Maria

Modular Reachability Analyzer for Algebraic System Nets
20 June 2003, Maria Version 1.3.4

by Marko Mäkelä

Introduction

Maria, or *Modular Reachability Analyzer*, is a reachability analyzer for Algebraic System Nets. It will generate reachability graphs for Algebraic System Nets, detect deadlocks and check properties expressed using temporal logic formulae.

Maria is remotely based on Prod, a reachability analyzer for Proposition/Transition Nets developed earlier at the Laboratory for Theoretical Computer Science. Its data type system and expression syntax are heavily inspired by the C programming language. In order to understand this manual, you should be familiar with the basic concepts of Petri Nets and also have some knowledge in C.

Maria was written by Marko Mäkelä in a research project at the Helsinki University of Technology in the Laboratory for Theoretical Computer Science. The project is financed by the National Technology Agency of Finland (TEKES), Nokia Research Center, Nokia Networks, the Helsinki Telephone Corporation and the Finnish Rail Administration.

Many of the ideas implemented in Maria were brought up in discussions among the laboratory staff. Ideas expressed by Dr. Kimmo Varpaaniemi, Dr. Nisse Husberg, Keijo Heljanko and Tommi Junttila have influenced the design. Dr. Varpaaniemi sketched the first version of the unification algorithm (see Section 3.1 [Unification], page 52), and he helped in debugging by stressing the analyzer with quite obscure examples. Also Simo Blom, who was the first one to use the analyzer for something real, reported some bugs, which have now been fixed.

The model-checking functionality (see Section 3.2 [Model Checking], page 54) is based on the ideas of Keijo Heljanko and others, and the algorithms were initially implemented by Timo Latvala.

This edition corresponds to version 1.3.4 of Maria.

1 The Net Description Language

Petri Nets are often represented as directed bipartite graphs using a graphical notation. An entirely graphical notation only works for relatively simple nets that can be represented on one sheet of paper. We decided to use a purely textual notation, since it avoids many problems, such as creating an optimal graphical layout for an automatically generated Petri Net model and creating a graphical user interface that works flawlessly on various hardware and software platforms.

1.1 Design Criteria

As an admirer of the programming languages C and C++, I decided to make the net description language resemble C as closely as possible. Users familiar with C should feel comfortable with the way data types are defined and expressions are written in the net description language.

In the following, we will present the grammar of the net description language using regular expressions (see section "Patterns" in *The Flex Manual*) and the Extended Backus−Naur Form (see section "Languages and Context-Free Grammars" in *The GNU Bison Manual*).

1.2 Lexical Conventions

The lexical conventions of the Maria Petri Net description language determine how to format net descriptions in text files, including the mechanisms for embedding explanatory comments.

1.2.1 Formatting

The net description language is not sensitive to the presence or absence of white space between language elements provided that all keywords and identifiers are distinguished. Thus the user has the same degree of freedom that C and C++ allow in formatting code. White space includes these characters: space (‘' '’), newline (‘'\n'’), carriage return (‘'\r'’), form feed (‘'\f'’), horizontal tabulator (‘'\t'’) and vertical tabulator (‘'\v'’).

1.2.2 Comments

Comments are indicated as they are in the C++ programming language. Two contiguous slashes (‘//’) indicate the start of a comment that continues to the end of the current line. A slash immediately followed by an asterisk (‘/*’) indicates a comment that continues until the reverse sequence (‘*/’) is encountered. Comments may not be nested, and comments are interpreted only between language elements (e.g., not inside names enclosed in double quotes).

1.2.3 Lexical Tokens

Lexical tokens are the atomic constituents of the language, consisting of characters or character sequences.

1.2.3.1 Reserved Words

There are quite a few reserved words in the net description language:

`atom`	`cardinality`	`const`	`deadlock`
`empty`	`enabled`	`enum`	`equals`
`false`	`fatal`	`gate`	`hide`
`id`	`in`	`infinite`	`intersect`
`is`	`map`	`max`	`min`
`minus`	`out`	`place`	`prop`
`queue`	`reject`	`release`	`stack`
`strongly_fair`	`struct`	`subnet`	`subset`
`trans`	`true`	`typedef`	`undefined`
`union`	`until`	`weakly_fair`	

In order to use any of these words as an identifier (see Section 1.2.3.4 [Identifiers], page 4), you will have to enclose it in double quotation marks (‘`"`’).

1.2.3.2 Numeric Constants

Maria has three interchangeable ways of entering integer numeric constants. The constants can be entered using one of three different notations: with decimal numbers (‘`[1-9][0-9]*`’), octal numbers (‘`0[0-7]*`’) or hexadecimal numbers (‘`0x[0-9a-fA-F]+`’). When a numeric constant is too long to fit in the internal representation, the lexical analyzer will detect it and issue a diagnostic message. Decimal numbers are always unsigned, but the hexadecimal and octal representations are translated directly to the system-dependent internal representation, which usually is a 32-bit word interpreted as a signed integer using two’s complement arithmetic.

1.2.3.3 Character Constants

Character constants are just like in the C programming language: a single character enclosed in apostrophes (‘`'`’). The backslash (‘`\`’) is used for entering special characters:

‘`\a`’ alert, bell (*BEL*)

‘`\b`’ backspace (*BS*)

‘`\t`’ horizontal tabulator (*HT*)

‘`\n`’ newline, line feed (*LF*)

‘`\v`’ vertical tabulator (*VT*)

‘`\f`’ form feed (*FF*)

‘`\r`’ carriage return (*CR*)

‘`\[0-7]{1,3}`’
octal notation

‘`\x[0-9a-fA-F]{1,2}`’
hexadecimal notation

‘`\c`’ character *c* (`c != '\n'`)

Any other characters than the apostrophe or the backslash can be entered verbatim between the apostrophes.

Any non-special character quoted with the backslash will be entered as such, i.e. the backslash will be ignored. Thus, ‘`'\c'`’ is equivalent to ‘`'c'`’. The line break character is a special case. In order to maintain consistence with the quoted identifiers discussed below, a backslash followed by a line break and any amount of white space containing no line breaks will be ignored.

1.2.3.4 Identifiers

Maria uses textual names for identifying places, transitions, data types, variables, enumeration constants and many other things. None of the identifiers become reserved words. For instance, ‘`bool`’ may be a type name, a place name or the name of a structure component. Anything of the form ‘`[A-Za-z_][A-Za-z0-9_]*`’ that is not a reserved word is an identifier.

Identifiers can also be enclosed in double quotation marks. As with single-character constants, only the backslash and the quotation mark must be escaped with a backslash, and the backslash notation (see Section 1.2.3.3 [Character Constants], page 3) can be used for entering non-printable characters. Non-printable characters need not be escaped, though. The only character that is not allowed in identifiers is the *`NUL`* character.

The backslash character (‘\’) can be used to break up long identifiers. A backslash followed by a line break and any amount of white space containing no line breaks will be ignored in the input. Also, backslashes can be used to quote the following character. For instance, ‘`in\ k`’ is equivalent to ‘`"in k"`’.

In double quotes, the newline character is just as significant as any other character. Sometimes one wants to split a long quoted identifier to several lines without the line breaks being significant. This can be achieved by putting a backslash immediately before the line break. The lexical analyzer will ignore the backslash, the line break and the immediately following white space (not line break).

1.2.4 Preprocessor Directives

The language contains a subset of the directives implemented in the C language preprocessor. Conditional compilation and macro definitions are not supported in the current version.

Preprocessor directives are indicated with a number sign (‘`#`’) located in the first (leftmost) column. The number sign may be followed by any amount of lexical comments and other white space than newline. The line, which must be terminated with a newline, must contain exactly one preprocessor directive.

1.2.4.1 Embedding Other Files: ‘`#include`’

The ‘`#include`’ directive works just like in the C preprocessor, except that the file name must be enclosed in double quotes and never in angle brackets (‘`<`’ and ‘`>`’). The string in double quotes is interpreted as a quoted identifier (see Section 1.4.2.5 [Identifier], page 14).

1.2.4.2 Conditional Processing

The '#ifdef', '#ifndef', '#else' and '#endif' directives work just like in the C preprocessor, expecting an argument of the form '[A-Za-z_][A-Za-z0-9_]*'. Note that there is no '#if' directive and that the '#define' directive only takes one argument, the name of the symbol. The preprocessor symbols are not macros; no macro expansion will take place.

Conditional processing can be used to avoid problems with multiple inclusions of a file or to add some parameterization to the net model. Preprocessor symbols can also be defined on the command line.

1.2.4.3 Setting the Line Number: '#line'

Code generation tools usually generate '#line' directives for excerpts that are to be embedded verbatim in the generated code. This allows the compiler to refer to the relevant input file of the code generator in its diagnostic messages.

The Maria languages implement the '#line' directive in order to better support the diagnostics of automatically generated net descriptions. The 'line' keyword is followed by a numeric constant and a character string constant indicating the line number and the file name of the following line.

1.2.4.4 Preprocessor Comment: '#!'

The special preprocessor directive '#!', which causes the rest of the line to be ignored, was added in order to make it possible for Maria input files to also be executable scripts in systems having an appropriate `exec` system call.

1.3 Constructs for Defining Nets

1.3.1 Type Definitions: 'typedef'

In the Maria languages, only the predefined data types 'bool', 'int', 'unsigned' and 'char' can be used without naming them using the grammatical construct

```
type:   TYPEDEF typedefinition name
```

inspired by C. The built-in type names are not reserved words. For instance, any reference to the type 'int' following the definition `typedef int (-64..63) int;` will refer to an integer representable with 7 bits.

For a comprehensive description of the data type system, see Section 1.4 [Data Types], page 12. Here we will only present the syntax, not the semantics.

Some of the syntax for defining data types resembles the C programming language very closely:

```
typedefinition:
        ENUM '{' enum_item ( delim enum_item )* '}'
        |
        STRUCT '{' comp_list '}'
        |
        UNION '{' comp_list '}'
        |
        typereference
typereference:
        name
enum_item:
        name [ [ '=' ] number ]
comp_list:
        comp (delim comp)* [ delim ]
comp:
        typedefinition name
number:
        expr
delim:
        ','
        |
        ';'
```

The extra leaf data types include an empty '`struct`', often used for denoting black tokens, and an *identifier* type used for identifying e.g. processes or objects. Note that if you intend to compile the model (see Section 2.1.2 [Maria Options], page 29), the empty '`struct`' does not work on all C compilers.

```
typedefinition:
        STRUCT '{' '}'
        |
        ID '[' number ']'
```

It is a good idea to define an alias (see Section 1.3.2 [Functions], page 7) for the black token, so that its definition can be changed easily:

```
typedef struct {} token;
// typedef unsigned (0) token;
token token = <token;
```

The alert reader may have noticed that we have not introduced arrays yet. Arrays in Maria are not necessarily indexed by integers, but by any type having a limited set of possible values. One can also define a finite-capacity buffer that uses either a queue or a stack discipline.

```
typedefinition:
        typedefinition '[' typedefinition ']'
        |
        typedefinition '[' QUEUE number ']'
        |
        typedefinition '[' STACK number ']'
```

Last but not least, it is possible to limit the set of possible values for a type by defining *constraints*, unions of closed or semi-open ranges of constant values.

```
typedefinition:
        typedefinition constraint
constraint:
        '(' range (delim range)* ')'
range:
        value
        |
        '..' value
        |
        value '..' value
        |
        value '..'
value:
        expr
```

1.3.2 Function Definitions

Commonly used expressions can be defined as functions, which can be viewed as macros with strictly typed parameters. Functions are defined in the following way:

```
function:
        typereference name ('='|'()') formula
        |
        typereference name '(' param_list ')' formula
param_list:
        [ param_list_item (delim param_list_item)* ]
param_list_item:
        typereference name
```

The function name and the optional parameter list are followed by a formula that usually refers to all the function parameters. When the function is "called" (the macro is expanded), each instance of a named parameter will be substituted with the corresponding expression passed as an argument to the function. Parameterless functions are practical for defining constants.

Functions can be defined both in the global scope and in the transition scope. Functions defined in the declaration block of a transition will only be accessible to that transition, and they will temporarily override a global function definition.

1.3.3 Place Definition: 'place'

All persistent data in Petri Nets is stored in *places* that can contain a number of *tokens*, which in the case of Algebraic System Nets have values. When a place is defined, it is given a unique name, and it is assigned a type.

```
place:
        PLACE name constraint* typedefinition
        [ CONST ] [ ':' marking_list ]
```

It is possible to define a *capacity constraint*, a non-negative integer constraint (see Section 1.4.4 [Constraints], page 16) that constrains the total number of tokens the place can contain. Note that the constraint does not need to be of the form '`(0..`*n*`)`' for some positive integer *n*; it can be e.g. '`(1)`' if the place always contains exactly one token, or '`(0,2,4)`'. Defining constraints has two major advantages. First, they help to catch errors in the model. Second, analysis algorithms may benefit from them, and resources can be saved when maintaining the reachability graph (see Appendix B [Graph Files], page 66).

Places can be assigned an *initial marking*, a multi-set valued expression (see Section 1.6 [Multi-Sets], page 24) that evaluates to the collection of tokens that will be assigned to the place in the system's initial state. To parameterise initial markings, you may use the multi-set summation operator.

Sometimes, it is necessary to introduce control places in the model whose contents remains constant. It would be unnecessary to include such places in the representation of model states, or in the unfolding (see Section 2.2.2.3 [Unfold], page 38) of the model. The keyword '`const`' in the place definition indicates a constant place.

When the marking of a place is a function of the marking of other places, the place is called a *redundant place*. Examples of such places include counters and complement places. It is possible to identify such places by writing initialization expressions that make use of the '`place` *`name`*' operation (see Section 1.6 [Multi-Sets], page 24). Doing so not only reduces disk space consumption; Maria will also check that such invariant properties hold in all states it generates.

1.3.4 Transition Definition: '`trans`'

The state of a Petri system is changed by *firing* transitions, removing tokens from their *input places* and adding tokens to their *output places*. A transition is *enabled* if its each input place contains at least the amount of tokens the transition is about to remove. Only enabled transitions may be fired.

```
transition:
        TRANS [ ':' ] name [ '!' ] trans*
trans:
        '{' [ var_expr (delim var_expr)* [ delim ] ] '}'
        |
        IN trans_places
        |
        OUT trans_places
        |
        GATE expr (',' expr)*
        |
        HIDE expr
        |
        STRONGLY_FAIR expr
        |
        WEAKLY_FAIR expr
        |
        ENABLED expr
```

```
                |
                ':' [ TRANS ] name
                |
                NUMBER
        var_expr:
                [ HIDE ] typereference name
                |
```

Maria supports a form of *transition fusion*, which is a key feature to constructing models in a modular way (See also see Section 1.3.5 [Subnets], page 10). A transition whose name is preceded with a colon (`trans :callee`) is not an actual transition but it designates a kind of a macro or a function body that can be substituted to other transitions, like this: `trans caller:trans callee`. The Maria parser would merge the definitions of `trans :callee` to the definition of `trans caller`.

There is a simple priority method in the search algorithm of Maria that works as follows. When computing the successors of a marking, Maria investigates the transitions in the order they were defined in the model, from top to bottom. Whenever a transition having a nonzero priority class is found to be enabled, no further transitions of other priority classes will be analyzed in the marking.

The default priority class is zero. It is worth noting that non-prioritized transitions that are defined before any prioritized transitions are completely independent of other transitions in the model. Any non-prioritized transitions that are defined after prioritized ones will be analyzed only if no prioritized transitions are enabled.

The priority class can be specified by writing a non-negative integer number after the name of the transition. The exclamation point '`!`' is an alternative mechanism for providing backward compatibility. It assigns a nonzero priority class to the transition from a global counter and then decrements the counter by one, so that the next transitions marked with '`!`' will be assigned other priority classes.

The '`hide`' keyword affects the output of labelled state transition systems (see Section 2.2.2.4 [LSTS], page 38) and the '`-Y`' command line option for suppressing hidden states (see Section 2.1 [Invoking Maria], page 29). A transition instance is hidden (renamed to the special "tau" action) if the hiding condition (a truth-valued expression on the transition variables) holds. Transition variables can be omitted from action names by declaring them with the '`hide`' keyword.

1.3.5 Defining Subnets for Modular State Space Exploration

When a system being modelled consists of a number of loosely synchronised processes, it is often useful to distinguish between the internal actions of these processes and the actions that model the communication or synchronisation of the processes.

In Maria, it is possible to define tree-like hierarchies of high-level nets. In the outer-level net, all transitions are *visible*, that is, their occurrences directly corresponds to edges in the global reachability graph (synchronisation graph).

In a subnet, defined within a '`subnet`' block, normal transitions model internal actions, which do not show up in the synchronisation graph. Synchronisation or communication of subnets is modelled with transition fusion (see Section 1.3.4 [Transitions], page 8). When a transition in a subnet calls a transition in its parent net, its occurrences will show in the state space of the parent net.

When the model contains subnets, modular analysis should be applied (command line option '`-R`' or '`--modular`'; see Section 2.1 [Invoking Maria], page 29).

```
subnet:
        SUBNET [ name ] '{' net '}'
```

1.3.6 On-the-Fly Verification

1.3.6.1 Verifying Safety Properties

The 'reject' and 'deadlock' statements are used in conjunction with Boolean conditions on markings. When these statements are used, the reachability analyzer reports an error whenever

1. a state violating the 'reject' formula has been generated
2. the 'deadlock' formula holds in a marking where no transitions are enabled

If the formula cannot be evaluated, the analysis will be stopped. Thus, commanding

```
deadlock fatal;
```

will cause the analysis to stop when a deadlock is reached. The following construct can be used to stop the analysis when an undesired state is reached:

```
reject place fork equals empty && fatal;
```

An alternative mechanism for specifying undesired states is to define "assertion" transitions with 'undefined' or 'fatal' in their gate expressions (see Section 1.3.4 [Transitions], page 8).

```
verify:
        REJECT expr
        |
        DEADLOCK expr
```

1.3.6.2 Defining Fairness Constraints

It is possible to define fairness sets of transition instances to guide the on-the-fly model checker. Such sets may contain instances of a single transition (see Section 1.3.4 [Transitions], page 8) or of several transitions. The latter case is handled by identifying the transition instances that are to be included to a set using qualifier expressions, which consist of transition names followed by Boolean conditions for transition variables.

A generic fairness set definition consists of one of the reserved words 'strongly_fair' and 'weakly_fair' followed by a list of qualifier expressions. Each qualifier expression identifies some transition instances that are to be treated fairly. All instances fulfilling a qualifier expression will be treated as one fairness set.

The 'enabled' keyword allows the definition of enabledness sets of transition instances. If no transition instances belonging to an enabledness set are enabled, the set will be reported at the end of the analysis. Use the 'dump' command (see Section 2.2.2.2 [Dump], page 37) to see the enabledness set numbers.

```
fairness:
        STRONGLY_FAIR qual_expr ( ',' qual_expr )*
        |
        WEAKLY_FAIR qual_expr ( ',' qual_expr )*
        |
        ENABLED qual_expr ( ',' qual_expr )*
```

```
qual_expr:
        TRANS name [ ':' expr ]
        |
        '(' qual_expr ')'
        |
        '!' qual_expr
        |
        qual_expr '&&' qual_expr
        |
        qual_expr '^^' qual_expr
        |
        qual_expr '||' qual_expr
        |
        qual_expr '<=>' qual_expr
        |
        qual_expr '=>' qual_expr
```

Qualifier expressions may also be quantified. The semantics of the multi-set summation operation is that each summand is associated with a new fairness set. Universal and existential quantification have their normal semantics, i.e. they are translated into chains of conjunctions or disjunctions.

```
qual_expr:
        typereference name [ '(' expr ')' ] ':' qual_expr
        |
        typereference name [ '(' expr ')' ] '&&' qual_expr
        |
        typereference name [ '(' expr ')' ] '||' qual_expr
```

1.3.6.3 Specifying State Propositions for LSTS Output

The labelled state transition systems that Maria is able to output (see Section 2.2.2.4 [LSTS], page 38) have a notion of state propositions. As they do not have a natural counterpart in a high-level Petri net, a special construct has to be used for specifying them.

```
proposition:
        PROP name ':' expr
```

The 'prop' definition does not affect anything else except the output of labelled state transition systems and the resolution of identifiers in the query language. The expression must be a truth-valued state formula.

1.4 Data Types

In programming languages, user-defined structural data types became popular in the 1970s with the success of C and other high-level languages. The original Petri Net formalism did not make use of any data types or algebraic sorts, and its high-level variations usually keep the data types pretty simple, often restricted to tuples of integer numbers or enumerated data types.

In order to efficiently analyze the behavior of parallel programs written in a high-level language, there must exist a straightforward translation from the data types in the source formalism to the types supported by the analyzer. Using tuples of integers, one can emulate simple higher-level types, such as a unidimensional array of a leaf type, but anything beyond that is next to impossible.

The data type system in Maria holds the comparison with any high-level programming language. The only thing that is missing is pointers or references, which would entirely break the locality principle of Petri Nets.

1.4.1 Background

The Maria software stores data in two forms: as trees formed by C++ objects, which are easy to manipulate but use up much space, or as compact bit strings. Converting structured values to a fixed-length bit string and vice versa requires that the number of all possible values of a type is known and that there exists a total order among the values.

For supporting iteration through all values of a type, there are successor and predecessor functions and functions for determining the smallest and the largest value of a type. These are in harmony with the conversion functions: the numeric representation of the smallest value of a type is 0, and its successor is 1 (unless the type only has one value), and so on.

1.4.2 Leaf Types

Data types that do not have any further structure are called *leaf types*. In Maria, all leaf types can be represented using a machine word, an unsigned integer of usually 32 bits.

1.4.2.1 Integer Types

There are two predefined integer types: ‘`int`’, a signed integer in the range `INT_MIN` to `INT_MAX`, as defined by ‘`<limits.h>`’ in C, and ‘`unsigned`’, an unsigned integer in the range `0` to `UINT_MAX`.

Integer literals are numeric constants, given by the regular expressions ‘`[1-9][0-9]*`’, ‘`0[0-7]*`’ and ‘`0x[0-9a-fA-F]+`’. Negative decimal numbers are formed using the unary ‘`-`’ operator.

The unconstrained built-in integer types have `INT_MAX-INT_MIN+1` or `UINT_MAX+1` possible values, both usually 2^{32}. The order among the values is determined by integer arithmetics. All arithmetic operations in the expression evaluator are performed using the unconstrained integer type. Binary operators require their operands to be either both signed or both unsigned.

1.4.2.2 Boolean Type

The Boolean type ‘`bool`’ is for storing truth values. It has two literals: ‘`false`’ (the smallest value) and ‘`true`’.

1.4.2.3 Character Type

Characters internally use the `unsigned char` type in C++. Analogously with the unsigned integer type, the smallest value is `0` and the largest `UCHAR_MAX`, and the total number of different values is `UCHAR_MAX+1`, usually 2^8.

1.4.2.4 Enumerated Type

There are two variants of the enumerated type. A constrained enumerated type acts just like an integer having some named constants. In the following we will concentrate on the unconstrained variant of the enumerated type.

The domain of an unconstrained enumeration ranges from the smallest enumeration constant to the largest one. The order among the enumeration constants is determined by their integer value. Enumeration constants whose value is not explicitly specified in the type definition get their values just like in the C programming language: it is the successor of the value given to the last declared constant. By default, the first constant will get the value 0.

1.4.2.5 Identifier Type

In SDL, there is a data type for process identifiers. Values of this type can only be compared for equality and inequality, and there is an operator for obtaining a new identifier value. SDL assumes an unlimited pool of distinct identifier values: the operator for obtaining a new value will always return something that is different from all previous values. This is impossible for any practical system. The identifier type in Maria has otherwise the same properties as the one in SDL, but one must declare the size of the identifier pool when declaring an identifier type.

Internally, identifier values are unsigned integers ranging up to the size of the identifier pool, exclusive. The operator for obtaining a new identifier value has not been implemented yet, but quantification (see Section 1.6 [Multi-Sets], page 24) is possible. In the future, it is intended to implement symmetry reductions of the state space, making use of the properties of the identifier type.

1.4.3 Composite Types

The leaf data types represented in previous section are adequate for representing any kind of data. Composite data types can be viewed as syntactic sugar: by defining structural types, one can group data items together, which can drastically simplify the notation required for referring to the data items.

Composite types in Maria are constructed in a truly recursive way. There are no arbitrary limits. For example, it is entirely possible to define a buffer of buffers containing a union of an array and a structure.

1.4.3.1 Structure

A structure type describes a sequentially allocated set of member objects, each of which has a distinct name and possibly distinct type. The order of structure values is determined in little-endian way: the most significant component is stored last. For instance, the four values of the type '`struct{bool a;bool b}`' are ordered '`{false,false}`', '`{true,false}`', '`{false,true}`' and '`{true,true}`'.

A structure type that has n members, each member i having c_i possible values, has

$$\prod_{i=1}^{n} c_i$$

possible values. An empty structure has only one possible value, '`{}`'.

1.4.3.2 Union

Often there is a need to pass a parameter whose type is determined dynamically. The tagged union type in Maria serves exactly this purpose. It describes an overlapping nonempty set of member objects, each of which have a name and a possibly distinct type. Whenever a union value is initialized, also the union component must be identified.

The union can be viewed as a special kind of structure of two components: the actual value inside the union and the identifier of the component to which the value belongs. For instance, the values of the type '`union{bool a;struct{} b;}`' are ordered '`a=false`', '`a=true`' and '`b={}`'. A union type that has n members, each member i having c_i possible values, has

$$\sum_{i=1}^{n} c_i$$

possible values.

1.4.3.3 Array

An array type describes a contiguously allocated nonempty set of objects with a particular member object, called the *element type*. The members (elements) of an array are accessed by indexing the array with values of the *index type* of the array. The number of possible values in the index type determines the number of elements in the array.

The order of values in an array type is determined in the little-endian manner. The four values of the type '`bool[bool]`' are ordered '`{false,false}`', '`{true,false}`', '`{false,true}`' and '`{true,true}`'.

An array whose element type has c_e possible values and index type c_i possible values, has

$${c_e}^{c_i}$$

possible values.

1.4.3.4 Buffer (Queue or Stack)

Communication protocols use queue-like transmission buffers heavily. Many algorithms and the translation of procedural programming languages to Petri Net models require stack-like buffers. The buffer type in Maria has a maximum size, and it has two variants, queue and stack. A buffer is much like an array, but it may contain a variable number of items.

The order of values is determined in the little-endian manner. Shorter buffers come first. For instance, the values of the type '`bool[queue 2]`' are ordered '`{}`', '`{false}`', '`{true}`', '`{false,false}`', '`{true,false}`', '`{false,true}`' and '`{true,true}`'.

A buffer of at most n elements whose element type has c_e possible values has

$$\sum_{i=0}^{n} c_e^i = \frac{c_e^{n+1} - 1}{c_e - 1}$$

possible values.

1.4.4 Constraints

When performing reachability analysis, it is often desirable to limit the analysis in many ways to address the problem known as the *state space explosion*. One of the ways is to limit the domain of data types. Instead of considering all 32-bit integer numbers, one may restrict the analysis to numbers ranging from 0 to 10, for instance.

In addition to limiting the search, type constraints can act as a valuable aid for detecting errors in system models. The expression evaluator will detect and report *constraint violations* whenever a subexpression evaluates to an unconstrained value. The successor and predecessor functions will, however, conveniently wrap around, so that the successor of the largest value of a type is the smallest value.

Constraints can be applied to all types whose values can be expressed with constant literals.[1] Not only leaf types can be constrained. For instance, it is possible (while not necessarily sensible) to define a type '`bool(false)[queue 347](..{false})[int(33101)]`', a single-element array of a buffer having two possible values: '`{{}}`' and '`{{false}}`'.

The net description language parser computes unions and intersections of value ranges while parsing constraints, which are internally stored as unions of disjoint ranges. In addition, it will combine adjacent constraints and eliminate overlapping constraints. The type '`int(1..4,3,5)`' is thus equivalent with the types '`int(1..)(..5)`' and '`int(1..5)`', the canonical form.

1.5 Expressions and Formulae

Expressions form the core of any language. One of the design goals of the Maria reachability analyzer was to make expressions as rich as possible, to add expressive power to

[1] The identifier type does not have any literals, and thus it is impossible to define constraints for types containing it.

the language and to make the language attractive to people who are familiar with and accustomed to modern high-level programming languages.

Like the data type system, the expression system of Maria has been greatly inspired by that of the programming language C. There are no pointer operations and no expressions with side-effects, and some operators of our language have no direct counterpart in C.

1.5.1 Literals

When an expression is viewed as a tree, the leaves of the tree are called *literals*. There are three kinds of literals in Maria expressions: constant values, variable names and the reserved words '`undefined`' and '`fatal`'. Also invocations of functions with zero arguments, also called *nullary functions* or *named constants*, could be viewed as literals, but they can expand to more complex expressions or formulae.

1.5.1.1 Constants

The constants in the Maria languages are type-specific. The Boolean type (types derived from the built-in '`bool`' type) has two constants: '`false`' and '`true`'. Character types ('`char`') uses single character constants enclosed in single quotes (see Section 1.2.3 [Lexical Tokens], page 2), and integer types ('`int`' and '`unsigned`') use decimal, octal or hexadecimal numbers with a notation familiar from the programming language C.

There are three unary pseudo-operators that deal with type names and yield constant. The number-of-values operator '`#`' can be used to refer to the number of possible values a type can receive. For instance, '`#bool`' is equivalent to '`2`', unless you have redefined the built-in type '`bool`' to be something else. The operators '`<`' and '`>`' refer to the smallest and to the largest value of a type, respectively. For instance, you can use '`<bool`' interchangeably with '`false`', unless the type has been redefined.

The constants of enumerated types are written either using numbers, just as with integer types, or using the names of the enumeration symbols. The names do not have global scope, and in the cases when the parser cannot determine the type from the context, you must use the type casting operator (see Section 1.5.2.6 [Type Casting], page 21). If you want an enumeration symbol to have global scope, you can define it as a nullary function or named constant:

```
typedef enum { a, b, c } enum_t;
enum_t a = is enum_t a;
reject <enum_t != a;
```

Constants of compound types are written using the respective expressions for creating compound values, restricting the literals in the expressions to constants. The expressions will be evaluated while parsing them and replaced with corresponding constants. This is called *constant folding*.

1.5.1.2 Variables

Variables make expressions behave dynamically. In high-level Petri Nets, variables make the arc expressions of transitions evaluate in different ways. Each combination of variable–value pairs (usually referred to as *valuation*) that enables a transition is called an enabled *instance* for the transition.[2] Enabled transition instances are sought in a process called *unification* (see Section 3.1 [Unification], page 52). Variables cannot be explicitly assigned to in the high-level Petri Net formalism.

Variables can be declared either explicitly e.g. in the declaration blocks of transitions, or implicitly in the input arc expressions of transitions.

1.5.1.3 Dynamic Errors

The ‘`undefined`’ and ‘`fatal`’ keywords can be used to catch dynamic errors in the model. Both can be seen as type-less nullary operators. When the ‘`undefined`’ symbol is evaluated while searching for enabled transition instances, the valuation built so far will be marked as erroneous and the instance will not be fired. The ‘`fatal`’ keyword is similar, but evaluating it will cause the whole analysis to be aborted.

These two keywords are usually used on the right-hand-side of the if-then-else operator or of the logical ‘`&&`’, ‘`||`’ or ‘`=>`’ operators.[3] To improve readability, one could define an ‘`assert()`’ macro. Unlike its counterpart in the C library, the following macro will return a value.

```
bool assert (bool expr) expr || fatal;
place p unsigned: 42;
trans t in { place p: p; }
gate assert (p == 42);
```

The search for enabled transition instances is an iterative process. Because of this, expressions containing ‘`undefined`’ and ‘`fatal`’ keywords may be evaluated before the transition instance is complete. If this is not desirable, the expression for catching dynamic errors should be placed on an output arc, perhaps on the right-hand-side of an if-then-else operator.

If you use either keyword in a gate expression as the right-hand-side of the ‘`&&`’ operator, you’d better declare the expression atomic by enclosing it with ‘`atom()`’, so that the parser will not split the gate expression into two, which would break the short-circuit evaluation of the ‘`&&`’ operator.

1.5.2 Operators

The operator precedence in the Maria languages is as follows. Each table row forms a precedence class.

<table>
<tr><td>‘?:’</td><td>‘!’</td><td>‘:’</td></tr>
<tr><td>(selection)</td><td>(output)</td><td>(multi-sets)</td></tr>
<tr><td>‘until’</td><td>‘release’</td><td>‘=’</td></tr>
<tr><td>‘=>’</td><td>‘<=>’</td><td></td></tr>
<tr><td>‘||’</td><td></td><td></td></tr>
<tr><td>‘^^’</td><td></td><td></td></tr>
<tr><td>‘&&’</td><td></td><td></td></tr>
</table>

[2] Also the terms *binding* and *firing mode* have been used.

[3] This relies on *short-circuit evaluation*: if the left-hand-side of one these operators alone can determine the result of the operation, the right-hand-side will not be evaluated.

‘<>’	‘[]’	‘()’		
‘|’				
‘^’				
‘&’				
‘!=’	‘==’			
‘>=’	‘<=’	‘<’	‘>’	
‘<<’	‘>>’			
‘+’	‘-’			
‘*’	‘/’	‘%’		
‘#’ (binary)	‘is’			
unary: ‘~’, ‘-’, ‘!’	‘#’	‘<’, ‘>’, ‘|’, ‘+’	‘*’, ‘/’, ‘%’	
‘cardinality’	‘max’	‘min’	‘subset’ (ternary)	‘map’
‘equals’				
‘subset’				
‘minus’	‘union’			
‘intersect’				
‘atom’	‘is’ (cast)			
‘.’	‘[’	‘(’		

Some operators have several meanings. For instance, there are two variants of the ‘is’ operator, one that performs type conversions, and another that determines whether a union component is active:

```
expr:
        IS typereference expr
        |
        expr IS name
```

1.5.2.1 Integer Arithmetic

For performing integer arithmetic, the language contains all the integer operators of C: negation (sign change, unary ‘-’), bitwise complementation (unary ‘~’), basic arithmetics (‘+’, ‘-’, ‘/’, ‘*’, ‘%’) and bit operations (‘&’, ‘|’, ‘^’, ‘<<’, ‘>>’).

Actually there are two sets of integer operators: signed and unsigned operators. Both operands must be either signed or unsigned, and the result is accordingly signed or unsigned. Unsigned arithmetics is a little faster than signed arithmetics. Numeric constants, unsigned by default, are automatically converted to signed integers. For binary integer operators, the type of the first operand determines whether the operation is signed or unsigned.

All integer operators in Maria have been implemented in terms of the corresponding C++ operators, but some error handling has been added. The operators whose result can exceed the limits of the built-in integer type will detect overflows. These operators are negation, addition, subtraction and multiplication. The division and modulus operators will check for division by zero, and the bit shifting operators will ensure that the amount to be shifted does not exceed the size of the integer type in bits. Last but not least, if the result type has a constraint, the result will be checked against it.

1.5.2.2 Successor and Predecessor

The successor and the predecessor are defined for all ordered types. All other types than the identifier type and structural types containing an identifier component are ordered. The successor of the largest value of a type is the smallest value, whose predecessor in turn is the largest value.[4] No errors can occur while computing the successor (unary ‘`+`’) or predecessor (unary ‘`|`’).

1.5.2.3 Comparison

The language contains the usual comparison operators (‘`==`’, ‘`!=`’, ‘`<`’, ‘`<=`’, ‘`>=`’ and ‘`>`’). Equality and inequality comparisons are available for all types, while other comparisons are only available for ordered types. The only error that can occur in a comparison is a constraint violation, in case the result type is constrained.[5]

1.5.2.4 Boolean Logic

Boolean logic in Maria has the familiar operators from C: negation (unary ‘`!`’), conjunction (‘`&&`’) and disjunction (‘`||`’). There is also some syntactic sugar: implication (‘`a=>b`’ is equivalent to ‘`!a||b`’), logical equivalence (‘`a<=>b`’ is equivalent to ‘`(a&&b)||!(a||b)`’) and logical exclusive or (‘`a^^b`’, equivalent to ‘`!(a<=>b)`’).

Also, the language supports universal and existential quantification, and translates them to conjunctions and disjunctions:

```
formula:
        typereference name [ ’(’ expr ’)’ ] ’&&’ formula
        |
        typereference name [ ’(’ expr ’)’ ] ’||’ formula
```

For instance, the existential quantification

```
char a (a>=’a’ && a<=’c’) || b==a
```

expands to the formula ‘`b==’a’||b==’b’||b==’c’`’.

In the quantified formulae, it is possible to refer to *quantified variables*, variables indexed by a quantifier or the preceding value of a quantifier:

```
expr:
        ’.’ name [ name ]
        |
        ’:’ name [ name ]
```

For instance, if there is a declaration ‘`typedef unsigned (1..3) index_t`’, the universal quantification

```
index_t e && (e == <index_t || :n < .n)
```

is equivalent to ‘`n[1]<n[2]&&n[2]<n[3]`’.

[4] For discussion on the order of values in types, see Section 1.3.1 [Types], page 5.

[5] One does not need a constrained Boolean type very often, but nevertheless it can be defined.

The conjunction and disjunction operators use *short-circuit evaluation*, meaning that the expression is evaluated in depth-first manner from left to right, and if the value of the left-hand-side expression of an operator alone can determine the value of the expression, no matter what the right-hand-side evaluates to, the right-hand-side will not be evaluated.

1.5.2.5 Selection

The ternary '?:' operator of C selects its second or third argument based on its first argument interpreted as a truth value. The '?:' operator in Maria is more generic. It is not a ternary operator but *n*-ary, with $n-1$ being the number of possible values for the type of its first argument.

Here is an example of the '?:' operator for a type having three possible values.

```
typedef int (1..3) i3_t;
place p i3_t: 1;
trans t
in { place p: p; }
out { place p: p ? p : |p : +p; };
```

The left-hand-side of the '?' on the output arc of transition 't' is the variable 'p', which is of type 'i3_t' and has three possible values. When the expression has its largest value (in this case '3'), the argument immediately following the '?' will be selected. The second largest value ('2') will select the third argument ('|p'), and the smallest value ('1') will select the last argument ('+p').

In this simple example, the marking of place 'p' will alternate between '1' and '2' in two states. It really should be emphasized that it is the number of values of the type that matters, not the number of possible (reachable) values for the left-hand-side expression. Selecting an initial marking '3' for place 'p' would yield only one state in the reachability graph.

1.5.2.6 Type Casting

The type casting operator, the prefix 'is', has two purposes. First, it can be used to set the *context type* in case it cannot be determined correctly. The parser is pretty good at guessing the context type, but there is one remarkable case when it cannot do so. Comparison expressions occur in Boolean context, but the arguments to the comparison operator typically are of some other type. If the argument is an enumeration constant or a construction expression for a compound value, the context must be set with the 'is' operator:

```
typedef enum { a, b, c } e3_t;
place p e3_t: a;
trans t
in { place p: p; }
out { place p: b; }
gate p != is e3_t a;
```

Boolean, integer and character literals will always be detected as such, no matter what the context type is.

The dynamic behavior of type casting is to convert a value of one type to an equivalent value of another type. Type conversion is only allowed if the two types have at least one common value. The only error that can occur during the conversion is a constraint violation.

Also compound values can be converted. Two compound types are considered compatible for the conversion if all their components are compatible. A union-typed value can be converted to the type of one of its components, or vice versa.

1.5.2.7 Atomicity

The parser may perform optimizations on expressions by converting them to equivalent forms. Currently transition gate expressions containing a conjunction in the top level will be split to several gate expressions, and it is likely that the model checker will do something similar to temporal logic formulae.

The transformations of expressions have one disadvantage: they break short-circuit evaluation. If you rely on short-circuit evaluation when writing an expression, it is a good practice to enclose the expression with ‘`atom()`’, e.g. ‘`atom (y==0||x/y>z)`’. Short-circuit evaluation takes place with the operators ‘`||`’, ‘`&&`’, ‘`=>`’ and ‘`?:`’.

1.5.3 Structures

There are two basic operators for structures: for constructing a value of a structure type and for accessing a structure component. Both inherit their syntax and semantics from C.

```
expr:
        ’{’ [ [ name ’:’ ] expr ( ’,’ [ name ’:’ ] expr )* ] ’}’
        |
        expr ’.’ name
```

In the constructor expression, the expressions for individual components may be preceded by the name of the corresponding struct component.

Since Petri nets have no assignment statement, a special operation is needed for modifying a structure component.

```
expr:
        expr ’.’ ’{’ name expr ’}’
```

For instance, ‘`a.{b c}`’ has otherwise the same value as the structure ‘`a`’, but with the component ‘`b`’ equal to ‘`c`’.

1.5.4 Unions

The union type in Maria is tagged: it is always known which component of the union has been assigned to (or is the *active component*), and this information can also be enquired by using the infix ‘`is`’ operator.

```
expr:
        name '=' expr
        |
        expr IS name
        |
        expr '.' name
```

The '.' operator can be used to access the value of the active component. If the specified component is not active, a union violation will occur.

Structures and unions are very practical for modeling inherited classes of object-oriented languages. The example file 'object.pn' in the Maria distribution shows how a simple class structure can be translated to Maria types and how objects can be converted between a base class and derived classes.

1.5.5 Arrays

Array values are constructed and indexed with a syntax familiar from the C programming language.

```
expr:
        '{' [ expr ] ( ',' [ expr ] )* '}'
        |
        expr '[' expr ']'
```

In addition, the contents of an array can be shifted by a given number of items. The amount to be shifted is always reduced to the modulo of number of items in the array. For instance, shifting an array indexed by Boolean values by an even number of items has no effect.

```
expr:
        expr '<<' expr
        |
        expr '>>' expr
```

Since Petri nets have no assignment statement, a special operation is needed for modifying an array item.

```
expr:
        expr '.' '{' '[' expr ']' expr '}'
```

For instance, 'a.{[b] c}' has otherwise the same value as the array 'a', but with the item at 'b' equal to 'c'.

1.5.6 Buffers

For ideal buffers, two operations ought to be enough: removing an item for reading, and writing an item. Since there is no well-defined semantics for Petri Nets whose arc expressions have side-effects, reading a buffer has to be split in two separate operations: peeking at an item ('*') and removing ('-').

Some applications need to access buffers out of order. It is possible to specify the buffer position using an index.

```
expr:
        expr index '+' expr
        |
        '-' expr index
        |
        expr index '-' expr
        |
        '*' expr index
index:
        [ '[' expr ']' ]
```

The unary '-' operator removes one item from a buffer. With the binary '-' operator, it is possible to remove several successive items. The second operand indicates the number of items to be removed.

Many applications need to know the remaining or used buffer capacity. That is what the unary operators '%' and '/' are for. It is also possible to construct the whole buffer contents with one expression, just like struct and array values.

```
expr:
        '{' [ expr ] ( ',' [ expr ] )* '}'
        |
        '/' expr
        |
        '%' expr
```

1.6 Operations on Multi-Sets

The state of a Petri Net is identified by the distribution of tokens in the places. The contents of the places, also called *markings*, can be represented as multi-sets, which are sets that can contain several instances of an item, meaning that the *multiplicity* of an item in the set may be greater than one.

Arc expressions of transitions and initialization expressions of the model typically make use of two multi-set operations. The simpler one, specifying a multiplicity or a token multiplier, makes use of the binary '#' operator.

```
marking:
        '(' marking_list ')'
        |
        expr '#' marking
```

For instance, '347#33101' stands for 347 tokens having the value 33101, and '4#(3#2,1)' is another way for expressing '12#2,4#1', which means 12 tokens of the value 2 and four tokens of the value 1.

The other operation, computing a multi-set sum, is called *quantification*.

```
marking:
        typereference name [ '(' expr ')' ] ':' marking
```

This is equivalent to the formula

$$\sum_{n \in T \land e} m$$

where n corresponds to '`name`', T to '`typereference`', e to '`expr`' and m to '`marking`'. Usually both e and m make use of n.

The construct will iterate through all the values of the type, binding each value to a named variable. If the condition expression evaluates to true when the variable is bound to a value, the marking expression will be evaluated using that binding, and the resulting items will be added to the quantification result.

In transition expressions, it is possible to define *quantified variables*, variables indexed by a quantifier:

```
expr:
        '.' name [ name ]
        |
        ':' name [ name ]
```

For instance, writing '`bool b: .a`' is equivalent to writing '`"a[false]", "a[true]"`'. The optional second name refers to a quantifier variable. It is needed in nested quantifications when indexing a variable by something else than the innermost quantifier.

The variant with the colon indexes variables based on the predecessor of the quantifier variable. It is more useful in universal and existential quantification (see Section 1.5.2.4 [Logic], page 20).

There are two multi-set valued literals: the empty set, which is useful in subset and equality comparisons, and the marking of a place.

In a formula, the marking of a place is a function of the current state. In the output arc of a transition, it is defined as the multi-set of tokens that will be removed from the place by the input arcs of the transition instance. Referring to the contents of a place in an input arc or in a gate expression yields a dynamic error.

```
marking:
        EMPTY
        |
        PLACE name
```

Multi-sets can be compared in two ways. Two sets are equal if both contain the same amount of the same items. Multi-set A is subset of multi-set B if for each item in A there are at least as many such items in B.

```
formula:
        marking SUBSET marking
        |
        marking EQUALS marking
```

Multi-sets can be combined in three ways. The union of two multi-sets is computed by adding the multiplicities of all items in each set together. The intersection is formed by computing the maximum multiplicity of each item in both sets. Finally, multi-sets can be subtracted from each other. In $A - B$ there are only items that have a greater multiplicity in A than in B. The multiplicity of each item is the difference of the multiplicities in A and in B.

```
marking:
        marking UNION marking
        |
        marking INTERSECT marking
        |
        marking MINUS marking
```

As a special case of intersection, it is possible to filter multi-sets based on conditions. A named variable will iterate through all the items in a multi-set, and if the condition expression is true, the item will be included in the resulting set, with its original multiplicity. This construct can be viewed as an intersection with a multi-set whose items have either zero or infinite multiplicity.

```
marking:
        SUBSET name ’{’ marking_list ’}’ expr
```

For instance, ‘`subset t { 3#true, 2#false } !t`’ equals ‘`2#false`’.

There are two mapping operations similar to the filtering operation described above. One operation preserves the cardinality of a multi-set (the number of contained items). It transforms a multi-set of one type to a multi-set of another type by iterating through all the items in the source multi-set with a named variable, and by computing the items for the target multi-set by evaluating a basic expression.

A more generic mapping iterates through a multi-set, assigning the multiplicity and the value of each distinct item to a pair of variables, and mapping the items by evaluating a multi-set valued expression.

```
marking:
        MAP name ’{’ marking_list ’}’ expr
        |
        MAP name ’#’ name ’{’ marking_list ’}’ marking
```

For instance, ‘`map t { 3#true, 2#false } !t`’ equals ‘`3#false, 2#true`’, while the expression ‘`map n#t { 3#true, 2#false } (t?1#!t:n#t)`’ evaluates to ‘`3#false`’.

Last but not least, there are operations for determining the minimum or maximum multiplicity of the items belonging to a multi-set, or for computing the cardinality, the sum of the multiplicities. The minimum multiplicity of an empty set is `UINT_MAX`.

```
formula:
        MAX marking
        |
        MIN marking
        |
        CARDINALITY marking
```

1.7 Temporal Logic

The grammar for temporal logic is *LTL*, Linear Temporal Logic. The operators “eventually”, “henceforth”, and “in the next state” have the digraph representations ‘`<>`’, ‘`[]`’ and ‘`()`’, respectively.

```
          '<>' formula
          |
          '[]' formula
          |
          '()' formula
          |
          formula UNTIL formula
          |
          formula RELEASE formula
```

1.8 Non-Determinism in Transitions

Sometimes it is necessary to have non-deterministic transitions, that is, transitions that have several possible outputs for one input. Often such behavior is modeled by adding a '`const`' place to the net and binding a token from the place to the non-deterministic transition through a bidirectional arc. This approach may disturb partial order reduction algorithms.

The Maria language contains a construct for declaring non-deterministic transition variables, called *output variables*.[6]

```
expr:
        typereference [ name ] '!' [ '(' expr ')' ]
```

The expression will evaluate to the value of the output variable, which will iterate through all the values of the type, or to all values for which the condition expression is true. The output variable can be given a name, and doing so is recommended if the output variable is used in several expressions or if a condition expression is used.

```
place p bool: true;
trans t in { place p: p } out { place p: bool! };
```

The default names for unnamed output variables are '`:0`', '`:1`', and so on, a colon followed by a hexadecimal number. Those who want to write obfuscated net descriptions can refer to these names using the double-quoted notation.

1.9 Scoping of Identifiers

Even though Petri Nets are a very flat formalism, there is pretty much depth in the scoping of names. In the Petri Net level, there are three name spaces that cannot be overridden: one for place names, one for transition names and one for data type names. Function names can be overridden in the transition level.

In expressions (see Section 1.5.1 [Literals], page 17), names can stand for several things. The names will be looked up in the following order:

1. function parameters (in a function definition),
2. iterator variables (multi-set operations, non-determinism)

[6] With output variables, we only mean these non-deterministic variables, not all the variables that may occur in the output arc expressions of a transition.

3. previously declared transition variables
4. nullary functions
5. enumeration constants (in the type context)
6. state propositions (in ‘`deadlock`’ and ‘`reject`’ formulae)

If all the look-ups fail for a name, a transition variable will be declared, if a transition definition is being parsed and if the type context is known.

The parser can issue warning messages about names that exist in several name spaces. If you know what you are doing, these warnings can safely be ignored.

2 Reachability Analysis with Maria

Determining all interesting behaviors of a concurrent system is done by computing a graph of all system states reachable from the initial state. This process is called *reachability analysis*. Once a (possibly incomplete) reachability graph has been generated, it can be examined, and one can verify temporal properties from it.

It is also possible to verify temporal properties during the generation of the reachability graph (see Section 3.2 [Model Checking], page 54). This can guide the search and speed up the analysis, but verifying another temporal property from the system will usually require the reachability analysis to be performed again.

2.1 Invoking Maria

The usual way to invoke Maria is as follows:

```
maria -b model
```

Here *model* is the Petri Net file name, which usually ends in '`.pn`'. The base name for reachability graph files (see Appendix B [Graph Files], page 66) is generated by removing the directory name part and the file name suffix (the last part of the name starting with a period '`.`') from the Petri Net file name. Three files will be generated using the base name, with the suffixes '`.rgd`', '`.rgs`', '`.rga`' and '`.rgh`'. Thus, '`maria -b dining.pn`' yields '`dining.rgd`', '`dining.rgs`', '`dining.rga`' and '`dining.rgh`', and '`maria -d test/order.pn`' yields '`order.rgd`', '`order.rgs`', '`order.rga`' and '`order.rgh`', in the current directory.

2.1.1 Interrupting the Reachability Analysis

Maria traps the interrupt signal (`SIGINT`), which is usually bound to *C-c*. When it catches the signal, it will stop processing new states. After Maria has finished with the state it is currently processing, analyzing all the enabled transition instances and generating the successor states, it will update the reachability graph directory and stop the analysis.

Analyzing and firing enabled transition instances in a state may take a long time. Be patient or issue a `SIGQUIT` (normally bound to *C-*) or a `SIGKILL` signal to abort the process immediately, leaving the reachability graph files in an inconsistent state.

2.1.2 Options

Maria supports both traditional single-letter options and mnemonic long option names (if compiled with '`getopt_long`', see Appendix C [Compiling], page 67). Long option names are indicated with '`--`' instead of '`-`'. Abbreviations for option names are allowed as long as they are unique. When a long option takes an argument, like '`--include`', connect the option name and the argument with '`=`' or provide the argument in the immediately following command line argument.

The options are interpreted in the order they are entered on the command line. In other words, the options for interpreting a model should be specified before loading the model. For instance, '`maria -b dbm.pn -DUNUSED`' does not make much sense, while '`maria -DUNUSED -b dbm.pn`' does.

Here is a list of options that can be used with Maria, alphabetized by short option. It is followed by a cross key alphabetized by long option.

'`-a` *`limit`*'
'`--array-limit=`*`limit`*'
: Limit the size of array index types to *limit* possible values. A limit of 0 disables the checks.

'`-b` *`model`*'
'`--breadth-first-search=`*`model`*'
: Generate the reachability graph of *model* using breadth-first search. Equivalent to '`-m` *`model`* `-e breadth`'.

'`-C` *`directory`*'
'`--compile=`*`directory`*'
: Generate C code in *directory* for evaluating expressions and for the low-level routines of the transition instance analysis algorithm (see Section 3.1.5 [Instance Analysis], page 54). In order for this option to work, the program must have been compiled with support for compiled expressions enabled (see Appendix C [Compiling], page 67). Also, the compilation script '*`progname`*`-cso`' must be found in the search path, where '*`progname`*' is the name used to invoke the analyzer. It is a good idea to have a look at the compilation script, and to adjust the variable definition `INCLUDES`, so that the required header files can always be found.

You may also want to adjust the variables `DEFINES`, `CC`, `CFLAGS`, `LD` and `LDFLAGS` either in the script or in the environment of the reachability analyzer. For instance, if you are using Bourne shell or one of its descendants, prefixing the command line for invoking Maria with '`DEFINES="" CFLAGS=""`' disables all compiler optimizations for the model being analyzed. This may be useful if the optimization algorithms of the C compiler would consume too many resources.

Applying this option should not affect the behaviour of the model (i.e. it is a bug if it does). When this option is used, evaluation errors are reported in a slightly different way. The interpreter displays the valuation and expression that caused the first error in a state; the compiled code displays the number of errors. For performance reasons, the generated code does not check for overflow errors when adding items to multi-sets.

'`-c`'
'`--no-compile`'
: The opposite of '`-C`'. Evaluate all expressions in the built-in interpreter. This is the default behavior.

'`-D` *`symbol`*'
'`--define=`*`symbol`*'
: Define the preprocessor symbol *symbol*. See Section 1.2.4.2 [Conditions], page 5.

‘`-d` *`model`*’
‘`--depth-first-search=`*`model`*’
Generate the reachability graph of *model* using depth-first search. Equivalent to ‘`-m` *`model`* `-e depth`’.

‘`-E` *`interval`*’
‘`--edges=`*`interval`*’
When generating the reachability graph, report the size of the graph after every *interval* generated edges.

‘`-e` *`string`*’
‘`--execute=`*`string`*’
Execute *string*. See Section 2.2 [Maria Shell], page 36.

‘`-g` *`graphfile`*’
‘`--graph=`*`graphfile`*’
Load a previously generated reachability graph from ‘*`graphfile`*`.rgh`’.

‘`-H` *`h`*`[,`*`f`*`[,`*`t`*`]]`’
‘`--hashes=`*`h`*`[,`*`f`*`[,`*`t`*`]]`’
Configure the parameters for lossy verification (‘`-P`’ or ‘`-M`’). For ‘`-P`’, allocate *t* universal hash functions of *f* elements and corresponding hash tables of *h* bits each. For ‘`-M`’, allocate a *f* bytes for all *h* elements of the hash table. The value *h* (and for ‘`-P`’, also the value *f*) will be rounded up to suitable values.

‘`-?`’
‘`-h`’
‘`--help`’ Print a summary of the command-line options to Maria and exit.

‘`-I` *`directory`*’
‘`--include=`*`directory`*’
Append *directory* to the list of directories searched for include files.

‘`-i` *`columns`*’
‘`--width=`*`columns`*’
Set the right margin of the output to *columns*. The default is 80.

‘`-j` *`processes`*’
‘`--jobs=`*`processes`*’
When checking safety properties (options ‘`-L`’, ‘`-M`’ and ‘`-P`’), use this many worker processes to speed up the analysis on a multiprocessor computer. See also ‘`-k`’ and ‘`-Z`’. Note that on Digital UNIX, this option may not work properly if the model is generating successor states too fast (especially when using the ‘`-C`’ option).

‘`-k` *`port`*`[/`*`host`*`]`’
‘`--connect=`*`port`*`[/`*`host`*`]`’
Distribute safety model checking (options ‘`-L`’, ‘`-M`’ and ‘`-P`’) in a TCP/IP network. For the server, only *port* is specified as a 16-bit unsigned integer, usually between 1024 and 65535. For the worker processes, *port*/*host* specifies the port and the address of the server. See also ‘`-j`’.

All processes should use the same command line options, except that on the server, the ‘-k’ switch takes only the *port* argument, and the worker processes exit after executing the ‘breadth’ or ‘depth’ command. To gain best performance, start the server and one client on the fastest computer of the network.

Note that when a computer arranges machine words in the big endian byte order, all computers must be big endian and have the same word length. Little endian computers interoperate regardless of their word lengths.

‘-L *model*’
‘--lossless=*model*’

Load *model* and prepare for analysing it by constructing a set of reachable states in disk files. See also ‘-M’, ‘-P’, ‘-j’ and ‘-k’.

‘-m *model*’
‘--model=*model*’

Load *model* and clear its reachability graph. See also ‘-b’ and ‘-d’.

‘-M *model*’
‘--md5-compacted=*model*’

Load *model* and prepare for analysing it by constructing an overapproximation of the set of reachable states in the main memory by using a technique called *hash compaction*. See also ‘-P’, ‘-L’, ‘-j’ and ‘-k’.

‘-N *cregexp*’
‘--name=*cregexp*’

Specify the names allowed in context *c* as the extended regular expression *regexp*; See section “Regular Expressions” in *GNU GREP Manual*. In order for this option to work, the program must have been compiled with the POSIX regular expression library enabled (see Appendix C [Compiling], page 67). The context is identified by the first character of the parameter string; the succeeding characters constitute the regular expression that allowed names must match:

‘P’	place name
‘T’	transition name
‘t’	type name
‘e’	name of enumeration constant
‘c’	name of struct or union component
‘f’	function name
‘p’	function parameter name
‘v’	transition variable name
‘i’	iterator variable name

‘-n *cregexp*’
‘--no-name=*cregexp*’

Specify the names not allowed in context *c* as the extended regular expression *regexp*.

If both '-N' and and '-n' are specified for a context *c*, then the allowing match takes precedence. For instance, to require that all user defined type names be terminated with '_t', specify '-nt -Nt'_t$''. The quotes in the latter parameter are required to remove the special meaning from '$' in the command line shell you are probably using to invoke Maria.

'-P *model*'
'--probabilistic=*model*'

Load *model* and prepare for analysing it by constructing an overapproximation of the set of reachable states in the main memory by using a technique called *bitstate hashing*. See also '-M', '-L', '-j' and '-k'.

'-p *command*'
'--property-translator=*command*'

Specify the command to use for translating property automata. The command should read a formula from the standard input and write a corresponding automaton description to the standard output. The translator 'lbt' (available separately) is compatible with this option.

'-q *limit*'
'--quantification-limit=*limit*'

Prevent quantification (multi-set sum) of types having more than *limit* possible values. A limit of 0 disables the checks.

'-R'
'--modular'

Explore the state space in a modular way. See Section 1.3.5 [Subnets], page 10.

'-r'
'--no-modular'

Consider only one state space. This is the default.

'-t *limit*'
'--tolerance=*limit*'

Abort the analysis when *limit* non-fatal errors have been reported. A limit of 0, the default, allows an infinite number of errors to occur.

'-U *symbol*'
'--undefine=*symbol*'

Undefine the preprocessor symbol *symbol*. See Section 1.2.4.2 [Conditions], page 5.

'-u [M][*f*[*outfile*]]'
'--unfold=[M][*f*[*outfile*]]'

Unfold the net (optionally reducing it by constructing a *coverable marking* ('M')) and write it in format *f* to *outfile*. If *outfile* is not specified, dump the unfolded net to the standard output. Possible formats are 'm' (Maria (human-readable), default), 'l' (LoLA), 'p' (PEP), and 'r' (PROD).

When the PROD output is chosen, *outfile* must be specified and end in '.net'. Since PROD has no low-level input format, the transitions in the low-level net are grouped to equivalence classes based on the input and output arc weights.

Each equivalence class is folded to a high-level transition. The translation makes use of tables that are written to a separate file. If *outfile* is '*out*.net', the tables are written to '*out*.src/tables.c'.

'-V'
'--version'
Print the version number of Maria and exit.

'-v'
'--verbose'
Display verbose information on different stages of the analysis.

'-W'
'--warnings'
Enable warnings about suspicious net constructs. This is the default behavior.

'-w'
'--no-warnings'
The opposite of '-W'. Disable all warnings.

'-x *numberbase*'
'--radix=*numberbase*'
Specify the number base for diagnostic output. Allowed values for *numberbase* are 'oct', 'octal', '8', 'hex', 'hexadecimal', '16', 'dec', 'decimal' and '10'. The default is to use decimal numbers.

'-Y'
'--compress-hidden'
Reduce the set of reachable states by not storing the successor states of transitions instances for which a 'hide' condition holds. The hidden successors are stored to a separate state set. This option may save memory ('-L' or '-m') or reduce the probability that states are omitted ('-M' or '-P'), and it may improve the efficiency of parallel analysis ('-j' or '-k'), but it may also considerably increase the processor time requirement. The option also works with liveness model checking, but there is no guarantee that the truth values of liveness properties remain unchanged. This option can be combined with '-Z'.

'-y'
'--no-compress-hidden'
The opposite of '-Y'. This is the default behavior.

'-Z'
'--compress-paths'
Reduce the set of reachable states by not storing intermediate states that have at most one successor. This option may save memory ('-L' or '-m') or reduce the probability that states are omitted ('-M' or '-P'), and it may improve the efficiency of parallel analysis ('-j' or '-k'), but it may also considerably increase the processor time requirement. The option also works with liveness model checking, but there is no guarantee that the truth values of liveness properties remain unchanged. This option can be combined with '-Y'.

‘-z’
‘--no-compress-paths’

The opposite of ‘-Z’. This is the default behavior.

2.1.3 Option Cross Key

Here is a list of options, alphabetized by long option, to help you find the corresponding short option.

--array-limit=*limit*	-a
--breadth-first-search=*model*	-b
--compile=*directory*	-C
--compress-hidden	-Y
--compress-paths	-Z
--connect=*port* [/*host*]	-k
--define=*symbol*	-D
--depth-first-search=*model*	-d
--edges=*interval*	-E
--execute=*string*	-e
--graph=*graphfile*	-g
--hashes=*h* [,*f* [,*t*]]	-H
--help	-h
--include=*directory*	-I
--jobs=*processes*	-j
--lossless=*model*	-L
--md5-compacted=*model*	-M
--model=*model*	-m
--modular	-R
--name=*cregexp*	-N
--no-compile	-c
--no-compress-hidden	-y
--no-compress-paths	-z
--no-modular	-r
--no-name=*cregexp*	-n
--no-warnings	-w
--probabilistic=*model*	-P
--property-translator=*command*	-p
--quantification-limit=*limit*	-q
--radix=*numberbase*	-x
--tolerance=*limit*	-t
--undefine=*symbol*	-U
--unfold=[M][*f* [*outfile*]]	-u
--verbose	-v
--version	-V
--warnings	-W
--width=*columns*	-i

2.2 The Maria Shell

The Maria shell can be used to interactively examine the reachability graph of a model or to check temporal properties in it.

2.2.1 The Line Editor

The Maria shell has a command-line driven user interface. When support for the GNU Readline library has been enabled at compile time (see Appendix C [Compiling], page 67), using the command line is pretty comfortable. Without Readline, you have to cope with the system's default line editor, which certainly lacks command line history and completion features.

When invoked, Maria will greet you with the prompt ‘`@0$`’, where ‘`0`’ is the number of the current state, or ‘`$`’ if no model has been loaded. The initial state always has the number zero. Command lines may be split over multiple physical lines. When there is an unbalanced single or double quote or a multi-line (C-style) comment, or when the line ends in a backslash ‘`\`’, Maria will present the continuation prompt ‘`@`*n*`>`’. In case of a typing mistake, *`EOF`* will tell Maria to abort reading continuation lines.

2.2.1.1 Name Completion

This is by no means a complete list of Readline features. See section “Command Line Editing” in *GNU Readline Library*, if you are not familiar with Readline.

One of the nicest features of the Readline library is context-sensitive completion of names. The Maria shell completes names of data types (immediately following the ‘`is`’ keyword), places (following ‘`place`’), transitions (‘`trans`’), and keywords. Let us now assume that you have generated the reachability graph for the dining philosopher example (see Section D.1 [Dining], page 70). The following example illustrates how the context-sensitive completion of names works.

```
$graph diningRET
@0$paTAB
@0$path @29 plTAB
@0$path @29 place "fTAB
@0$path @29 place "fork" eqTAB
@0$path @29 place "fork" equals emTAB
@0$path @29 place "fork" equals empty RET
shortest path from condition to @29 (2 nodes):
@74:state (
 state:
  {3,thinking},{1,hungry},{4,hungry},{5,hungry},{2,eating}
)
4 predecessors
1 successor
```

```
transition finish->@29
<
 p:2
>
@29:state (
 fork:
  2,3
 state:
  {2,thinking},{3,thinking},{1,hungry},{4,hungry},{5,hungry}
)
4 predecessors
3 successors
@0$exit RET
```

Unfortunately backslash-quoted names cannot be completed. This is due to a limitation of the Readline library, which will only call the dequoting function if its built-in filename completion function is being used. This is why the keyword completer adds a double quotation mark after keywords that are likely to be followed by a name.

2.2.2 The Query Language

Maria uses a script language in which statements are separated by semicolons (‘;’). See Section 2.2.3.1 [Separating Statements], page 48, for other considerations.

There are commands for moving around in the graph and for evaluating formulae in different graph nodes. Maria shell commands are reserved words only in the beginning of a statement: there is no need to quote a place name ‘exit’, for instance.

2.2.2.1 Loading a Model

```
statement:
        GRAPH name
        |
        MODEL name
```

The ‘graph’ command loads a previously generated reachability graph. The supplied name must exclude the ‘.rgh’ suffix.

The ‘model’ command loads a Petri net model and initializes its reachability graph. Be careful with this command; it will delete a previously generated reachability graph of the model, if one exists in the current directory.

2.2.2.2 Displaying a Model

```
statement:
        [ VISUAL ] DUMP
```

The ‘dump’ command displays the syntax tree of the current model in the modelling language. When the ‘visual’ prefix is present, the Petri net will be displayed graphically (see Section 2.2.4 [Visual], page 48). This command may be useful when trying to find out how Maria expands multi-set summations (see Section 1.6 [Multi-Sets], page 24) or quantifications (see Section 1.5.2.4 [Logic], page 20).

2.2.2.3 Unfolding a Model

```
statement:
        UNFOLD [ name ]
```

The '`unfold`' command unfolds the currently loaded Petri net model. The argument takes the same format as the command line option '`-u`' (see Section 2.1.2 [Maria Options], page 29).

2.2.2.4 Exporting a Labelled State Transition System

```
statement:
        LSTS [ name ]
```

Maria is able to export the reachability graph or parts thereof as a labelled state transition system. The '`lsts`' command specifies a file name for an LSTS. When the command is invoked without a file name, any currently open LSTS files are closed and the function cancelled.

The '`lsts`' command affects exhaustive analysis (see Section 2.2.2.6 [Depth and Breadth], page 39), non-visual path queries (see Section 2.2.2.17 [Path], page 45) and the representation of counterexample paths of liveness properties (see Section 3.2.2 [Liveness], page 54).

In exhaustive analysis, the command should be invoked before any states have been explored, right after the model has been loaded. When using the path commands, the '`lsts`' command should be issued right before generating the counterexample. These measures are necessary, because for efficiency reasons, the '`lsts`' command has been implemented so that all generated states and arcs are exported to LSTS files if ones have been opened. The path query commands export results to LSTS if the files are open, and close the files immediately after that. It is impossible to combine several query results into one exported LSTS.

The LSTS output can be controlled by hiding variables or transition instances (see Section 1.3.4 [Transitions], page 8) and by specifying state propositions (see Section 1.3.6.3 [Propositions], page 12).

2.2.2.5 Exporting the Reachability Graph

```
statement:
        [ VISUAL ] DUMPGRAPH
```

The command '`dumpgraph`' exports the portion of the reachability graph that has been generated so far. When the '`visual`' prefix is present, the reachability graph will be displayed graphically (see Section 2.2.4 [Visual], page 48). See Section 2.2.2.6 [Depth and Breadth], page 39, for information on generating the reachability graph.

2.2.2.6 Exhaustive Analysis

```
statement:
        BREADTH [ STATE ]
        |
        DEPTH [ STATE ]
```

The commands 'breadth' and 'depth' generate all states that are reachable either from the current state or from a specified state. This exhaustive search will only be performed if the successors of the state have not already been generated.

```
$model "dining.pn"
@0$breadth
"dining.pn": 82 states (3..6 bytes), 265 arcs
```

At the end of the search, the numbers of generated states and events are reported. The byte range indicates the minimum and maximum lengths of encoded state representations. This number is affected by modelling techniques, such as the use of capacity constraints and invariants (see Section 1.3.3 [Places], page 7), and by not folding places unnecessarily. For this particular model (see Section D.1 [Dining], page 70), the maximum length of encoded states can be reduced from 6 to 4 bytes by splitting the place that holds pairs of philosophers and their states (thinking, hungry, eating) to three places, holding a set of those philosophers that are in the state, and by defining an invariant initialisation expression for the "thinking" place.

The 'breadth' and 'depth' commands can be also used in purely state-based safety verification (command-line options '-L', '-M' and '-P'), without generating a reachability graph. If either command is given a safety LTL formula as an argument, the model contained in the file will be verified against the property. When the command is preceded by a 'visual' keyword, any path that violates the formula will be displayed graphically (see Section 2.2.4 [Visual], page 48).

```
statement:
        [ VISUAL ] BREADTH formula
        |
        [ VISUAL ] DEPTH formula
```

2.2.2.7 Evaluating Expressions and Formulae

```
statement:
        [ VISUAL ] [ EVAL ] [ STATE ] formula
```

Expressions and formulae can be evaluated either in the current state or in a given state.

```
@0$place fork
fork:
 1,2,3,4,5
@0$eval @4 place fork
fork:
 1,2,3,5
```

When the 'visual' keyword is present and a temporal formula is being evaluated, a path violating the formula is displayed graphically (see Section 2.2.4 [Visual], page 48).

2.2.2.8 Displaying Markings

With the '`show`' command it is possible to view the marking of a node in the reachability graph. When a node is not specified, the current node will be shown.

The command can also be used to view the nodes in a strongly connected component (see Section 2.2.2.16 [Strong], page 44). It is possible to specify a Boolean condition to select a subset of the states in the strongly connected component to be displayed.

```
statement:
        [ VISUAL ] SHOW [ STATE ]
        |
        [ VISUAL ] [ SHOW ] COMP [ expr ]
        |
        [ VISUAL ] SHOW STATE STATE ( STATE )*
```

When the keyword '`show`' is followed by a sequence of at least two state numbers, Maria displays the sequence of states in the same way as the '`path`' command does (see Section 2.2.2.17 [Path], page 45). This command is most useful with the '`visual`' prefix, since it can be used for visualizing a previously reported path.

```
@0$show @4
@4:state (
 fork:
  1,2,3,5
 state:
  {1,thinking},{2,thinking},{3,thinking},{5,thinking},{4,hungry}
)
4 predecessors
5 successors
```

2.2.2.9 Excluding Places from Displayed Markings

Petri net models often contain a large number of places whose contents are of less significance when investigating a particular property of the model. The '`hide`' command controls which places are displayed by the '`show`' command (see Section 2.2.2.8 [Show], page 40) and in the graphical interface (see Section 2.2.4 [Visual], page 48).

```
statement:
        HIDE [ '!' ] [ [ PLACE ] name ( ',' [ PLACE ] name )* ]
```

When followed by an exclamation point, the command selects places to be shown (instead of to be hidden). An empty list of place names selects all places. Typically, one can issue the command '`hide`' followed by '`hide !` *`placename`*' for each place to be shown.

2.2.2.10 Selecting the Active Subnet

When the model contains subnets (see Section 2.2.2.10 [Subnet], page 40) and Maria has been told to generate a reachability graph and apply modular analysis (see Section 2.1 [Invoking Maria], page 29), the '`subnet`' command can be used for navigating in the state spaces of the modules.

The ‘subnet’ command can be used for navigating in the hierarchy tree in a similar way as the ‘cd’ command navigates in the directory tree. It takes a sequence of subnet identifiers separated by slashes. The string ‘..’ denotes the parent net.

When invoked without parameters, ‘subnet’ selects the root net, which represents the whole system. If the parameter string starts with a slash, the tree is selected relative to the root net. Otherwise, it is selected relative to the currently selected net.

Nets that have been given a name (see Section 2.2.2.10 [Subnet], page 40) can be referred to by that name. All nets can be referred to by their index number in the parent net. The subnets are numbered starting from zero.

```
statement:
        SUBNET [ [ netid ] ( '/'+ netid )* '/'* ]
netid:  '..' | name | number
```

2.2.2.11 Listing Successor Nodes

The ‘succ’ command lists all successors of a state, or possible events in the state. If no successors have been generated for the state, they will be generated on demand. This makes Maria suitable for simulating or debugging a Petri Net model without generating the complete reachability graph: to create the interesting part of the reachability graph, one can keep listing the successors of the states he is interested in.

The command can also be used to view the successors of a strongly connected component (see Section 2.2.2.16 [Strong], page 44).

```
statement:
        [ VISUAL ] SUCC [ '!' ] [ STATE ]
        |
        [ VISUAL ] SUCC COMP
```

When the ‘visual’ keyword is present, the successor states or components are displayed graphically (see Section 2.2.4 [Visual], page 48).

When the ‘succ’ keyword is followed by an exclamation point (‘!’), Maria will follow the chain of successor states until a state with several successors is encountered.

```
@0$succ
transition left->@1
<
 p:1
>
transition left->@2
<
 p:2
>
transition left->@3
<
 p:3
>
```

```
transition left->@4
<
 p:4
>
transition left->@5
<
 p:5
>
```

2.2.2.12 Listing Predecessor Nodes

The ‘pred’ command lists all the generated predecessors of a state, or all events leading to the state.

The command can also be used to view the predecessors of a strongly connected component (see Section 2.2.2.16 [Strong], page 44).

```
statement:
        [ VISUAL ] PRED [ '!' ] [ STATE ]
        |
        [ VISUAL ] PRED COMP
```

When the ‘visual’ keyword is present, the predecessor states or components are displayed graphically (see Section 2.2.4 [Visual], page 48).

When the ‘pred’ keyword is followed by an exclamation point (‘!’), Maria will follow the chain of predecessor states until a state with several predecessors is encountered.

```
@0$pred
transition finish<-@20
<
 p:5
>
transition finish<-@18
<
 p:4
>
transition finish<-@15
<
 p:3
>
transition finish<-@11
<
 p:2
>
transition finish<-@6
<
 p:1
>
```

2.2.2.13 Moving in the Graph

Moving from a state to another is as simple as entering the state number. To make scripts more readable, you may also use the ‘go’ keyword.

```
statement:
        [ GO ] STATE

@0$@4
@4$go @0
@0$
```

2.2.2.14 Anonymous Transitions

Sometimes, when modelling a complex system, it may be useful to specify a different initial state as a starting point for the analysis. For instance, one might want to slightly modify the marking of an erroneous state to see how the analysis would proceed from there.

In Maria, new states can be computed by entering an anonymous transition that will be evaluated in a specified state, or in the current state if no state is specified. No arcs will be added to the reachability graph. The generated states will be displayed either textually or graphically.

```
statement:
        [ VISUAL ] [ STATE ] TRANS atrans*
atrans:
        ’{’ [ avar_expr ( delim avar_expr )* [ delim ] ] ’}’
        |
        IN trans_places
        |
        OUT trans_places
        |
        GATE expr ( ’,’ expr )*
avar_expr:
        typereference name
        |
        typereference name ’!’ [ ’(’ expr ’)’ ]

@0$trans in { place fork: 1 } out { place fork: 2 }
@82:unprocessed state (
 fork:
  2#2,3,4,5
 state:
  {1,thinking},{2,thinking},{3,thinking},{4,thinking},{5,thinking}
)
@0$
```

2.2.2.15 Defining Functions

All the functions defined in the Petri Net are available in the query tool. It is also possible to define additional functions by using the '`function`' keyword.

```
statement:
        FUNCTION function
```

```
@0$function bool assert (bool f) f || fatal
@0$assert (false);
<fatalExpression:
 fatal
>
```

If there are many function definitions, it is advisable to write them in a Maria command file, e.g. '`dining`':

```
#!/usr/local/bin/maria
graph dining;
function bool assert (bool f) f || fatal;
```

The definitions can be loaded in at least three different ways:

```
$maria dining
@0$exit
$./dining #the script must be executable
@0$exit
$maria
$#include "dining"
```

2.2.2.16 Strongly Connected Components

A *strongly connected component* of a directed graph (e.g. a reachability graph) is a set of nodes that are reachable from each other by following the arcs. If the reachability graph of a model has only one strongly connected component, it is guaranteed to be free of deadlocks, since the initial state is reachable from all states.

A strongly connected component, or a node of a *component graph*, may be *trivial*, meaning that it contains only one node of the underlying graph. A *terminal component* does not have any successors.

Maria computes the strongly connected components by starting from a specified state (by default, the current state) and considering a transitive closure of its generated successors in the reachability graph. It is possible to limit the transitive closure by specifying a condition. States for which the condition does not evaluate to '`true`' are ignored by the search algorithm.

Once the component graph has been generated, it is possible to list all its non-trivial terminal components, or all components. Also the '`show`', '`succ`' and '`prev`' commands can be used to examine the component graph. The '`visual`' keyword selects graphical display (see Section 2.2.4 [Visual], page 48).

```
statement:
        STRONG [ STATE ] [ expr ]
        |
        TERMINAL
        |
        [ VISUAL ] COMPONENTS
        |
        [ VISUAL ] [ SHOW ] COMP [ expr ]
        |
        [ VISUAL ] ( SUCC | PRED ) COMP
```

```
@0$strong
dining: 82 states (3..6 bytes), 265 arcs, 2 components, 1
 terminal component
@0$@@0
terminal trivial strongly connected component @@0: @71
@0$pred @@0
@@1
```

Maria deploys Tarjan's algorithm for computing strongly connected components. The algorithm was implemented in Prod by Vesa Hirvisalo and initially ported to Maria by Emil Falck. His port was rewritten by Marko Mäkelä to optimize memory usage. The implementation requires two bits for each node in the reachability graph, and one search stack whose length is limited by the length of the longest cycle or acyclic path in the generated reachability graph. Everything else is managed in disk files.

2.2.2.17 Shortest Paths

The '`path`' command lets one to find out the shortest path between a specific state and a set of states.

```
statement:
        [ VISUAL ] PATH ( STATE | COMP | expr ) [ STATE ] [ ',' expr ]
        |
        [ VISUAL ] PATH STATE ( COMP | expr ) [ ',' expr ]
```

The command takes up to three arguments: the target state, the source state, and an optional condition that all states on the path must fulfill. If the source state is omitted, the command finds out the shortest path to the given target state. Either the source or the target state must be specified; the other end of the path may be identified with a state formula or with a strongly connected component number.

When the '`visual`' keyword is present, the path is displayed graphically (see Section 2.2.4 [Visual], page 48).

```
@0$path @6
shortest path from @0 to @6 (3 nodes):
```

```
@0:state (
 fork:
  1,2,3,4,5
 state:
  {1,thinking},{2,thinking},{3,thinking},{4,thinking},{5,thinking}
)
5 predecessors
5 successors
transition left->@2
<
 p:2
>
@2:state (
 fork:
  1,3,4,5
 state:
  {1,thinking},{3,thinking},{4,thinking},{5,thinking},{2,hungry}
)
4 predecessors
5 successors
transition left->@6
<
 p:1
>
@6:state (
 fork:
  3,4,5
 state:
  {3,thinking},{4,thinking},{5,thinking},{1,hungry},{2,hungry}
)
4 predecessors
4 successors
```

A third variant of the ‘path’ command finds the shortest path from the current state to a loop. In order to avoid an ambiguity in the grammar, the loop must consist of at least three states.

```
statement:
        [ VISUAL ] PATH STATE STATE STATE ( STATE )* [ ',' expr ]
```

Maria does not check whether the given states really form a loop; it just finds the shortest path to any of the specified states so that the optional condition holds in all states along the path. When the path enters a state that is not the first (and last) state specified, the loop is shifted. The result is an acyclic path leading to a cycle, resembling the shape of the number 6.

2.2.2.18 Miscellanous Commands

The ‘help’ command displays a short list of all commands. The ‘stats’ command displays some statistical information of the graph being analyzed. The ‘time’ command displays timing statistics since the last invocation of the ‘time’ command, or since the time when the analyzer was started.

The ‘cd’ command can be used to switch the current directory. Without a parameter, it tries to switch to the directory the environment variable HOME expands to. The ‘translator’ command is used to specify the program for translating temporal logic formulae into property automata. When the command is invoked without arguments, the verification of temporal formulae is disabled.

In order for the ‘compiledir’ command to work, Maria must have been compiled with support for compiled expressions enabled (see Appendix C [Compiling], page 67). This command specifies the directory that will be used for generating executable code from the model. Invoking the command without arguments disables the code generator.

The ‘log’ command allows the textual output of query language commands to be redirected to a file, or to the standard error stream when no file name is specified to the command.

The ‘prompt’ command may be useful when Maria is executed as a subprocess. When the command is invoked with a non-null character constant argument, this character will be echoed on its own line before the command prompt is shown. The plain ‘prompt’ command disables this feature again.

```
        HELP
        |
        STATS
        |
        TIME
        |
        CD [ name ]
        |
        TRANSLATOR [ name ]
        |
        COMPILEDIR [ name ]
        |
        LOG [ name ]
|
        PROMPT [ ’c’ ]
```

```
@0$stats
dining: 82 states (3..4 bytes), 265 arcs
```

2.2.2.19 Exiting

The query tool can be exited either by issuing the command ‘exit’ or by typing the end-of-file character *EOF* at the command prompt.

2.2.3 Some Quirks with the Query Language

Due to the way the query language and the preprocessor (see Section 1.2.4 [Preprocessor], page 4) have been implemented, there are some things that are not obvious for the novice user.

2.2.3.1 Separating Statements

Maria uses a script language in which statements are separated by semicolons (‘`;`’). The separator is optional at the end of input. In the interactive mode, every logical line is parsed separately and semicolons are only required when issuing multiple commands on one command line. In script files, semicolons are mandatory between the statements. Consider the following example:

```
#!/usr/local/bin/maria
graph dining;
function bool eval () false;
eval;
eval
```

This example will evaluate the function twice. If the semicolon was omitted, the function would be evaluated only once, since ‘`eval`’ would be treated as a keyword in that case.

2.2.3.2 Conditional Processing in the Editor

The line editor of Maria performs simple lexical analysis to find out whether the line entered is complete. For instance, if the line contains an unbalanced amount of quotes, a continuation prompt will be presented and further lines will be read until all quotes and multi-line comments are balanced or the entry is aborted by typing *`EOF`* at the beginning of a continuation line.

However, preprocessor directives for conditional processing are not detected by the command line interpreter. In case you want to enter conditional statements (which must span several physical lines) interactively, you must enter the physical lines as one logical line by quoting all newline characters but the last one. This can be achieved on many systems by typing *`C-v C-j`*.

2.2.4 Visualizing Graphs and Paths

Many commands in the query language can be preceded by the ‘`visual`’ keyword. When this keyword is present, the results of the command will be displayed in a separate visualization process. The visual variant often displays more information, such as edge attributes (transition names and valuations) in displayed paths.

2.2.4.1 GraphViz, the Graph Visualizer

In order for this option to work, a software package for drawing graphs called Graphviz (`http://www.graphviz.org`) must be installed. The ‘`lefty`’ command, belonging to the package, must be in the search path, and the script ‘`dotty.lefty`’ must be located either in the default ‘`lefty`’ search path or in a directory specified in `LEFTYPATH`. Also, the visualization script ‘*progname*`-vis`’ must be found in the search path, where ‘*progname*’ is the name used to invoke the analyzer.

On the Microsoft Windows platform, Maria does not invoke GraphViz as a subprocess. Instead, it writes the visualization data to the file ‘`maria-vis.out`’ in the current directory. This file must be slightly edited before feeding it to the ‘`dotty`’ or ‘`dot`’ programs of GraphViz.

The visualization program reacts to some keyboard and mouse commands. Pressing the left or middle mouse button while the mouse pointer is located on a graph node requests for the successors or predecessors of the node to be generated. Holding down the right mouse button brings up a menu whose contents depends on whether the mouse pointer is located above a node, an edge or somewhere else in the graph. There exist keyboard short-cuts for most menu entries.

The visualization program communicates fully asynchronously with Maria via its standard input and output. It reads commands and graph descriptions from standard input and writes Maria commands to standard output. Maria is ready to read and execute these commands whenever it is sitting in the command prompt waiting for input.

Maria uses two commands of the visualization program. It issues the ‘`new();`’ command followed by a graph description whenever a query language command is preprended with a ‘`visual`’ keyword. This command causes a new graph to be displayed. The ‘`add();`’ command is issued whenever a query language command is prepended with several ‘`visual`’ keywords. This command causes the visualization program to merge the immediately following graph description with the last graph the user has interacted with. The visualization program always issues query language commands prepended with ‘`visual visual`’, causing the currently displayed graph to be extended.

If you want to access the visualization commands generated by Maria, you can rename the ‘`maria-vis`’ script and replace it with something like the following script:

```
#!/bin/sh
tee /tmp/maria-vis.out | exec maria-vis-orig "$@"
```

In order to print a visualized graph, you can save it to a file by selecting a command in the graphical user interface, and then input this file to the ‘`dot`’ command. Alternatively, you can use a script like the above one and extract the graph descriptions from the intercepted log file. As the ‘`dot`’ command does not directly support the DIN A4 paper size, you may want to try out one of the following command lines:

```
dot -Gsize=11.69,8.26 -Gcenter=1 -Gmargin=0 -Grotate=90 -Tps
dot -Gsize=8.26,11.69 -Gcenter=1 -Gmargin=0 -Tps
```

The former line is for horizontal layouts and the latter is for vertical layouts. You may want to add ‘`-Grankdir=LR`’ to force left-to-right layouting of the graph instead of the default top-to-bottom. The attributes in the graph file override the command line switches.

Both command lines act as filters, reading the graph description from standard input and writing the Postscript code to standard output. The Postscript can be printed directly as such, or embedded in a document, since it follows the Encapsulated Postscript conventions.

2.2.4.2 Known Bugs in the Visualizer

Graphical user interfaces are more challenging to program than plain textual programs. We have avoided most of the problems by choosing an external tool that lays out directed graphs. The GraphViz developer team has been very responsive and nice towards us, even though it has limited resources.

Two bugs that may not be fixed soon concern labels in the graph. Some GraphViz utilities fail if labels are longer than about one thousand characters. You can use the '`hide`' command (see Section 2.2.2.9 [Hide], page 40) to shorten node labels. Also backslash characters in labels may be handled incorrectly.

There are also some known bugs in Maria. Presently, Maria identifies paths only by state numbers. When a path is displayed graphically, Maria displays all transitions between each neighboring state along the path.

This treatment of transitions may be confusing especially when displaying counterexample paths (see Section 3.2.2 [Liveness], page 54), since Maria omits the property automaton states from the counterexample. Effectively, it projects the product of the reachability graph and the property automaton on the reachability graph. If there are several enabled actions between two states, or if fairness constraints are present, the presented counterexample path may contain extraneous transitions.

2.3 Editing Petri Nets with GNU Emacs

We recommend that you use GNU Emacs for editing Petri Net models. There is an Emacs major mode for editing Maria Petri Nets. Its main features are context-sensitive indentation and syntax highlighting. If you are not familiar with these features of Emacs, it is recommended that you read the Emacs tutorial (press *`C-h t`*, that is, first hold down the *`Control`* key, press *`h`*, then release *`Control`* and press *`t`*) and learn about these features from the on-line help system of Emacs e.g. with the commands *`C-h i`* (*`M-x info`*) and *`C-h a`* (*`M-x apropos-command`*).

2.3.1 Installing the Petri Net mode

The Petri Net mode '`pn-mode`' has been written for GNU Emacs, versions 20.3 and 21. It does not work in version 19.34, which lacks '`cc-mode`' and some features of '`font-lock-mode`' the Petri Net mode requires.

Installing the Petri Net mode is simple. Just add the name of the directory containing '`pn-mode.el`' to *load-path* and do '`(require 'pn-mode)`'. To do this, you can add e.g. the following lines to your '`.emacs`' file.

```
(cond ((>= emacs-major-version 20)
       (let ((pn-lisp-dir (expand-file-name "~/elisp")))
         (cond ((file-readable-p pn-lisp-dir)
                (add-to-list 'load-path pn-lisp-dir))
                (require 'pn-mode)
                (setq pn-font-lock-extra-types
                      '("token" "\\sw+_t")))))))
```

In this example, we also set the variable *pn-font-lock-extra-types* so that the word '`token`' and all words ending in '`_t`' will be treated as type names.

2.3.2 Syntax Highlighting

If you are not familiar with the current syntax highlighting features of Emacs ('`font-lock-mode`'), you should read the documentation (using the Info system and maybe also the `C-h f` and `C-h v` commands).

Adding the following declarations to your '`.emacs`' file will enable maximum degree of syntax highlighting in all Emacs modes. You may want to consult the documentation of the variable *font-lock-maximum-decoration* if you want to limit the degree of highlighting in some modes.

```
(global-font-lock-mode t)
(setq font-lock-maximum-decoration t)
(turn-on-font-lock)
```

2.3.3 Customizing Emacs

One of the most frequent problems of Emacs newcomers is how to convince Emacs to use 8-bit characters, as in the character set standardized by ISO 8859-1, also known as the Latin-1 alphabet. The default 7-bit character set will suffice for editing Petri Net models. But since this section is about fine-tuning, here is something for your '`.emacs`' file:

```
(standard-display-european t)
(set-input-mode nil nil 'dummy)
(cond ((< emacs-major-version 20)
       (require 'iso-syntax))
      (t
       (setq default-enable-multibyte-characters nil)
       (set-language-environment "Latin-1")))
```

In the following you will see some more definitions from my '`.emacs`' file. Feel free to use any of them. In order for '`pn-mode`' to automatically register the '`.pn`' extension with '`speedbar`', the latter should be loaded first.

```
(setq next-line-add-newlines nil);no accidentally inserted empty lines
(setq make-backup-files nil)      ;unless you like the '~' files
(setq mouse-yank-at-point t)      ;for more accurate yanking (pasting)
(setq inhibit-startup-message t) ;disable the startup message
(if (null window-system)
    (menu-bar-mode -1)            ;hide the menu line in text mode
  (if (>= emacs-major-version 20)
      (speedbar-frame-mode t)))  ;enable speedbar, the navigator
(setq visible-bell t)     ;do not beep—flash instead
(line-number-mode t)      ;show line numbers in the mode line
(column-number-mode t)    ;show column numbers in the mode line
(show-paren-mode t)       ;highlight matching parentheses
(auto-compression-mode t);transparently (de)compress '.gz' files
```

3 Algorithms used in Maria

3.1 The Unification Algorithm

Enabled transition instances are sought in a process called *unification*.

3.1.1 Concepts

There are some concepts related to the unification process that may be somewhat unclear, even though you know the basics of high level Petri Nets.

input arc An arc leading from a place to a transition, labelled with a multi-set expression evaluating to the tokens that are to be removed from the place when the transition fires

output arc
An arc leading from a transition to a place, labelled with a multi-set expression evaluating to the tokens that are to be inserted to the place when the transition fires

variable A named and strictly typed entity that may be assigned a value

output variable
A variable occurring only on output arcs of a transition; the value will be picked non-deterministically at the end of the unification process

input variable
A variable occurring on the input arcs of a transition

valuation
binding A binding of values to variables

expression A formula that evaluates to a value; expressions in net descriptions do not contain temporal logic operators or other set operations than multi-set summing

arc expression
A multi-set valued expression occurring on an input or output arc

gate
gate expression
A condition for the valuation of the input variables; if omitted, it is treated as identically true

token an entity having a value; moved between places by the firing of transitions

concrete token
a token contained in a place

abstract token
an item of an arc expression

lvalue Left-hand-side value of an assignment; a data object (in our case, an input variable) that is assigned to *rvalue* Right-hand-side value of an assignment, evaluating to the value that will be assigned to the lvalue

3.1.2 Expanding Quantifications

Arc expressions may contain multi-set sums (see Section 1.6 [Multi-Sets], page 24) or universal or existential quantification with fixed or variable bounds. They are expanded at parsing time. Consider the following net description:

```
place p bool: false, true;
place r bool: 2#(2#false, true);
trans t in {
  place p: p;
  place r: bool s: (s, bool t (p): s && t);
};
```

The arc expression is expanded as follows:

```
trans t in {
  place p: p;
  place r: false, true, (p?3:0)#false, (p?1:0)#true;
};
```

3.1.3 Matching Concrete and Formal Tokens

The Petri Net formalism does not have the concept of assignment as known in programming languages. Also, expressions may not have any side-effects. The only way a Petri system can change its state is through the firing of an enabled transition.

Assignments in the high level Petri Net formalism are implicit. In order for a high-level transition to be enabled, its variables must be bound to suitable values. Values for input variables will be found by matching concrete tokens in places with abstract ones on input arc expressions.

For each abstract token with known multiplicity, all concrete tokens in the input place that have not been associated with other abstract tokens will be considered. If the tokens are compatible (given the valuation generated so far, each component of the abstract token evaluates to the corresponding component of the concrete token or is undefined), a quantity of the concrete token will be associated with the abstract token. For instance, if the place contains '`3#true,2#false`' and the abstract token is '`2#x`', the unification algorithm will associate 2 of the 3 '`true`' tokens or all the '`false`' tokens with the abstract token.

If the abstract token contains assignment candidates (see Section 3.1.4 [Lvalues], page 53), the valuation will be extended. Continuing our example, the variable '`x`' in the abstract token '`2#x`' is an lvalue, and it can be assigned the corresponding value of the concrete token, in this case '`true`' or '`false`'.

3.1.4 Finding Assignment Candidates

Assignment candidates, or left-hand-side values of assignments (*lvalues*), are expressions that can be assigned to. In Maria, they are variables or array variables indexed by a known index. Lvalues must occur either as such (an abstract token is an lvalue) or in components of structured expressions.

For example, the abstract token ‘42#{x,{y+1,z},+t}’ has two lvalues, the variables ‘x’ and ‘z’. The abstract token ‘x#y’ has one lvalue, ‘y’; the token cannot be matched with a concrete one until the value of ‘x’ is known. The token ‘|x’ does not have any lvalues.

3.1.5 Transition Instance Analysis

A preprocessor sorts the input arcs of each transition in such a way that they can be processed in one pass. Static analysis finds out for each input arc the set of variables that are assigned a value based on the token assigned to the arc expression.

Tokens are assigned to input arcs by a depth-first search algorithm. If no variables are unified from an arc, the arc is "constant" under the assignment gathered so far. In this case, the corresponding input place is searched for the token the arc evaluates to. Otherwise, the arc is matched with each token in the input place, one at a time, and the assignment is augmented in such a way that evaluating the arc expression under the augmented (but still incomplete) assignment can yield the token picked from the input place.

In general, the less variables there are in the arc inscriptions and the less tokens in the input places, the faster the instance analysis can be completed.

3.2 Model Checking Algorithms

The on-the-fly model checker in Maria verifies properties expressed in temporal logic by computing a product of a property automaton (which corresponds to a formula) and the reachability graph interpreted as an automaton.

3.2.1 Checking Safety Properties

Safety properties can be specified with ‘deadlock’ and ‘reject’ conditions (see Section 1.3.6.1 [Assertions], page 11), as well as with so-called safe LTL formulae that can be translated to automata on finite words. Also some built-in features of the modelling language, such as capacity constraints, marking-dependent initialization expressions, and checks for expression evaluation errors, can be viewed as safety checking.

While exploring the reachable state space, Maria reports all violations of safety properties and may abort the analysis in case of a fatal violation.

Safety properties can be verified without constructing a reachability graph; it is sufficient to construct the set of reachable states. The command line options ‘-M’, ‘-P’ and ‘-L’ are more efficient than ‘-m’, and the search can be distributed on multiple processors.

3.2.2 Checking Liveness Properties

Currently, the ‘eval’ command (see Section 2.2.2.7 [Eval], page 39) supports LTL formulae on state properties. It is possible to examine counterexamples (paths violating the property being verified), and the algorithm respects both weak and strong fairness conditions.

Appendix A The Grammar

This appendix presents the grammar of the Maria languages using a Bison-like Backus−Naur Form (see section “Languages and Context-Free Grammars” in *The GNU Bison Manual*) with the following extensions inspired by regular expressions (see section “Patterns” in *The Flex Manual*):

1. Square brackets (‘[]’) indicate optional symbols
2. Asterisk (‘*’) denotes any amount repetition (0 or more instances)
3. Parentheses (‘()’) are used for grouping grammar symbols

A.1 Terminal Symbols

By convention, terminal symbols are written in all-uppercase letters or as character strings enclosed in single quotation marks, and non-terminal symbols are written in all-lowercase letters. The non-trivial terminal symbols used in the grammar are as follows.

‘NUMBER’ decimal (‘[1-9][0-9]*’), octal (‘0[0-7]*’) or hexadecimal (‘0x[0-9a-fA-F]+’)

‘CHARACTER’
‘’*c*’’: Section 1.2.3.3 [Character Constants], page 3

‘STATE’ number of a state in the reachability graph: ‘@[1-9][0-9]*’, ‘@0[0-7]*’ or ‘@0x[0-9a-fA-F]+’

‘COMP’ number of a strongly connected component of (part of) the reachability graph ‘@@[1-9][0-9]*’, ‘@@0[0-7]*’ or ‘@@0x[0-9a-fA-F]+’

‘name’ ‘[a-zA-Z_][0-9a-zA-Z_]*|"*c**"’: Section 1.2.3.4 [Identifiers], page 4

Reserved words are not listed in the above table. For instance, the symbol ‘PLACE’ stands for the keyword ‘place’.

In addition, there are a few low-level grammar rules that are almost like terminal symbols in their nature:

```
delim:
        ','
        |
        ';'
```

A.2 The Net Description Language

```
net:
        ( netcomponent ';' )*
```

```
netcomponent:
        type
        |
        function
        |
        place
        |
        transition
        |
        verify
        |
        fairness
        |
        proposition
        |
        subnet
subnet:
        SUBNET [ name ] '{' net '}'
```

A.2.1 Type

```
type:
        TYPEDEF typedefinition name
typedefinition:
        ENUM '{' enum_item ( delim enum_item )* '}'
        |
        STRUCT '{' [ comp_list ] '}'
        |
        UNION '{' comp_list '}'
        |
        ID '[' number ']'
        |
        typereference
        |
        typedefinition constraint
        |
        typedefinition '[' typedefinition ']'
        |
        typedefinition '[' QUEUE number ']'
        |
        typedefinition '[' STACK number ']'
typereference:
        name
enum_item:
        name [ [ '=' ] number ]
```

```
comp_list:
        comp ( delim comp )* [ delim ]
comp:
        typedefinition name
number:
        expr
```

A.2.1.1 Constraint

```
constraint:
        '(' range ( delim range )* ')'
range:
        value
        |
        '..' value
        |
        value '..' value
        |
        value '..'
value:
        expr
```

A.2.2 Function

```
function:
        typereference name eq formula
        |
        typereference name '(' param_list ')' formula
eq:     '=' | '()'
param_list:
        [ param_list_item ( delim param_list_item )* ]
param_list_item:
        typereference name
```

A.2.3 Place

```
place:
        PLACE name constraint* typedefinition
        [ CONST ] [ ':' marking_list ]
```

A.2.4 Transition

```
transition:
        TRANS [ ':' ] name [ '!' ] trans*
```

```
trans:
        '{' [ var_expr ( delim var_expr )* [ delim ] ] '}'
        |
        IN trans_places
        |
        OUT trans_places
        |
        GATE expr ( ',' expr )*
        |
        HIDE expr
        |
        STRONGLY_FAIR expr
        |
        WEAKLY_FAIR expr
        |
        ENABLED expr
        |
        ':' [ TRANS ] name
        |
        NUMBER
var_expr:
        [ HIDE ] typereference name
        |
        [ HIDE ] typereference name '!' [ '(' expr ')' ]
        |
        function
trans_places:
        '{' place_marking ( ';' place_marking )* '}'
place_marking:
        [ PLACE ] name ':' marking_list
```

A.2.5 State Properties

```
verify:
        REJECT expr
        |
        DEADLOCK expr
fairness:
        STRONGLY_FAIR qual_expr ( ',' qual_expr )*
        |
        WEAKLY_FAIR qual_expr ( ',' qual_expr )*
        |
        ENABLED qual_expr ( ',' qual_expr )*
proposition:
        PROP name ':' expr
```

A.3 The Query Language

Keywords of the query language are reserved words only in the beginning of a statement. For instance, ‘eval eval’ will try to evaluate the symbol ‘eval’ in the current state.

```
script:
        [ statement ( ';' statement )* [ ';' ] ]
statement:
        MODEL name
        |
        GRAPH name
        |
        [ VISUAL ] DUMP
        |
        UNFOLD name
        |
        LSTS [ name ]
        |
        [ VISUAL ] DUMPGRAPH
        |
        ( BREADTH | DEPTH ) [ STATE ]
        |
        [ VISUAL ] ( BREADTH | DEPTH ) formula
        |
        [ VISUAL ] [ EVAL ] [ STATE ] formula
        |
        [ VISUAL ] SHOW [ STATE ]
        |
        [ VISUAL ] SHOW STATE STATE ( STATE )*
        |
        HIDE [ '!' ] [ [ PLACE ] name ( ',' [ PLACE ] name )* ]
        |
        [ VISUAL ] ( SUCC | PRED ) [ '!' ] [ STATE ]
        |
        [ GO ] STATE
        |
        [ VISUAL ] [ STATE ] TRANS atrans*
atrans:
        '{' [ avar_expr ( delim avar_expr )* [ delim ] ] '}'
        |
        IN trans_places
        |
        OUT trans_places
        |
        GATE expr ( ',' expr )*
```

```
avar_expr:
        typereference name
        |
        typereference name '!' [ '(' expr ')' ]
statement:
        STRONG [ STATE ] [expr]
        |
        TERMINAL
        |
        [ VISUAL ] COMPONENTS
        |
        [ VISUAL ] ( SUCC | PRED ) COMP
        |
        [ VISUAL ] [ SHOW ] COMP [ expr ]
        |
        [ VISUAL ] PATH ( STATE | COMP | expr ) [ STATE ] [ ',' expr ]
        |
        [ VISUAL ] PATH STATE ( COMP | expr ) [ ',' expr ]
        |
        [ VISUAL ] PATH STATE STATE STATE ( STATE )* [ ',' expr ]
        FUNCTION function
        |
        STATS
        |
        TIME
        |

        CD [ name ]
        |
        TRANSLATOR [ name ]
        |
        COMPILEDIR [ name ]
        |
        LOG [ name ]
        |
        PROMPT [ 'c' ]
        |
        HELP
        |
        EXIT
```

A.4 Formulae and Expressions

```
expr:
        formula
```

```
marking:
        formula
marking_list:
        marking ( ’,’ marking )*
```

A.4.1 Literals

```
formula:
        TRUE | FALSE
        |
        NUMBER
        |
        CHARACTER
        |
        UNDEFINED | FATAL
        |
        ’#’ typereference
        |
        ’<’ typereference | ’>’ typereference
        |
        name
```

A.4.2 Functions

```
formula:
        name ’()’
        |
        name ’(’ arg_list ’)’
arg_list:
        [ formula ( ’,’ formula )* ]
```

A.4.3 Basic Formulae

```
formula:
        ’(’ formula ’)’
        |
        ATOM formula
        |
        formula ’<’ formula
        |
        formula ’==’ formula
        |
        formula ’>’ formula
        |
        formula ’>=’ formula
        |
        formula ’!=’ formula
        |
        formula ’<=’ formula
        |
        ’-’ formula
        |
        formula ’+’ formula
        |
        formula ’-’ formula
        |
        formula ’/’ formula
        |
        formula ’*’ formula
        |
        formula ’%’ formula
        |
        ’|’ formula | ’+’ formula
        |
        ’*’ formula
        |
        ’/’ formula | ’%’ formula
        |
        ’˜’ formula
        |
        formula ’<<’ formula
        |
        formula ’>>’ formula
        |
        formula ’&’ formula
        |
        formula ’ˆ’ formula
        |
        formula ’|’ formula
        |
        formula ’?’ formula ( ’:’ formula )*
        |
        '{' [ name ':' ] [ expr ] ( ',' [ name ':' ] [ expr ] )* '}'
        |
        formula '.' name
        |
        formula '.' '{' name expr '}'
        |
        formula '[' formula ']'
        |
        formula '.' '{' '[' expr ']' expr '}'
        |
        '!' formula
        |
        formula '&&' formula
        |
        formula '^^' formula
        |
        formula '||' formula
        |
        formula '<=>' formula
        |
        formula '=>' formula
```

A.4.4 Typecasting and Union Values

```
formula:
        IS typereference formula
        |
        name '=' formula
        |
        formula IS name
```

A.4.5 Non-Determinism and Quantification

```
formula:
        typereference [ name ] '!' [ '(' expr ')' ]
        |
        typereference name [ '(' expr ')' ] ':' formula
        |
        typereference name [ '(' expr ')' ] '&&' formula
        |
        typereference name [ '(' expr ')' ] '||' formula
        |
        '.' name [ name ]
        |
        ':' name [ name ]
```

A.4.6 Multi-Set Operations

```
     formula:
             EMPTY
             |
             '(' marking_list ')'
             |
             formula '#' marking
             |
             PLACE name
             |
             marking SUBSET marking
             |
             marking INTERSECT marking
             |
             marking MINUS marking
             |
             marking UNION marking
             |
             marking EQUALS marking
             |
             CARDINALITY marking
             |
             MAX marking
             |
             MIN marking
             |
             SUBSET name '{' marking_list '}' expr
             |
             MAP name '{' marking_list '}' expr
             |
             MAP name '#' name '{' marking_list '}' marking
```

A.4.7 Temporal Logic

```
             '<>' formula
             |
             '[]' formula
             |
             '()' formula
             |
             formula UNTIL formula
             |
             formula RELEASE formula
```

```
qual_expr:
        TRANS name [ ':' expr ]
        |
        '(' qual_expr ')'
        |
        '!' qual_expr
        |
        qual_expr '&&' qual_expr
        |
        qual_expr '^^' qual_expr
        |
        qual_expr '||' qual_expr
        |
        qual_expr '<=>' qual_expr
        |
        qual_expr '=>' qual_expr
```

Appendix B The Graph Files

For maximum flexibility and performance, the reachability graph is laid out in disk files in machine dependent format. Please refer to the files '`Graph/fileformats.html`' and '`Graph/Graph.C`' in the Maria source code for exact information.

Note that when Maria has been compiled for using a memory mapped file interface (see Section C.2 [Configuring], page 67), the sizes of the graph files will be rounded up to a power of two while Maria is running. This arrangement reduces the number of costly system calls.

Appendix C Compiling Maria

C.1 System Requirements

Special care has been taken to ensure the quality and portability of the Maria source code. The code should compile on any ISO/IEC 14882 compliant C++ compiler. Unfortunately the standard is fairly new (summer 1998), and until 2001 or 2002, many C++ compilers did not support even the subset of it that compiling Maria requires.

We use the GNU Compiler Collection (gcc) on Debian GNU/Linux as the development platform. GCC should be able to compile the program also on FreeBSD, NetBSD, OpenBSD and IBM AIX systems. Furthermore, the code can be compiled with the native compilers of Digital UNIX 4.0 and 5.1, HP-UX 11.22, Sun Solaris 8, SGI IRIX 6.2 and Apple Darwin 5.3.

It is recommended that you install the freely available gcc 2.95 or later on your system if you have problems compiling the code.

We would like to hear success reports from people using other compilers. Patches, even to the '`Makefile`' files, are welcome.

C.2 Editing the '`Makefile`' files

In the top-level directory, in '`Makefile`' and in '`Makefile`'.*system*, there are a couple of variable definitions that you should check before invoking the compilation by typing '`make`'. It is recommended to make '`reallyclean`' before re-starting the compilation after making modifications.[1]

'`HAS_READLINE`'
'`LIBREADLINE`'
'`INCREADLINE`'

Set '`HAS_READLINE=yes`' if the GNU Readline library is available, and ensure that the directories have been set up properly. The library is not a necessity; See Section 2.2.1 [Line Editor], page 36, for the features it provides. On some systems, you may need to add '`-lncurses`' to the list of libraries.

'`EXPR_COMPILE`'

Set '`EXPR_COMPILE=yes`' if your system supports dynamic loading of shared libraries and if you want to enable the '`-C`' command line option (see Section 2.1.2 [Maria Options], page 29), which will considerably speed up the reachability graph generation.

'`EXTRA_DEFINES`'

Extra definitions for the macro preprocessor. Set '`-DUSE_MMAP`' in order to enable memory mapped access to the reachability graph files on systems that implement the POSIX.1b mmap(2) interface. This option can considerably

[1] If the '`make`' command on your system does not seem to come along with the '`Makefile`', use GNU Make.

speed up the analysis of models that have a relatively small number of high-level transitions. Please note that with this option, 32-bit systems can only handle graph files whose total size is less than 4 gigabytes, maybe even less than 1 gigabyte, depending on the operating system.

'`DEBUG`' Debugging flags. Usually '`-DNDEBUG`' for compiling the production version and '`-g`' for the debugging version.

'`PROF`' Profiling flags. Usually empty.

'`CXX`'
'`CC`' Commands for compiling modules written in C++ and C, respectively.

'`EXTRA_INCLUDES`'
Define '`EXTRA_INCLUDES=-Idummy`' if your compiler does not implement the '`<slist>`' extension to the Standard Template Library.

'`EXTRA_LIBS`'
Define '`EXTRA_LIBS=-ldl`' or similar, if you want to enable support for compiled expressions and the dynamic loader routines are not part of the standard libraries on your system.

'`DEFINES`' Extra flags e.g. for enabling the usage of the POSIX regular expression library ('`-DHAS_REGEX`'), of the '`getopt_long`' function ('`-DHAS_GETOPT_LONG`'), a GNU extension to the standard, and of some extensions to the Standard Template Library ('`-DHASH_MAP_LOC`' and '`-DSLIST_LOC`').

'`CFLAGS`'
'`CXXFLAGS`'
Flags for the C and C++ compilers e.g. for enabling optimizations.

C.3 Installing Maria

The file '`Makefile`' in the top-level directory contains rules for installing the '`maria`' executable and some related files on Unix-like systems. The installation is invoked by typing '`make install installman installinfo`'. You may want to redefine some of the following variables either in the '`Makefile`' or on the '`make`' command line:

PREFIX The base directory where Maria should be installed. You might want to change this to '`/usr/local`'. The default is '`/usr`'.

INSTALLDIR
INSTALLBIN
INSTALLDATA
The commands for creating directories, installing executable files and installing data files. On Apple Darwin, you will need to add the '`-c`' switch to the latter two commands.

C.4 Compiling Maria for Debugging

Compiling a debugging version of Maria is simple: in '`Makefile`', define '`DEBUG=-g`', and you are all set. You could also disable '`assert()`' macros by defining '`-DNDEBUG`', but they are very useful, since they often catch errors introduced by modifying code that seems completely unrelated to the failed assertion at first sight.

For detecting and debugging memory management issues, you can use the Electric Fence Library, Valgrind for GNU/Linux, the Debug Malloc Library (See Info file ‘dmalloc’, node ‘Top’ or http://www.dmalloc.com) or commercial tools such as Third Degree (‘third’) on Digital UNIX.

Debuggers often have problems with C++. For us, the GNU debugger (gdb 5.0) and has worked pretty well on Debian GNU/Linux. Version 4.18 has problems calling virtual methods.

Defining the ‘YYDEBUG’ macro enables grammar debugging. An executable compiled with this macro defined will look for the environment variable DEBUG. When DEBUG=1, the parser will print out more than enough information on the parsing process. Often it makes sense to set a conditional breakpoint on the lexical analyzer function based on the input line number, and to enable the parser debugging output only for a certain region of the input by setting or clearing the status variable in the debugger.

C.5 Reporting Bugs

Every non-trivial program is likely to contain bugs. Fatal bugs, such as assertion failures or segmentation faults, are easiest to locate. Our intention has been to make the parsers in Maria bullet-proof: no matter what the input is, the program should not crash.

Sometimes a program may behave in a counter-intuitive way, doing something else than one would except. Such situations can be caused by a bug, or the program might behave just as planned. The latter case is often fixed by rephrasing or extending the documentation.

Bug reports and suggestions are welcome at ‘msmakela@tcs.hut.fi’. In the bug reports, please mention which platform you are using (including version numbers of the operating system and the relevant compilers and libraries) and include a stripped-down input file that is enough for triggering the bug. You can also compile the analyzer with support for debugging enabled, since it can help tracking down the error. Please use the ‘diff -c’ format for any patches you send.

Appendix D Examples

More examples can be found in the Maria source code, in the directory ‘parser/test’.

D.1 Dining Philosophers (‘dining.pn’)

```
#!/usr/local/bin/maria
typedef unsigned (1..5) philosopher;
typedef struct {
  philosopher p,
  enum { thinking, hungry, eating } s
} status;
place fork (0..#philosopher) philosopher: philosopher p: p;
place state (#philosopher) status: philosopher p: { p, thinking };
trans left
in { place state: { p, thinking }; place fork: p; }
out { place state: { p, hungry }; };
trans right
in { place state: { p, hungry }; place fork: +p; }
out { place state: { p, eating }; };
trans finish
in { place state: { p, eating }; }
out { place state: { p, thinking }; place fork: p, +p; };
```

D.2 Distributed Database Management (‘dbm.pn’)

The parameter of the model, the number of database agents, is only present in the data type definition. The initial marking expression and the arc expressions are independent of that parameter, thanks to the multi-set sum operator.

```
typedef unsigned (1..10) db_t;
typedef struct {
  db_t first;
  db_t second;
} db_pair_t;

place waiting (0..1) db_t;
place performing (0..#db_t-1) db_t;
place inactive (0..#db_t) db_t: (db_t d: d)
  minus (place waiting union place performing);
place exclusion (0..1) struct {}: (place waiting equals empty)#{};

place sent (0..#db_t-1) db_pair_t;
place received (0..#db_t-1) db_pair_t;
place acknowledged (0..#db_t-1) db_pair_t;

place unused (1+#db_pair_t-2*#db_t,#db_pair_t-#db_t) db_pair_t:
(db_pair_t p (p.first != p.second): p)
  minus (db_t t: (map s { place waiting } {s, t}));
```

```
trans update_and_send_messages
in {
  place inactive: s;
  place exclusion: {};
  place unused: db_t t (t != s): { s, t };
}
out {
  place waiting: s;
  place sent: db_t t (t != s): { s, t };
};
```

```
trans receive_acknowledgements
in {
  place waiting: s;
  place acknowledged: db_t t (t != s): { s, t };
}
out {
  place inactive: s;
  place exclusion: {};
  place unused: db_t t (t != s): { s, t };
};
```

```
trans receive_message
in {
  place inactive: r;
  place sent: { s, r };
}
out {
 place performing: r;
 place received: { s, r };
}
gate s != r;
```

```
trans send_acknowledgement
in {
  place performing: r;
  place received: { s, r };
}
out {
  place inactive: r;
  place acknowledged: { s, r };
}
gate s != r;
```

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

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.

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

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) 19yy  name of author

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330,
Boston, MA 02111-1307, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type 'show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'show c' for details.
```

The hypothetical commands '`show w`' and '`show c`' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than '`show w`' and '`show c`'; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the program
'Gnomovision' (which makes passes at compilers) written by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Index

Table of Contents