (intList l, int steps) -> intList;
28 function isSorted (intList l)           -> bool;
29
```

```
30 constant intList toSort := initList();
31
32 place myList { dom : intList * int; init: <( toSort, 1 )>; }
33
34 transition swap {
35    in  { myList: <( l, steps )>; }
36    out { myList: <( quickSort(toSort, steps), steps + 1 )>; }
37    guard: not isSorted(l);
38 }
39
40 import function initList ()                      -> intList;
41 import function quickSort (intList l, int steps) -> intList;
42 import function isSorted (intList l)             -> bool;
43
44 }
```

The transition swap is only firable if the list taken in place myList is not already sorted. The variable steps of transition swap is used to count the number of swaps we have to perform with quick-sort algorithm. Note that the function quickSort is always called with list toSort. Hence place myList will successively contain the following token:

```
toSort
quickSort(toSort, 1)
quickSort(toSort, 2)
quickSort(toSort, 3)
...
```

until the function returns a sorted list.

Now let us suppose that we do not want the functions used in this net to be written in Helena but directly in C. The easiest solution is to use the **import** feature of Helena. Note that after these imports, the bodies of these functions do not have to be declared. This would actually be an error.

Lastly, in order to follow the sequence of swaps performed by quick-sort we write the following line in the property file.

```
state property not_dead:
   reject deadlock;
```

In order to write these C functions we need to know how the types of their parameters have been mapped to C. This is the purpose of the `helena-generate-interface` tool that is invoked with only two parameters as below:

```
helena-generate-interface sort.lna sort_interface.h
```

File `sort_interface.h` is the resulting C header file that contains all declarations that could be required by the user to write his (her) imported functions.

We only provide here the declarations that are required for the understanding of this tutorial. Looking at the net specification, we need to access the declarations of types int, intList and bool. These three types are mapped to the three following types.

```
typedef int TYPE_int;

typedef char TYPE_bool;
#define TYPE__ENUM_CONST_bool__false 0
#define TYPE__ENUM_CONST_bool__true 1

typedef struct {
   TYPE_int items[1000];
   unsigned int length;
} TYPE_intList;
```

We first notice that each Helena type myType is mapped to a C type TYPE_myType. For the boolean type, TYPE_bool, we notice that its constants **false** and **true** have been mapped to 0 and 1. The list type intList is mapped to a structured type TYPE_intList with two components:

- items is the content of the list stored in an array. The size of this array is equal to the capacity of the list type intList .

- length is the length of the list, i.e., the number of integers in array items that are actually part of the list.

We are now able to write the three functions that are imported in our net specification. Each function has been put in a separate file. The content of these three files is depicted on Figure 4.1. For each imported function myFunc in the net declaration there must be a C function IMPORTED_FUNCTION_myFunc. The return type and the parameter types of the C function and the function in the net declaration must match. Otherwise a compilation error will occur when invoking Helena.

Now that we have written our imported functions we can analyze this net with Helena. First, we compile the C code of imported functions as follows:

```
gcc -c initList.c
gcc -c quickSort.c
gcc -c isSorted.c
```

We can now invoke Helena with option `-L` in order to specify which object files must passed to the linker.

```
helena -L=initList.o -L=quickSort.o -L=isSorted.o --action=check-not_dead sort.lna
```

Helena automatically compiles C files generated for the net file `sort.lna` and link them with files `initList.o`, `quickSort.o` and `isSorted.o`. After the search is completed we can see a simulation of the quick-sort algorithm for the simple list returned by function initList :

```
{
  myList = <( |4, 1, 3, 0, 2|, 1 )>
}
(swap, [1 = |4, 1, 3, 0, 2|, steps = 1]) ->
{
  myList = <( |2, 1, 3, 0, 4|, 2 )>
}
(swap, [1 = |2, 1, 3, 0, 4|, steps = 2]) ->
{
  myList = <( |2, 1, 0, 3, 4|, 3 )>
}
(swap, [1 = |2, 1, 0, 3, 4|, steps = 3]) ->
{
  myList = <( |0, 1, 2, 3, 4|, 4 )>
}
```

4.2 The interface file

As shown in our tutorial, the `helena-generate-interface` tool must be invoked in order to generate an header file containing the C code that could be required to implement imported modules. The purpose of this section is to describe exactly the content of this header file.

4.2.1 Generated types

The Table 4.1 contains for each kind of type or sub-type in the Helena net the corresponding C type that is generated. The translation is pretty straightforward. We can however make the following comments:

- Each Helena type or sub-type t is mapped to a C type TYPE_t.
- A numeric type is mapped to type **short** or **int** depending on its range of values. The same applies for numeric types.
- Each value val of an enumerate type t is mapped to a macro TYPE__ENUM_CONST_t__val that is expanded to the position (minus 1) of the value in the list that defines the enumerate type. Note that some names are actually quite long. The reason is that we thus avoid name conflicts in the generated code.
- A vector type vt is mapped to a structured type containing a single array element called vector. The dimension(s) of this array match(es) with the cardinal(s) of the type(s) used to the define the Helena vector type. In our example, the integer at index [blue, **false**] of a C variable var of type TYPE_colors can be accessed as follows : var.vector[TYPE__ENUM_CONST_color__blue][TYPE__ENUM_CONST_bool__false].

```c
/*  File: initList.c  */
#include "sort_interface.h"

TYPE_intList IMPORTED_FUNCTION_initList() {
  TYPE_intList result;
  result.length = 5;
  result.items[0] = 4;
  result.items[1] = 1;
  result.items[2] = 3;
  result.items[3] = 0;
  result.items[4] = 2;
  return result;
}
```

```c
/*  File: quickSort.c  */
#include "sort_interface.h"

void swap(int *a, int *b) {
  int t = *a; *a = *b; *b = t;
}
void quickSort(int arr[], int beg, int end, int * nb) {
  if (end > beg + 1) {
    int piv = arr[beg], l = beg + 1, r = end;
    while(l < r) {
      if(arr[l] <= piv) l++;
      else {
        if(*nb == 0) return;
        swap(&arr[l], &arr[--r]);
        (*nb) --;
      }
    }
    if(*nb == 0) return;
    swap(&arr[--l], &arr[beg]);
    (*nb) --;
    quickSort(arr, beg, l, nb);
    quickSort(arr, r, end, nb);
  }
}
TYPE_intList IMPORTED_FUNCTION_quickSort(TYPE_intList l, TYPE_int nb) {
  TYPE_intList result = l;
  quickSort(result.items, 0, result.length, &nb);
  return result;
}
```

```c
/*  File: isSorted.c  */
#include "sort_interface.h"

TYPE_bool IMPORTED_FUNCTION_isSorted(TYPE_intList l) {
  int i = 0;
  for(i=0; i<l.length-1; i++)
    if(l.items[i] > l.items[i+1])
      return TYPE__ENUM_CONST_bool__false;
  return TYPE__ENUM_CONST_bool__true;
}
```

Figure 4.1: Imported functions of the tutorial

Table 4.1: Mapping Helena types to C

Helena type	C type
Numeric types	
`type small: range 0 .. 255;`	`typedef short TYPE_small;`
`type big : range 0 .. 65535;`	`typedef int TYPE_big;`
Enumerate types	

`type color: enum (`
`  red,`
`  green,`
`  blue,`
`  yellow,`
`  cyan);`
	`typedef char TYPE_color;`
	`#define TYPE__ENUM_CONST_color__red 0`
	`#define TYPE__ENUM_CONST_color__green 1`
	`#define TYPE__ENUM_CONST_color__blue 2`
	`#define TYPE__ENUM_CONST_color__yellow 3`
	`#define TYPE__ENUM_CONST_color__cyan 4`

Vector types	

`type colors: vector[color, bool]`
`  of int;`
	`typedef struct {`
	`  TYPE_int vector[5][2];`
	`} TYPE_colors;`

Structured types	

`type rgbColor: struct {`
`  small r;`
`  small g;`
`  small b;`
`};`
	`typedef struct {`
	`  TYPE_small r;`
	`  TYPE_small g;`
	`  TYPE_small b;`
	`} TYPE_rgbColor;`

Container types	

`type colorList: list[small]`
`  of color with capacity 5;`
	`typedef struct {`
	`  TYPE_color items[5];`
	`  unsigned int length;`
	`} TYPE_colorList;`

`type smallSet: set`
`  of small with capacity 5;`
	`typedef struct {`
	`  TYPE_small items[5];`
	`  unsigned int length;`
	`} TYPE_smallSet;`

Sub-types	
`subtype tiny: small` `  range 0 .. 15;`	`typedef TYPE_small TYPE_tiny;`

`subtype rgColor: color`
`  range red .. green;`
	`typedef TYPE_color TYPE_rgColor;`
	`#define TYPE__ENUM_CONST_rgColor__red 0`
	`#define TYPE__ENUM_CONST_rgColor__green 1`
	`#define TYPE__ENUM_CONST_rgColor__blue 2`
	`#define TYPE__ENUM_CONST_rgColor__yellow 3`
	`#define TYPE__ENUM_CONST_rgColor__cyan 4`

Table 4.2: Mapping Helena constants and functions to C

Helena construct	C construct
Constants	
`constant rgbColor BLUE := {0, 0, 255};`	`TYPE_rgbColor CONSTANT_BLUE;`
Functions	
`function isBlack (rgbColor c) -> bool { return c.r + c.g + c.b = 0; }`	`TYPE_bool FUNCTION_isBlack (TYPE_rgbColor V2);`

- A structured type is mapped to a C **struct** type that has exactly the same structure.
- A container type ct is mapped to a C **struct** type TYPE_ct containing two elements: the items of the list (or set) stored in an array items; and, stored in an integer component length, the number of items in this array that are actually part of the container. For instance, the Helena expression |red, green, cyan| of type colorList is equivalent to a C expression ex of type TYPE_ct defined by:
  - ex.items[0] = TYPE__ENUM_CONST_color__red
  - ex.items[1] = TYPE__ENUM_CONST_color__green
  - ex.items[2] = TYPE__ENUM_CONST_color__cyan
  - ex.length = 3
  - the value of ex.items[3] and ex.items[4] are irrelevant.
- A sub-type is simply translated to its parent type.

4.2.2 Generated constants and functions

It may be useful for the user to access in imported modules the values of some constant(s) or the function(s) declared in the net specification. Hence, each constant or function is also accessible in the header file generated by the `helena-generate-interface` tool.

An example of translation is provided by Table 4.2. The mapping is straightforward. We simply notice that an Helena constant const is mapped to a C variable CONSTANT_const and that an Helena function func is mapped to a C function FUNCTION_func. In addition, the parameter and return types of the Helena function and the C function must match.

4.3 Requirements on imported modules

Imported functions must fulfill some requirements so that it does not impact negatively on the behavior of Helena. These are listed below.

- An imported function may not have any side effect. In particular, it is absolutely necessary that the function frees all memory it allocates. Otherwise, since the function will be called multiple times during the search, memory could be quickly saturated.
- An imported function must terminate. This guarantees that the search also does.
- An imported function must be deterministic. If the function is not, Helena is not guaranted to report the same result across different executions. The only exception is for functions that are used only once for the initialization of some net constant(s). The value of a constant may for instance be read from a file or from user inputs. Note that all files accessed in imported functions must necessarily be accessed via an absolute path.

In addition it must also hold that, for each imported function func in the net description, there is in imported module(s) a function IMPORTED_FUNCTION_func such that its parameter and return types match with the declaration of function func in the net description.

Chapter 5

Help

5.1 Evaluation of Transitions

Helena puts some restrictions on the variables of transitions in order to be able to efficiently compute enabled transition bindings at a given marking. Besides the fact that all variables of a transition have to be declared by appearing in a tuple labelling its input or inhibitor arcs or otherwise in the pick section, additional constraints are put on these variables. We summarize below the evaluation process in two situations: when the transition does not have inhibitor arcs and when it does.

5.1.1 Evaluation in the absence of inhibitor arcs

During the computation of the enabled bindings of a transition that does not have any inhibitor arc, Helena has to evaluate the labels of the input arcs of the transition, its guard, and finally has to pick all the possible acceptable values for its free variables defined in the pick section. Hence, three types of items have to be evaluated to bind all the variables of the transition and find enabled bindings: tuples of input arcs, the guard, and free variables.

Each of these items define some variables, i.e., bind them by giving them a value, and use some variables that have to be defined so that the item can be evaluated. The table below summarizes which variables are used/defined by each kind of item.

Item	Variables used	Variables defined
Tuple	all variables appearing in the tuple (and not defined in it)	all variables appearing in the tuple at the top level (not in a sub-expression)
Guard	all variables appearing in the guard	none
Free variable	all variables appearing in the definition of the variable	the variable

In the absence of inhibitor arcs a transition must fulfill the following requirement to be firable: there must be an evaluation order $item_1,\ldots,item_n$ such that, for any $i \in \{1..n\}$ and any variable v used by $item_i$, there is $j \in \{1..i-1\}$ such that $item_j$ defines variable v. Otherwise, the transition will not be evaluable. This is for example the case with transitions t and u defined below:

```
transition t {
   in  { p: <(x, f(y))> + <(y, x * 2)>; }
   out { q: <(x)>; }
}
transition u {
   in  { p: <(x, f(y))>; }
   out { q: <(x)>; }
   pick { y in g(x); }  // g returns a set of integers
}
```

For transition t the tuple <(x, f(y))> defines variable x but to evaluate it we require that y must be defined. The tuple <(y, x * 2)> defines this variable but needs itself variable x to be defined. Hence, there is here a cyclic dependency. We face the same problem for transition u.

The invokation of Helena on this example will raise the following errors:

```
test.lna:4: Transition t cannot be evaluated
test.lna:8: Transition u cannot be evaluated
```

5.1.2 Evaluation in the presence of inhibitor arcs

If the transition has inhibitor arcs then the process described in the previous section first takes place. If we have found some binding for the variables defined in the input tuples or in the pick section of the transition then a second evaluation process starts for the inhibitor arcs of the transition. To be firable, there must not be any binding for the variables defined by inhibitor arcs such that the corresponding tokens are present in the place linked by the inhibitor arc.

Let us for instance consider the following transitions.

```
transition t {
   in      { p: <(x)>; }
   out     { q: <(x)>; }
   inhibit { r: <(x)>; }
}
transition u {
   in      { p: <(x)>; }
   out     { q: <(x)>; }
   inhibit { s: <(x, y)>; }
}
```

Transition t is firable for some binding x if and only if the token <(x)> is present in p but not in r. Transition u is firable for some binding x if and only if the token <(x)> is present in p and there exists no y such that the token <(x, y)> is present in s.

Note that variables defined in inhibitor arcs are not variables of the transition. For example, variable y of transition u serves only during the evaluation of the transition.

5.2 Tips and Tricks

This section gives some hints to use Helena in an efficient way.

Saving memory Helena provides some predefined data types. However, we encourage users to define application specific data types in order to limit the possible values of variables, and to save memory when markings are stored in the reachability set. Let us consider for instance a place with domain `int`. Each token of this place will be encoded in a marking with 32 bits. However, if we know that the tokens of this place can only belong to the range [0..50], it is preferable to define a new range type ranging from 0 to 50. Thus tokens will fit in 6 bits instead of 32. If you are not sure of this range, use option `--run-time-checks` to detect expressions going out of range. For the same reason, try to limit capacities of places. Option `--run-time-checks` will also detect violated capacities.

Storage methods We advise to proceed as follows when verifying a property :

- Use a partial search with hash compaction method (option `--hash-compaction`). This method can explore a large portion of the state space and report errors much faster.
- If no error is found during the first stage, use the default storage method without compression technique.
- If the second search ran out of memory, use compression techniques (option `--delta`) and rerun the search.

Run time checks Enabling run time checks usually slows the analysis since additional code is put in the generated code to check that no error occurs. When the net contains many arithmetic operations the run time can grow significantly. Thus, we advise to disable run time checks if no error is suspected in the model.

Expressing properties Analysis techniques and reductions, i.e., partial order reduction, performed by Helena depends on the property to be checked. The reduction observed decreases with the complexity of the property. Thus, try to verify properties separately when possible. For instance, instead of verifying p **and** q, verify first p and then q.

Depth- vs. Breadth-first search Depth (option `--algo=DFS`) and breadth-first search (option `--algo=BFS`) both perform a full state space search. Depth-first search usually leads to better reduction than breadth-first search, especially when partial order methods are used (option `--partial-order`). Breadth-first search, however, reports counter examples of minimal length. Thus, keep in mind these two factors when choosing the type of search to apply. Though the techniques implemented in Helena are fully automatic and do not require any assistance from the user, some informations can be put in the specification in order to guide Helena into the search and optimize some of these techniques. We detail now these features.

5.3 Guiding Helena in the search

Some informations provided by the user are not mandatory but given to Helena in order to make its search more efficient. However, Helena does not have any mean to check the validity of the informations provided. Thus if some of these revealed to be erroneous Helena could produce wrong results, e.g., report that a property is verified whereas it is not. We therefore encourage users to be very careful when supplying these informations and to ignore them if there is any doubt regarding their validity.

5.3.1 Typing places

A type can be associated to each place of the net. This type specifies the nature of the information modeled by the place. The types provided allow to model concurrent systems and protocols synchronizations through shared variables and communication buffers. Six kinds of places are allowed: process places, local places, shared places, protected places, buffer places and ack places.

Process places (Figure 5.1, top left) Process places model the control flow of processes, i.e., its position in the code it executes. A process is therefore in exactly one of the process places of the net. The simple net depicted models cyclic process. Each process p can go from state $Idle$ to state $Working$. Thus, for each process p there is a token $\langle p \rangle$ in place $Idle$ or in place $Working$.

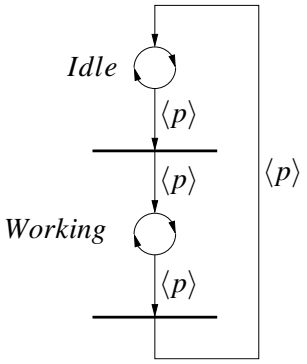
Local places (Figure 5.1, top right) Local places model resources local to a process. Intuitively, this means that two different processes cannot withdraw the same token from a local place. The net depicted models the incrementation by 1 of a local variable I. This variable is modeled by the place I. The first component of the domain of this place is the identifier of the process which owns the local variable and the second one gives the value of this variable. Since a process p can only access its own variable and withdraw a token of type $\langle p, i \rangle$ from place I we can type this place as local. Indeed, two different process cannot currently consume the same token in place I.

Shared and protected places (Figure 5.1, bottom left) Shared places model resources shared by processes. Shared variables or locks are examples of informations which can be modeled by a shared place. Protected places are special shared places. They are used to model resources which can be accessed by several processes but which cannot be accessed concurrently thanks to a mechanism, e.g., a mutex, which guarantees that two processes cannot simultaneously access the resource. The net depicted models the incrementation of a shared variable I. To update the value of I, a process must first grab a lock modeled by place $Lock$. This lock ensures that two processes cannot concurrently update I. We can thus declare I as protected. The place $Lock$ is naturally a shared place since processes compete for the acquisition of this lock.

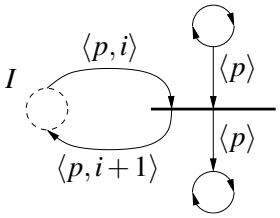
Buffer and ack places (Figure 5.1, bottom right) Processes can also synchronize themselves by sending messages on communication channels. These channels are modeled by buffer places. These places can be thought of as shared places but there is a major difference between the two: a token residing in a buffer place can only be removed by a single process. Thus, channels represented by buffer places are multiple senders - single receiver channels.

Ack places are special kinds of buffer places used to represent acknowledgments of synchronous exchanges. For instance if process p_1 sends a message to process p_2, then waits for the acknowledgment of p_2 before continuing its execution, the place which corresponds to the acknowledgment can be typed as ack.

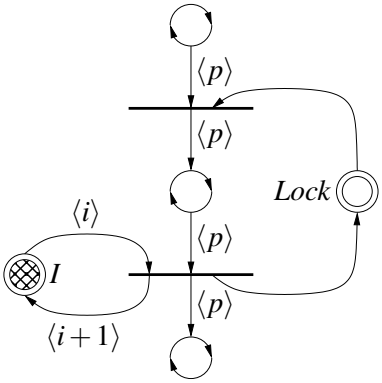
The depicted net models the behavior of a set of clients which interact through messages exchanges. A client c sends a request to a server s by putting a token $\langle c, s \rangle$ in place $Requests$. Since the only server which can receive this request and withdraw the token $\langle c, s \rangle$ from place $Requests$ is s we can type this place as a buffer place. Once received by the server the request is treated and an acknowledgment is sent to the client which can continue its execution. Since this exchange is a synchronous one, the place $Acks$ can be typed as an ack place.



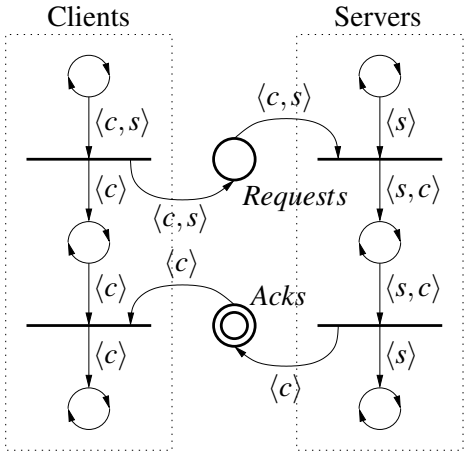
A set of cyclic processes



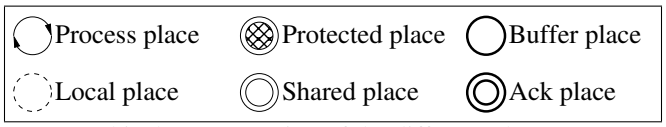
Incrementation of the local variable *I*



An incrementation of the shared variable *I*.
A lock guarantees exclusive access to *I*.



A synchronous exchange between clients and servers



Graphical representation of the different place types

Figure 5.1: Four example nets illustrating the possibility of place typing.

5.3.2 Safe transitions

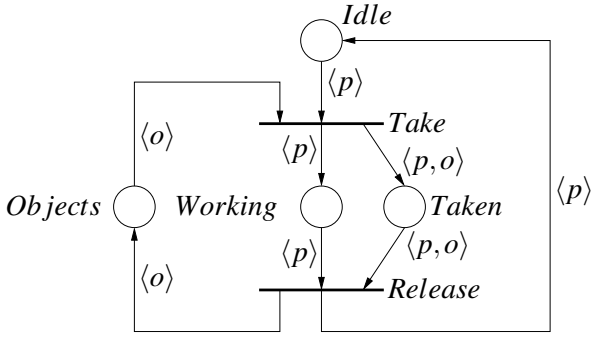


Figure 5.2: Transition *Release* is safe. *Take* is not.

Transitions of the net can be declared as safe. A transition binding is safe if it cannot be disabled by the firing of another binding. In other words, the tokens consumed by the binding cannot be stolen by another binding. If a transition of the net is declared as safe, then Helena considers that all the possible bindings of the transition are safe.

Figure 5.2 depicts an example of safe transition. Each process can be in place *Idle* in place *Working*. To go to work, a process must first acquire an object o. The objects are initially in the place *Objects*. When a process returns to place *Idle*, it puts back the object in place *Objects*. A token $\langle p, o \rangle$ in place *Taken* means that process p has taken object o.

The transition *Take* is clearly not safe. Indeed, each enabled binding $(Take, \langle p, o \rangle)$ needs a token o in place *Objects*, and this token can be removed by any other process q which wishes to acquire the same object. Thus $(Take, \langle q, o \rangle)$ disables $(Take, \langle p, o \rangle)$. Let us now have a look at transition *Release*. Once the binding $(Take, \langle p, o \rangle)$ is fired, there is a token $\langle p \rangle$ in place *Working* and a token $\langle p, o \rangle$ in place *Taken*. Since p is the only process which can remove token $\langle p, o \rangle$ from place *Taken*, the transition binding $(Release, \langle p, o \rangle)$ is safe. This obviously holds for any process p and object o. Consequently we can declare transition *Release* as safe.

Appendix A

Syntax summary

A.1 Net specification language

Nets

⟨*net*⟩ ::= ⟨*net name*⟩
['**(**' ⟨*net parameter list*⟩ '**)**']
'**{**' (⟨*definition*⟩)* '**}**'
⟨*net name*⟩ ::= ⟨*name*⟩
⟨*definition*⟩ ::= ⟨*type*⟩
| ⟨*constant*⟩
| ⟨*function*⟩
| ⟨*place*⟩
| ⟨*transition*⟩
| ⟨*state proposition*⟩

Net parameters

⟨*net parameter list*⟩ ::= ⟨*net parameter*⟩
| ⟨*net parameter*⟩ '**,**' ⟨*net parameter list*⟩
⟨*net parameter*⟩ ::= ⟨*net parameter name*⟩ '**:=**' ⟨*number*⟩
⟨*net parameter name*⟩ ::= ⟨*name*⟩

Types and subtypes

⟨*type*⟩ ::= ⟨*type name*⟩ '**:**' ⟨*type definition*⟩ '**;**'
| ⟨*subtype*⟩
⟨*type name*⟩ ::= ⟨*name*⟩
⟨*type definition*⟩ ::= ⟨*range type*⟩
| ⟨*modulo type*⟩
| ⟨*enumeration type*⟩
| ⟨*vector type*⟩
| ⟨*struct type*⟩
| ⟨*list type*⟩
| ⟨*set type*⟩

Range type

⟨*range type*⟩ ::= ⟨*range*⟩
⟨*range*⟩ ::= '**range**' ⟨*expression*⟩ '**..**' ⟨*expression*⟩

Modulo type

⟨*modular type*⟩ ::= '**mod**' ⟨*expression*⟩

Enumeration type

⟨*enumeration type*⟩ ::= '**enum**' '**(**' ⟨*enumeration constant*⟩ ('**,**' ⟨*enumeration constant*⟩)* '**)**'
⟨*enumeration constant*⟩ ::= ⟨*name*⟩

Vector type

⟨*vector type*⟩ ::= '**vector**' '**[**' ⟨*index type list*⟩ '**]**' '**of**' ⟨*type name*⟩
⟨*index type list*⟩ ::= ⟨*type name*⟩ ('**,**' ⟨*type name*⟩)*

Structured type

⟨*struct type*⟩ ::= '**struct**' '**{**' (⟨*component*⟩)+ '**}**'
⟨*component*⟩ ::= ⟨*type name*⟩ ⟨*component name*⟩ '**;**'
⟨*component name*⟩ ::= ⟨*name*⟩

List type

⟨*list type*⟩ ::= '**list**' '**[**' ⟨*type name*⟩ '**]**' '**of**' ⟨*type name*⟩ '**with**' '**capacity**' ⟨*expression*⟩

Set type

⟨*set type*⟩ ::= '**set**' '**of**' ⟨*type name*⟩ '**with**' '**capacity**' ⟨*expression*⟩

Subtype

⟨*subtype*⟩ ::= ⟨*subtype name*⟩ '**:**' ⟨*parent name*⟩ [⟨*constraint*⟩]
⟨*subtype name*⟩ ::= ⟨*type name*⟩
⟨*parent name*⟩ ::= ⟨*type name*⟩
⟨*constraint*⟩ ::= ⟨*range*⟩

Constants

⟨*constant*⟩ ::= '**constant**' ⟨*type name*⟩ ⟨*constant name*⟩ '**:=**' ⟨*expression*⟩ '**;**'
⟨*constant name*⟩ ::= ⟨*name*⟩

Functions

⟨*function*⟩ ::= ⟨*function declaration*⟩
| ⟨*function body*⟩
⟨*function prototype*⟩ ::= '**function**' ⟨*function name*⟩
'**(**' ⟨*parameters specification*⟩ '**)**' '**->**' ⟨*type name*⟩
⟨*function declaration*⟩ ::= ⟨*function prototype*⟩ '**;**'
⟨*function body*⟩ ::= '**import**' ⟨*function prototype*⟩ '**;**'
| ⟨*function prototype*⟩ ⟨*statement*⟩
⟨*parameters specification*⟩ ::= [⟨*parameter specification*⟩ ('**,**' ⟨*parameter specification*⟩)*]
⟨*parameter specification*⟩ ::= ⟨*type name*⟩ ⟨*parameter name*⟩
⟨*function name*⟩ ::= ⟨*name*⟩
⟨*parameter name*⟩ ::= ⟨*name*⟩

Statements

⟨*statement*⟩ ::= ⟨*assignment*⟩
| ⟨*if statement*⟩
| ⟨*case statement*⟩
| ⟨*while statement*⟩
| ⟨*for statement*⟩
| ⟨*return statement*⟩
| ⟨*assert statement*⟩
| ⟨*block*⟩

Assignment

⟨*assignment*⟩ ::= ⟨*variable*⟩ '**:=**' ⟨*expression*⟩ '**;**'

If-then-else

⟨*if statement*⟩ ::= '**if**' '**(**' ⟨*expression*⟩ '**)**' ⟨*true statement*⟩ ['**else**' ⟨*false statement*⟩]
⟨*true statement*⟩ ::= ⟨*statement*⟩
⟨*false statement*⟩ ::= ⟨*statement*⟩

Case

⟨*case statement*⟩ ::= '**case**' '**(**' ⟨*expression*⟩ '**)**' '**{**' (⟨*case alternative*⟩)* [⟨*default alternative*⟩] '**}**'
⟨*case alternative*⟩ ::= ⟨*expression*⟩ '**:**' ⟨*statement*⟩
⟨*default alternative*⟩ ::= '**default**' '**:**' ⟨*statement*⟩

While

⟨*while statement*⟩ ::= '**while**' '**(**' ⟨*expression*⟩ '**)**' ⟨*statement*⟩

For loop

⟨*for statement*⟩ ::= '**for**' '**(**' ⟨*iteration scheme*⟩ '**)**' ⟨*statement*⟩
⟨*iteration scheme*⟩ ::= ⟨*iteration variable*⟩ ['**,**' ⟨*iteration variable*⟩]
⟨*iteration variable*⟩ ::= ⟨*variable name*⟩ '**in**' ⟨*type name*⟩ [⟨*range*⟩]
| ⟨*variable name*⟩ '**in**' ⟨*place name*⟩
| ⟨*variable name*⟩ '**in**' ⟨*expression*⟩

Return

⟨*return statement*⟩ ::= '**return**' ⟨*expression*⟩ '**;**'

Assertion

⟨*assert statement*⟩ ::= '**assert**' '**:**' ⟨*expression*⟩ '**;**'

Block

⟨*block*⟩ ::= '**{**' (⟨*declaration*⟩)* (⟨*statement*⟩)+ '**}**'
⟨*declaration*⟩ ::= ⟨*constant declaration*⟩
| ⟨*variable declaration*⟩
⟨*variable declaration*⟩ ::= ⟨*type name*⟩ ⟨*variable name*⟩ ['**:=**' ⟨*expression*⟩] '**;**'
⟨*variable name*⟩ ::= ⟨*name*⟩

Places

⟨*place*⟩ ::= '**place**' ⟨*place name*⟩ '**{**' ⟨*place domain*⟩ (⟨*place attribute*⟩)* '**}**'
⟨*place attribute*⟩ ::= ⟨*initial marking*⟩
| ⟨*capacity*⟩
| ⟨*place type*⟩

Domain

⟨*domain*⟩ ::= '**dom**' '**:**' ⟨*domain definition*⟩ '**;**'
⟨*domain definition*⟩ ::= '**epsilon**'
| ⟨*types product*⟩
⟨*types product*⟩ ::= ⟨*type name*⟩ ('*****' ⟨*type name*⟩)*

Initial marking

⟨*initial marking*⟩ ::= '**init**' '**:**' ⟨*marking*⟩ '**;**'
⟨*marking*⟩ ::= ⟨*arc label*⟩

Capacity

⟨*capacity*⟩ ::= '**capacity**' '**:**' ⟨*expression*⟩ '**;**'

Type

⟨*place type*⟩ ::= '**type**' '**:**' ⟨*place type name*⟩ '**;**'
⟨*place type name*⟩ ::= '**process**'
| '**local**'
| '**shared**'
| '**protected**'
| '**buffer**'
| '**ack**'

Transitions

⟨*transition*⟩ ::= '**transition**' ⟨*transition name*⟩
'**{**' ⟨*transition inputs*⟩
⟨*transition outputs*⟩
[⟨*transition inhibitors*⟩]
[⟨*transition free variables*⟩]
[⟨*transition bound variables*⟩]
(⟨*transition attribute*⟩)* '**}**'
⟨*transition name*⟩ ::= ⟨*name*⟩
⟨*transition inputs*⟩ ::= '**in**' '**{**' (⟨*arc*⟩)* '**}**'
⟨*transition outputs*⟩ ::= '**out**' '**{**' (⟨*arc*⟩)* '**}**'
⟨*transition inhibitors*⟩ ::= '**inhibit**' '**{**' (⟨*arc*⟩)* '**}**'
⟨*transition attribute*⟩ ::= ⟨*transition guard*⟩
| ⟨*transition priority*⟩
| ⟨*transition description*⟩
| ⟨*safe*⟩

Arcs

⟨*arc*⟩ ::= ⟨*place name*⟩ '**:**' ⟨*arc label*⟩ '**;**'

Free variables

⟨*transition free variables*⟩ ::= '**pick**' '**{**' (⟨*free variable*⟩)* '**}**'
⟨*free variable*⟩ ::= ⟨*free variable name*⟩ '**in**' ⟨*free variable domain*⟩
⟨*free variable domain*⟩ ::= ⟨*type name*⟩ [⟨*range*⟩]
| ⟨*expression*⟩

Bound variables

⟨*transition bound variables*⟩ ::= '**let**' '**{**' (⟨*transition bound variable*⟩)* '**}**'
⟨*transition bound variable*⟩ ::= ⟨*type name*⟩ ⟨*variable name*⟩ '**:=**' ⟨*expression*⟩ '**;**'

Guard

⟨*transition guard*⟩ ::= '**guard**' '**:**' ⟨*guard definition*⟩ '**;**'
⟨*guard definition*⟩ ::= ⟨*expression*⟩

Priority

⟨*transition priority*⟩ ::= '**priority**' '**:**' ⟨*expression*⟩ '**;**'

Safe attribute

⟨*safe*⟩ ::= '**safe**' '**;**'

Description

⟨*transition description*⟩ ::= '**description**' '**:**' ⟨*string*⟩ ['**,**' ⟨*non empty expression list*⟩] '**;**'

Expressions

⟨*expression*⟩ ::= '**(**' ⟨*expression*⟩ '**)**' | ⟨*numerical constant*⟩
| ⟨*enumeration constant*⟩ | ⟨*variable*⟩
| ⟨*predecessor-successor operation*⟩ | ⟨*integer operation*⟩
| ⟨*comparison operation*⟩ | ⟨*boolean operation*⟩
| ⟨*function call*⟩ | ⟨*cast*⟩
| ⟨*if-then-else*⟩ | ⟨*structure*⟩
| ⟨*structure component*⟩ | ⟨*structure assignment*⟩
| ⟨*vector*⟩ | ⟨*vector component*⟩
| ⟨*vector assignment*⟩ | ⟨*empty list*⟩
| ⟨*list*⟩ | ⟨*list component*⟩
| ⟨*list assignment*⟩ | ⟨*list slice*⟩
| ⟨*list concatenation*⟩ | ⟨*list membership*⟩
| ⟨*empty set*⟩ | ⟨*set*⟩
| ⟨*set membership*⟩ | ⟨*set operation*⟩
| ⟨*token component*⟩ | ⟨*attribute*⟩
| ⟨*iterator*⟩

⟨*expression list*⟩ ::= ε
| ⟨*non empty expression list*⟩

⟨*non empty expression list*⟩ ::= ⟨*expression*⟩ ('**,**' ⟨*expression*⟩)*

Numerical and enumeration constants

⟨*numerical constant*⟩ ::= ⟨*number*⟩
⟨*enumeration constant*⟩ ::= ⟨*name*⟩

Predecessor and successor operators

⟨*predecessor-successor operation*⟩ ::= '**pred**' ⟨*expression*⟩
| '**succ**' ⟨*expression*⟩

Integer arithmetic

⟨*integer operation*⟩ ::= ⟨*expression*⟩ '**+**' ⟨*expression*⟩
| ⟨*expression*⟩ '**-**' ⟨*expression*⟩
| ⟨*expression*⟩ '*****' ⟨*expression*⟩
| ⟨*expression*⟩ '**/**' ⟨*expression*⟩
| ⟨*expression*⟩ '**%**' ⟨*expression*⟩
| '**+**' ⟨*expression*⟩
| '**-**' ⟨*expression*⟩

Comparison operators

⟨*comparison operation*⟩ ::= ⟨*expression*⟩ '**=**' ⟨*expression*⟩
| ⟨*expression*⟩ '**!=**' ⟨*expression*⟩
| ⟨*expression*⟩ '**>**' ⟨*expression*⟩
| ⟨*expression*⟩ '**>=**' ⟨*expression*⟩
| ⟨*expression*⟩ '**<**' ⟨*expression*⟩
| ⟨*expression*⟩ '**<=**' ⟨*expression*⟩

Boolean logic

⟨*boolean operation*⟩ ::= ⟨*expression*⟩ '**or**' ⟨*expression*⟩
| ⟨*expression*⟩ '**and**' ⟨*expression*⟩
| '**not**' ⟨*expression*⟩

Variables

⟨*variable*⟩ ::= ⟨*variable name*⟩
| ⟨*structure component*⟩
| ⟨*vector component*⟩
| ⟨*list component*⟩

Structures

⟨*structure*⟩ ::= '**{**' ⟨*non empty expression list*⟩ '**}**'
⟨*structure component*⟩ ::= ⟨*variable*⟩ '**.**' ⟨*component name*⟩
⟨*structure assignment*⟩ ::= ⟨*expression*⟩ '**::**' '**(**' ⟨*component name*⟩ '**:=**' ⟨*expression*⟩ '**)**'

Vectors

⟨*vector*⟩ ::= '**[**' ⟨*non empty expression list*⟩ '**]**'
⟨*vector component*⟩ ::= ⟨*variable*⟩ '**[**' ⟨*non empty expression list*⟩ '**]**'
⟨*vector assignment*⟩ ::= ⟨*expression*⟩ '**::**' '**(**' '**[**' ⟨*non empty expression list*⟩ '**]**' '**:=**' ⟨*expression*⟩ '**)**'

Lists

⟨*empty list*⟩ ::= '**empty**'
⟨*list*⟩ ::= '**|**' ⟨*non empty expression list*⟩ '**|**'
⟨*list component*⟩ ::= ⟨*variable*⟩ '**[**' ⟨*expression*⟩ '**]**'
⟨*list assignment*⟩ ::= ⟨*expression*⟩ '**::**' '**(**' '**[**' ⟨*expression*⟩ '**]**' '**:=**' ⟨*expression*⟩ '**)**'
⟨*list slice*⟩ ::= ⟨*expression*⟩ '**[**' ⟨*expression*⟩ '**..**' ⟨*expression*⟩ '**]**'
⟨*list concatenation*⟩ ::= ⟨*expression*⟩ '**&**' ⟨*expression*⟩
⟨*list membership*⟩ ::= ⟨*expression*⟩ '**in**' ⟨*expression*⟩

Sets

⟨*empty set*⟩ ::= '**empty**'
⟨*set*⟩ ::= '**|**' ⟨*non empty expression list*⟩ '**|**'
⟨*set membership*⟩ ::= ⟨*expression*⟩ '**in**' ⟨*expression*⟩
⟨*set operation*⟩ ::= ⟨*expression*⟩ '**or**' ⟨*expression*⟩
| ⟨*expression*⟩ '**and**' ⟨*expression*⟩
| ⟨*expression*⟩ '**-**' ⟨*expression*⟩

Function call

⟨*function call*⟩ ::= ⟨*function name*⟩ '**(**' ⟨*expression list*⟩ '**)**'

Cast

⟨*cast*⟩ ::= ⟨*type name*⟩ '**(**' ⟨*expression*⟩ '**)**'

If-then-else

⟨*if-then-else*⟩ ::= ⟨*condition*⟩ '**?**' ⟨*true expression*⟩ '**:**' ⟨*false expression*⟩
⟨*condition*⟩ ::= ⟨*expression*⟩
⟨*true expression*⟩ ::= ⟨*expression*⟩
⟨*false expression*⟩ ::= ⟨*expression*⟩

Token component

⟨*token component*⟩ ::= ⟨*token*⟩ '**->**' ⟨*component number*⟩
⟨*token*⟩ ::= ⟨*variable name*⟩
⟨*component number*⟩ ::= ⟨*number*⟩

Attributes

⟨*attribute*⟩ ::= ⟨*type name*⟩ '**'**' ⟨*type attribute*⟩
| ⟨*place name*⟩ '**'**' ⟨*place attribute*⟩
| ⟨*expression*⟩ '**'**' ⟨*container attribute*⟩
| ⟨*expression*⟩ '**'**' ⟨*list attribute*⟩
⟨*type attribute*⟩ ::= '**first**' | '**last**' | '**card**'
⟨*place attribute*⟩ ::= '**card**' | '**mult**'
⟨*container attribute*⟩ ::= '**full**' | '**empty**' | '**capacity**'
| '**size**' | '**space**'
⟨*list attribute*⟩ ::= '**first**' | '**first_index**' | '**prefix**'
| '**last**' | '**last_index**' | '**suffix**'

Iterator

⟨*iterator*⟩ ::= ⟨*iterator type*⟩ '**(**' ⟨*iteration scheme*⟩ [⟨*iterator condition*⟩] [⟨*iterator expression*⟩] '**)**'
⟨*iterator type*⟩ ::= '**forall**' | '**exists**' | '**card**' | '**mult**' | '**min**' | '**max**' | '**sum**' | '**product**'
⟨*iterator condition*⟩ ::= '**|**' ⟨*expression*⟩
⟨*iterator expression*⟩ ::= '**:**' ⟨*expression*⟩

Arc labels

⟨*arc label*⟩ ::= ⟨*complex tuple*⟩ ('**+**' ⟨*complex tuple*⟩)*

Tuples

⟨*complex tuple*⟩ ::= [⟨*tuple for*⟩] [⟨*tuple guard*⟩] [⟨*tuple factor*⟩] ⟨*tuple*⟩
⟨*tuple for*⟩ ::= '**for**' '**(**' ⟨*iteration scheme*⟩ '**)**'
⟨*tuple guard*⟩ ::= '**if**' '**(**' ⟨*expression*⟩ '**)**'
⟨*tuple factor*⟩ ::= ⟨*expression*⟩ '***'
⟨*tuple*⟩ ::= '**<(**' ⟨*non empty expression list*⟩ '**)>**'
| '**epsilon**'

State propositions

⟨*state proposition*⟩ ::= '**proposition**' ⟨*state proposition name*⟩ '**:**' ⟨*expression*⟩ '**;**'
⟨*state proposition name*⟩ ::= ⟨*name*⟩

A.2 Property specification language

Properties

⟨*property specification*⟩ ::= (⟨*property*⟩)*
⟨*property*⟩ ::= ⟨*state property*⟩
| ⟨*temporal property*⟩

State properties

⟨*state property*⟩ ::= '**state**' '**property**' ⟨*property name*⟩ '**:**' ⟨*state property definition*⟩
⟨*property name*⟩ ::= ⟨*name*⟩
⟨*state property definition*⟩ ::= ⟨*reject clause*⟩ (⟨*accept clause*⟩)*
⟨*reject clause*⟩ ::= '**reject**' ⟨*predicate*⟩ '**;**'
⟨*accept clause*⟩ ::= '**accept**' ⟨*predicate*⟩ '**;**'
⟨*predicate*⟩ ::= '**deadlock**'
| ⟨*state proposition name*⟩

Temporal properties

⟨*temporal property*⟩ ::= '**ltl**' '**property**' ⟨*property name*⟩ '**:**' ⟨*temporal expression*⟩ '**;**'
⟨*temporal expression*⟩ ::= '**(**' ⟨*temporal expression*⟩ '**)**'
| '**true**'
| '**false**'
| ⟨*state proposition name*⟩
| '**not**' ⟨*temporal expression*⟩
| ⟨*temporal expression*⟩ '**or**' ⟨*temporal expression*⟩
| ⟨*temporal expression*⟩ '**and**' ⟨*temporal expression*⟩
| '**[]**' ⟨*temporal expression*⟩
| '**<>**' ⟨*temporal expression*⟩
| ⟨*temporal expression*⟩ '**until**' ⟨*temporal expression*⟩

Appendix B

Gnu general public license

Version 2, June 1991

59 Temple Place - Suite 330, Boston, MA 02111-1307, USA

PREAMBLE

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or

work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   (a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   (b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   (c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

   In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   (a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Index