NAME

mcl, MCL – Model Checking Language version 4 (value-passing modal mu-calculus)

DESCRIPTION

This manual page presents the version 4 of *MCL* (*Model Checking Language*), which is the temporal logic accepted as input by **evaluator4**(LOCAL). In the remainder of this page, "*MCL*" denotes version 4 of *MCL*; see **mcl**(LOCAL) for other versions of *MCL*.

A description of *MCL* can be found in article [MT08], which also describes the verification method implemented in version 4.0 of EVALUATOR.

The *MCL* language attempts to make a compromise between expressiveness, user-friendliness, and efficiency of model checking for temporal properties involving data. *MCL* is based on the alternation-free fragment of the modal mu-calculus [Koz83, EL86], to which it brings two kinds of extensions:

- Action predicates equipped with data variables and expressions, modalities containing extended regular expressions over action sequences, parameterized fixed point operators, and data-handling constructs inspired from the RICO temporal logic [Gar89] and from programming languages.
- An infinite looping operator (of alternation depth two [EL86]) similar to the one present in *PDL-delta* (Propositional Dynamic Logic with Looping) [Str82], which enables the expression of fairness properties by characterizing complex unfair cycles of transitions in the LTS.

An overview of the *MCL* language is presented below. The abstract syntax of each language construct is defined by a BNF grammar and the semantics is described informally. In the grammar, terminal symbols are written between double quotes. Optional constructs are enclosed between square brackets. The axiom of the grammar is the *F* symbol.

The following convention is adopted for the lists of symbols occurring in the grammar rules: the index *n* of the last symbol in the list is always greater or equal to 0, meaning that if the index of the first symbol is 0 (e.g., *E0*, ..., *En*), then the list must contain at least one symbol, and if the index of the first symbol is 1 (e.g., *E1*, ..., *En*), then the list may be empty.

When referring to a certain construct of the grammar, the term "enclosing formula" denotes the (action, regular, or state) formula immediately surrounding that construct.

	Symbol	Description
Non-terminal	*E*	expression
	P	pattern
	O	offer
	AP	action pattern
	A	action formula
	R	regular formula
	F	state formula
Terminal	*K*	constant
	X	data variable
	Y	propositional variable
	H	function or operator
	T	type

The formulas *A*, *R*, *F* are interpreted over an LTS *<S, A, T, s0>*, where: *S* is the set of *states*, *A* is the set of *actions* (transition labels), *T* is the *transition relation* (a subset of $S * A * S$), and *s0* is the *initial state*. A transition *(s1, a, s2)* of *T*, also written *s1-a->s2*, indicates that the system can move from state *s1* to state *s2* by performing action *a*. An action *a* has the following structure:

G v1 ... vn

where *G* is the name of a gate (communication channel) and *v1*, ..., *vn* are the values exchanged on *G* when the rendezvous underlying action *a* was executed. In the case that there are no values exchanged, the action is simply a gate name. There is an invisible action (named *i* in LOTOS and *tau* in other process algebras).

Note: Actions are also represented as character strings, which is useful for expressing certain action predicates (see ACTION FORMULAS below). The character string representation of actions depends on the language from which the LTS *spec* is generated. For example, if the input is given as a LOTOS program *spec*[**.lotos**], an action having the structure shown above will be represented as the character string "*G !v1 ... !vn*".

LEXICAL ELEMENTS

Identifiers are built from letters, digits, and underscores (beginning with a letter or an underscore). **evaluator4** is case-sensitive, except for the identifiers of predefined types and functions (see TYPES, FUNCTIONS AND CONSTANTS below). Keywords must be written in lowercase. Comments are enclosed between '**(***' and '***)**'. Nested comments are not allowed. The keywords of *MCL* are listed below.

among	**equ**	**in**	**repeat**
and	**exists**	**let**	**step**
any	**exit**	**loop**	**tau**
case	**export**	**mu**	**then**
choice	**false**	**nil**	**to**
continue	**for**	**not**	**true**
do	**forall**	**nu**	**until**
else	**from**	**of**	**where**
elsif	**if**	**on**	**while**
end	**implies**	**or**	**xor**

TYPES, FUNCTIONS AND CONSTANTS

MCL is a strongly-typed language: every variable used in an *MCL* formula must have a unique type, which is statically determined. In its current version, the language does not provide a mechanism for type or function definition. *MCL* predefines the usual types encountered in programming languages (**bool**, **nat**, **natset**, **int**, **real**, **char**, **string**), equipped with the standard functions and operators listed below.

Operator	Meaning
false, true : -> bool	boolean constants
not : bool -> bool	negation
or, and, implies, equ, xor : bool, bool -> bool	binary boolean operators
<, <=, >, >=, =, <> : bool, bool -> bool	comparison operators
succ : nat -> nat	successor
- : nat -> int	unary minus
+, - : nat, nat -> nat	addition, subtraction
*, / : nat, nat -> nat	multiplication, division

% : nat, nat -> nat	modulo
^ : nat, nat -> nat	power
string : nat -> string	convert to string
<, <=, >, >=, =, <> : nat, nat -> bool	comparison operators
empty : -> natset	empty set
insert : nat, natset -> natset	element insertion
remove : nat, natset -> natset	element deletion
isin : nat, natset -> bool	membership
union, inter, diff : natset, natset -> natset	binary set operators
<, <=, >, >=, =, <> : natset, natset -> bool	comparison operators
succ : int -> int	successor
abs : int -> nat	absolute value
sign : int -> int	returns -1, 0, 1 if arg. is < 0, = 0, > 0
- : int -> int	unary minus
+, - : int, int -> int	addition, subtraction
*, / : int, int -> int	multiplication, division
% : int, int -> int	modulo
^ : int, nat -> int	power
string : int -> string	convert to string
<, <=, >, >=, =, <> : int, int -> bool	comparison operators
- : real -> real	unary minus
+, - : real, real -> real	addition, subtraction
*, / : real, real -> real	multiplication, division
^ : real, real -> real	power
string : real -> string	convert to string
<, <=, >, >=, =, <> : real, real -> bool	comparison operators
tolower, toupper : char -> char	convert letter to lower- or upper-case
islower, isupper : char -> bool	test if lower- or upper- case letter
isalpha, isdigit, isalnum : char -> bool	test if letter, digit, letter or digit
isxdigit : char -> bool	test if hexadecimal digit
string : char -> string	convert to string
<, <=, >, >=, =, <> : char, char -> bool	comparison operators
length : string -> nat	length (number of chars)
empty : string -> bool	test if empty string
concat : string, string -> string	concatenation
index, rindex :	index of the first/last

string, string -> nat	occurrence of the 2nd arg. in the 1st
prefix, suffix : string, nat -> string	prefix/suffix of given length
nth : string, nat -> char	nth character
substr : string, nat, nat -> string	substring starting at an index and having a given length
<, <=, >, >=, =, <> : string, string -> bool	comparison operators

The numerical, character, and string constants have a C-like syntax (e.g., **13**, **-1**, **1.618**, **'a'**, **'\007'**, **'\n'**, **"hello world\n"**). In order to allow implicit type conversions, numerical constants are overloaded as follows: a constant of type **nat** can also be of type **int** or **real**, and a constant of type **int** can also be of type **real**.

The names of the operators of type **bool** (constants, negation, and binary operators) that coincide with the keywords operating on formulas must be written in lowercase.

The binary boolean operators, binary set and membership operators, the arithmetic operators (of types **nat**, **int**, and **real**), and all comparison operators must be written in infixed form (e.g., **1 + 2**, **X inter Y**, **1.0 < 2.0**, etc.). All the other operators must be written in prefixed form (e.g., **insert (0, empty)**, **concat ("a", "b")**, etc.).

The operands of the binary operators of type **bool** ("**or**", "**and**", "**implies**", "**equ**", "**xor**") are evaluated from left to right in a lazy way, i.e., the right operand is not evaluated if the value of the left operand can determine the value of the whole expression.

All the binary operators of the predefined types shown above are left-associative. Unary operators have the highest precedence, followed by binary operators. In the current version of the tool all binary operators are considered to be of equal precedence; parentheses must be used for imposing a desired parsing/evaluation order.

EXPRESSIONS

The syntax of *MCL* expressions is defined by the following grammar:

E **::=** *K*

| *X*

| *H E*

| *E1 H E2*

| *H* **"("** *E1* **","** **..."** **,"** *En* **")"**

| *E* **"of"** *T*

| **"("** *E* **")"**

The semantics of *MCL* expressions is described informally below. The evaluation of an expression *E* in a context assigning values to all data variables occurring in *E* yields a unique value.

K

is a literal constant of a predefined type (see TYPES, FUNCTIONS AND CONSTANTS above).

X

is a data variable (see DECLARATIONS below).

H E

denotes a call of the unary prefixed operator *H* on the argument *E*. The argument must be of the same type as the formal parameter of *H*.

E1 H E2

denotes a call of the binary infixed operator *H* on the arguments *E1* and *E2*. The arguments must be of the same types as the corresponding formal parameters of *H*.

H "(" *E1*"," ..."," *En* ")"

denotes a call of the function *F* on the arguments *E1*, ..., *En*. The arguments must be compatible (in number and type) with the formal parameters of *H*.

E "of" *T*

specifies that expression *E* has type *T*. This mechanism makes it possible to solve ambiguities that may be caused by the overloading of operators and constants (see TYPES, FUNCTIONS AND CONSTANTS above).

"(" *E* ")"

has the same meaning as the expression *E*. Parentheses are useful for imposing an evaluation order of subexpressions different from the order given by the associativity and precedence of operators. For example, **`x + (y / 2)`** is different from **`x + y / 2`**, which is evaluated by default as **`(x + y) / 2`** since all binary operators have the same precedence.

DECLARATIONS

Similarly to classical functional programming languages, *MCL* provides mechanisms for declaring and initializing data variables. Declarations (without initialization) have the following general form:

x01 **","** **...** **","** *x0m0* **":"** *T0*
"," **...** **","**
xn1 **","** **...** **","** *xnmn* **":"** *Tn*

which declares the data variables *xi1*, ..., *ximi* of type *Ti* for each 0 <= *i* <= *n*. This general form of declaration is equivalent to the simplified form below, which will be used in the remainder of this manual page:

x01 **":"** *T0* **","** **...** **","** *x0m0* **":"** *T0*
"," **...** **","**
xn1 **":"** *Tn* **","** **...** **","** *xnmn* **":"** *Tn*

In the same way, declarations with initialization have the following general form:

x01 **","** **...** **","** *x0m0* **":"** *T0* **":="** *E0*
"," **...** **","**
xn1 **","** **...** **","** *xnmn* **":"** *Tn* **":="** *En*

which declares the data variables *xi1*, ..., *ximi* of type *Ti* and initializes them with the value of the expression *Ei* for each 0 <= *i* <= *n*. This general form of declaration with initialization is equivalent to the simplified form below, which will be used in the remainder of this manual page:

x01 **":"** *T0* **":="** *E0* **","** **...** **","** *x0m0* **":"** *T0* **":="** *E0*
"," **...** **","**
xn1 **":"** *Tn* **":="** *En* **","** **...** **","** *xnmn* **":"** *Tn* **":="** *En*

PATTERNS

MCL allows the manipulation of data values by matching them against patterns and storing them in data variables. The syntax of *MCL* patterns is defined by the following grammar:

P **::= "any"**

| *K*

| *X* **":"** *T*

| *P* **"of"** *T*

| *P1* **"|"** *P2*

The semantics of *MCL* patterns is described informally below.

"any"

is the "wildcard" pattern, which matches any value of any type.

K

is the constant pattern, which matches a value identical to *K*.

X ":" *T*

matches any value of type *T* and stores it in variable *X*, which is exported to (i.e., made visible in) the enclosing formula.

P "of" *T*

removes ambiguities (caused, e.g., by overloaded functions) by imposing that a value can be matched by *P* only if it is of type *T*.

P1 "|" *P2*

matches a value if either *P1* or *P2* matches it. The patterns *P1* and *P2* must declare the same data variables, all of which are exported to the enclosing formula.

OFFERS

Expressions and patterns can be used in offers, which enable one to match a given value against an expression or to extract and store it in a variable. The syntax of *MCL* offers is defined by the following grammar:

O **::= "?"** *P*

| **"!"** *E*

The semantics of *MCL* offers is described informally below.

"?" *P*

is a pattern offer, which matches a value iff the pattern *P* matches that value. All variables declared in *P* are exported to the enclosing formula.

"!" *E*

is an expression offer, which matches a value iff the evaluation of the expression *E* yields that value, which also means that *E* and that value must be of the same type.

Note: Pattern and expression offers involving constants have the same semantics, e.g., **`? 3.1416`** is equivalent to **`! 3.1416`**.

ACTION PATTERNS

An action pattern *AP* specifies that a certain action (transition label) of the LTS matches a list of offers. The syntax of *MCL* action patterns is defined by the following grammar:

AP **::= "{"** *O0* **...** *On* **["where"** *E* **] "}"**

| **"{"** *O1* **...** *On* **"..."** *O'1* **...** *O'm* **["where"** *E* **] "}"**

Action patterns inspect the structure of actions *G v1 ... vn* by matching values *vi* against expression offers or extracting them using pattern offers. The optional clause "**`where`**" defines a boolean expression *E* (a guard) that must evaluate to true for the action pattern to match the action. All variables declared by the offers of an action pattern are visible in the guard *E* (if present) and are also exported to the enclosing formula.

The gate name *G* can be matched by the first offer of an action pattern in three different ways:

- As a character string constant, using an expression offer (e.g., **`!"Send"`**);
- As a gate identifier, using a particular form of expression offer without the "**`!`**" mark (e.g., **`Send`**). This form of matching can be applied only when the gate name has a syntax compatible with the syntax of *MCL* identifiers;
- As a character string value, using a pattern offer (e.g., **`?gate:string`**).

The semantics of *MCL* action patterns is described informally below.

"{" *O0 ... Om* ["where" *E*] "}"

matches an action *G v1 ... vn* iff *m = n*, the offer *O0* matches *G*, each offer *Oi* (for *i* = 1..*n*) matches its corresponding value *vi*, and the expression *E* (if present) evaluates to true in a context in which all variables declared in the offers *O0*, ..., *Om* are replaced with the corresponding values.

This is the basic action pattern, in which all values present in the action are explicitly matched by offers. The matching of the gate name *G* by the offer *O0* can be done in one of the three ways indicated above.

The simplest action pattern of this form consists of a gate name (e.g., **`{ Send }`**). For conciseness, the curly braces can be omitted in this case: one can write simply **`Send`** in order to match an action consisting only of a gate name **`Send`**.

"{" *O1 ... Om* "..." *O'1 ... O'p* ["where" *E*] "}"

matches an action *G v1 ... vn* iff *m+p* <= *n*, the offer *O1* (if present, i.e., if *m* > 0) matches *G*, each offer *Oi* (for *i* = 1..*m*) matches its corresponding value *vi*, each offer *O'j* (for *j* = 1..*p*) matches its corresponding value *vn-(p-j)*, and the expression *E* (if present) evaluates to true in a context in which all variables declared in the offers *O1*, ..., *Om*, *O'1*, ..., *O'p* are replaced with the corresponding values.

This is a form of action pattern that enables matching only the first *m* and the last *p* values contained in an action, and skipping the other values (if any) in the middle. The matching of the gate name *G* by the offer *O1* can be done in one of the three ways indicated above. Either one, or both groups of offers *O1 ... Om* and *O'1 ... O'p* can be absent (i.e., *m* = 0 or/and *p* = 0). The simplest action pattern of this form (which is always matched by any action) is **`{ ... }`**.

ACTION FORMULAS

An *action formula* is a logical formula built from action predicates (which can be action patterns, character strings, regular expressions over character strings, and the "**`tau`**" constant operator) and boolean operators. The syntax of *MCL* action formulas is defined by the following grammar:

A **::=** *AP*

| *action_string*

| *action_regexp*

| **"tau"**

| **"true"**

| **"false"**

| **"not"** *A*

| *A1* **"or"** *A2*

| *A1* **"xor"** *A2*

| *A1* **"and"** *A2*

| *A1* **"implies"** *A2*

| *A1* **"equ"** *A2*

| **"("** *A* **")"**

Syntactically, all binary operators on action formulas are left-associative. The "**not**" operator has the highest precedence, followed by "**and**", followed by "**or**" and "**xor**", followed by "**implies**", followed by "**equ**".

An action formula defines a predicate over the actions of the LTS. The semantics of *MCL* action formulas is described informally below.

AP

an action (transition label) of the LTS satisfies an action pattern *AP* if the content of the action matches the pattern. In this case, all variables declared by the offers of *AP* are initialized with the corresponding values extracted from the action and are also exported to the enclosing formula.

action_string

an *action_string* is a sequence of 0 or more characters, enclosed between double quotes ('**"**'), which denotes an action of the LTS. A string may contain any character but '**\n**' (end-of-line). Double quotes are also allowed, if preceded by a backslash ('****'). Strings can be concatenated using the binary operator '**#**' according to the grammar below:

action_string **::= "(***any char but end-of-line***)*"**

| *action_string1* "#" *action_string2*

An action of the LTS satisfies an *action_string* iff its string representation is identical to the corresponding character string (obtained after concatenation whenever needed).

action_regexp

an *action_regexp* is a UNIX regular expression (see the **regexp**(LOCAL) manual page for a detailed description of UNIX regular expressions), enclosed between single quotes ('**'**'), which denotes a predicate on the actions of the LTS. Regexps can be concatenated using the binary

operator '**#**' according to the grammar below. Strings can be concatenated to regexps, in which case they are implicitly converted into regexps.

action_regexp **::=** *'UNIX_regular_expression'*

| *action_regexp1* "#" *action_regexp2*

| *action_string1* "#" *action_regexp2*

| *action_regexp1* "#" *action_string2*

An action of the LTS satisfies an *action_regexp* iff its string representation matches the corresponding *UNIX_regular_expression* (obtained after concatenation whenever needed).

"tau"

an action of the LTS satisfies this action formula iff it is the invisible action.

"true"

an action of the LTS always satisfies this formula.

"false"

an action of the LTS never satisfies this formula.

"not" *A*

an action of the LTS satisfies this formula iff it does not satisfy *A*.

A1 "or" *A2*

an action of the LTS satisfies this formula iff it satisfies *A1* or it satisfies *A2*.

A1 "xor" *A2*

an action of the LTS satisfies this formula iff it satisfies exactly one of *A1* and *A2*.

A1 "and" *A2*

an action of the LTS satisfies this formula iff it satisfies both *A1* and *A2*.

A1 "implies" *A2*

an action of the LTS satisfies this formula iff it does not satisfy *A1* or it satisfies *A2*.

A1 "equ" *A2*

an action of the LTS satisfies this formula iff either it satisfies both *A1* and *A2*, or neither of them.

"(" *A* ")"

an action of the LTS satisfies this formula iff it satisfies *A*. Parentheses are useful for imposing an evaluation order of subformulas different from the order given by the associativity and precedence of operators.

If an action pattern *AP* occurs as operand of a unary or binary boolean operator, none of the data variables declared by the pattern offers of *AP* is exported to the enclosing formula (e.g., action formula **`not { Send ?msg:nat }`** does not export variable *msg* to the enclosing formula). In other words, only the action formulas consisting of action patterns can export data variables to the enclosing formula (see also REGULAR FORMULAS and the description of modalities in STATE FORMULAS below).

REGULAR FORMULAS

A *regular formula* is a logical formula built from action formulas, traditional and extended regular expression operators, and data-handling constructs inspired from functional programming languages. The syntax of *MCL* regular formulas is defined by the following grammar:

R **::=** *A*

| **"nil"**

| *R1* **"."** *R2*

| *R1* **"|"** *R2*

| *R* **"*"**

| *R* **"+"**

| *R* **"?"**

| *R* **"{"** *E* **"}"**

| *R* **"{"** *E* **"..."** **"}"**

| *R* **"{"** *E* **","** **"}"**

| *R* **"{"** *E1* **"..."** *E2* **"}"**

| *R* **"{"** *E1* **","** *E2* **"}"**

| **"let"** *X0* **":"** *T0* **":="** *E0* **","** **...** **","** *Xn* **":"** *Tn* **":="** *En* **"in"**
R
"end" **"let"**

| **"if"** *F0* **"then"**
R0
[**"elsif"** *F1* **"then"**
R1
...
"elsif" *Fn* **"then"**
Rn
"else"
Rn+1 **]**
"end" **"if"**

| **"case"** *E* **"in"**
P0 **[** **"where"** *E0* **]** **"->"** *R0*
...
"|" *Pn* **[** **"where"** *En* **]** **"->"** *Rn*
"end" **"case"**

| **"choice"** *X0* **":"** *T0* **[** **"among"** **"{"** *E01* **"..."** *E02* **"}"** **]**
"," **...** **","**
Xn **":"** *Tn* **[** **"among"** **"{"** *En1* **"..."** *En2* **"}"** **]**
"in"
R
"end" **"choice"**

| **"while"** *F* **"do"**

```
    R
  "end" "while"

| "repeat"
    R
  "until" F "end" "repeat"

| "for" X ":" T "from" E1 "to" E2 [ "step" E3 ] "do"
    R
  "end" "for"

| "loop" [ "(" X0 ":" T0 ":=" E0 "," ... "," Xn ":" Tn ":=" En ")" ]
      [ ":" "(" X'0 ":" T'0 "," ... "," X'm ":" T'm ")" ]
  "in"
    R
  "end" "loop"

| "continue" [ "(" E0 "," ..."," En ")" ]

| "exit" [ "(" E0 "," ... "," Em ")" ]

| "export" "(" X0 ":" T0 ":=" E0 "," ... "," Xn ":" Tn ":=" En ")"

| "(" R ")"
```

Syntactically, all binary operators on regular formulas are left-associative. The "*", "**+**", "**?**", and "**{** ... **}**" operators have the highest precedence, followed by "**.**", followed by "**|**".

A regular formula *R* denotes a sequence (represented by the couple of its source and target states) of consecutive LTS transitions such that the word obtained by concatenating the actions labeling them belongs to the regular language defined by *R*.

A transition sequence *weakly satisfies* a regular formula *R* iff, by deleting some of its invisible transitions, the resulting sub-sequence satisfies *R*. In other words, the transitions of the sequence matched by the action predicates of *R* (which can denote either visible or invisible actions) can be interspersed with sub-sequences of 0 or more invisible transitions.

The semantics of *MCL* regular formulas is described informally below.

A

> is the action regular formula, which denotes one-step transition sequences. It is satisfied by a sequence of LTS transitions iff this sequence consists of a single transition labeled by an action satisfying the action formula *A*.
>
> All data variables exported by *A* are also exported to the enclosing formula.

"nil"

> is the null regular formula, which denotes empty transition sequences. It is satisfied by any sequence of LTS transitions that is empty, i.e., it contains no transitions. An empty sequence has identical source and target states.

R1 "." *R2*

> is the concatenation regular formula, which denotes the concatenation of two transition sequences. It is satisfied by a sequence of LTS transitions iff this sequence consists of a first sub-sequence concatenated with a second one (the target state of the first sub-sequence being the source state of the second one), the first sub-sequence satisfying *R1* and the second one satisfying *R2*.

All data variables exported by *R1* are visible in *R2*. For each data variable *X* exported by both *R1* and *R2*, the occurrence of *X* exported by *R2* is also exported to the enclosing formula (i.e., it overrides the occurrence of *X* possibly exported by *R1*). All the other data variables (i.e., those exported by *R1* only and by *R2* only) are also exported to the enclosing formula.

R1 "|" *R2*

is the choice regular formula, which denotes the choice between two transition sequences. It is satisfied by a sequence of LTS transitions iff this sequence satisfies *R1* or it satisfies *R2*.

None of the data variables exported by *R1* (resp. by *R2*) is visible in *R2* (resp. in *R1*). All data variables exported both by *R1* and by *R2* are also exported to the enclosing formula.

R "∗"

is the repetition regular formula, which denotes the repetition of a transition sequence 0 or more times (transitive reflexive closure). It is satisfied by a sequence of LTS transitions iff this sequence consists of the concatenation of 0 or more sub-sequences, each of them satisfying *R*. Note that any empty sequence satisfies the repetition formula.

None of the data variables exported by *R* is exported to the enclosing formula, since none of these variables will be initialized when the repetition formula is satisfied by an empty sequence.

R "+"

is the strict repetition regular formula, which denotes the repetition of a transition sequence 1 or more times (transitive closure). It is satisfied by a sequence of LTS transitions iff this sequence consists of the concatenation of 1 or more sub-sequences, each of them satisfying *R*.

All data variables exported by *R* are also exported to the enclosing formula, since *R* is always satisfied by at least one sub-sequence of the current sequence.

R "?"

is the option regular formula, which denotes the optional occurrence of a transition sequence (i.e., its repetition 0 or 1 times). It is satisfied by a sequence of LTS transitions iff this sequence is empty or it satisfies *R*.

None of the data variables exported by *R* is exported to the enclosing formula, since none of these variables will be initialized when the option formula is satisfied by an empty sequence.

R "{" *E* "}"

is the counting regular formula, which denotes the repetition of a transition sequence *E* times, where *E* must be of type **nat**. It is satisfied by a sequence of LTS transitions iff this sequence is the concatenation of exactly *E* sub-sequences, each of them satisfying *R*.

None of the data variables exported by *R* is exported to the enclosing formula, since none of these variables will be initialized when the value of *E* is 0 (i.e., when the counting formula is satisfied by an empty sequence).

R "{" *E* "..." "}"
or
R "{" *E* "," "}"

is the left interval counting regular formula, which denotes the repetition of a transition sequence at least *E* times, where *E* must be of type **nat**. It is satisfied by a sequence of LTS transitions iff this sequence is the concatenation of *E* or more sub-sequences, each of them satisfying *R*.

None of the data variables exported by *R* is exported to the enclosing formula, since none of these variables will be initialized when the value of *E* is 0 (i.e., when the left interval counting formula

is satisfied by an empty sequence).

R "{" *E1* "..." *E2* "}"
or
R "{" *E1* "," *E2* "}"

is the interval counting regular formula, which denotes the repetition of a transition sequence at least *E1* times and at most *E2* times, where *E1* and *E2* must be of type **nat**. It is satisfied by a sequence of LTS transitions iff this sequence is the concatenation of *E1* or more (but not more than *E2*) sub-sequences, each of them satisfying *R*.

If the value of *E1* is equal to the value of *E2*, the regular formula is equivalent to *R* **{** *E1* **}**.

If the value of *E1* is larger than the value of *E2*, the regular formula is equivalent to **nil**.

None of the data variables exported by *R* is exported to the enclosing formula, since none of these variables will be initialized when the value of *E1* is 0 (i.e., when the interval counting formula is satisfied by an empty sequence).

"let" *X0* ":" *T0* ":=" *E0* "," ... "," *Xn* ":" *Tn* ":=" *En* "in"
  R
"end" "let"

is the variable definition regular formula, which declares and initializes data variables. It is satisfied by a sequence of LTS transitions iff this sequence satisfies the regular formula *R* in which all occurrences of variables *X0*, ..., *Xn* are substituted with the values of the expressions *E0*, ..., *En*, respectively. Each expression *Ei* must be of type *Ti* for 0 <= *i* <= *n*.

Variables *X0*, ..., *Xn* are visible in *R* but not in the enclosing formula. All data variables exported by *R* are also exported to the enclosing formula (regardless of whether or not they are identical to some of the variables *X0*, ..., *Xn*).

"if" *F0* "then"
  R0
["elsif" *F1* "then"
  R1
 ...
 "elsif" *Fn* "then"
  Rn
 "else"
  Rn+1]
"end" "if"

is the conditional regular formula, which denotes the conditional branching between several alternative transition sequences depending whether their source states satisfy or not certain state formulas. It is satisfied by a sequence of LTS transitions iff the source state of this sequence satisfies *F0* and the sequence satisfies *R0*, or the source state of this sequence satisfies *F1* (if present) and the sequence satisfies *R1*, ..., or the source state of this sequence satisfies *Fn* (if present) and the sequence satisfies *Rn*, or the sequence satisfies *Rn+1* (if present).

All state formulas *F0*, ..., *Fn* occurring as conditions of the branches must be propositionally closed (i.e., they cannot contain free occurrences of propositional variables, but may contain free occurrences of data variables) in order to ensure the syntactic monotonicity condition (see STATE FORMULAS below) for the whole *MCL* formula.

The branches "**elsif**" and "**else**" are optional; if they are all absent and the source state of the sequence does not satisfy *F0*, then the empty sequence consisting of that state satisfies the

conditional formula. In other words, the following equality holds:

"if" *F* "then" *R* "end if"

=

"if" *F* "then" *R* "else" "nil" "end" "if"

If the "**else**" clause is absent, none of the data variables exported by the regular formulas *R0*, ..., *Rn* is exported to the enclosing formula, since none of these variables will be initialized when the conditional formula is satisfied by an empty sequence. If the "**else**" clause is present, each data variable exported simultaneously by all regular formulas *R0*, ..., *Rn+1* is also exported to the enclosing formula.

"case" *E* "in"
  P0 ["where" *E0*] "->" *R0*
  ...
"|" *Pn* ["where" *En*] "->" *Rn*
"end" "case"

is the selection regular formula, which denotes the selection between several alternative transition sequences depending whether the value *v* of *E* matches or not certain patterns. It is satisfied by a sequence of LTS transitions iff this sequence matches one of the branches 0, ..., *n* of the selection, in this order. A sequence matches a branch *i* iff the following conditions hold:

- *v* matches the pattern *Pi*;

- the boolean expression *Ei* (if present) evaluates to true in a context in which all variables declared in *Pi* are replaced with the corresponding values extracted from *v*;

- the sequence satisfies *Ri* in the same context.

If the value of *E* does not match any of the patterns *P0*, ..., *Pn*, then an empty sequence satisfies the selection formula. In other words, in this case the selection formula becomes equivalent to "**nil**".

If some pattern *Pi* for some 0 <= *i* <= *n* is **any**, then each data variable exported simultaneously by all regular formulas *R0*, ..., *Ri* is also exported to the enclosing formula, since at least one of these regular formulas will be satisfied by the current sequence. If none of the patterns *Pi* is **any**, then none of the data variables exported by the regular formulas *R0*, ..., *Rn* is exported to the enclosing formula, since none of these variables will be initialized when the selection formula is satisfied by an empty sequence.

Note: For technical reasons (syntactic ambiguity concerning the "|" symbol occurring both as choice operator and as branch separator), formulas *R0*, ..., *Rn* must *not* contain the "|" operator at top-level. For instance, the following formula is illegal:

```
case E in
   P0 -> R1 | R2
|  P1 -> R3
end case
```

If regular formulas with the "|" operator at top-level are required as branches of a selection formula, then they must be surrounded by parentheses, as in the formula below:

```
case E in
   P0 -> (R1 | R2)
|  P1 -> R3
```

end case
which is legal.

"choice" *X0* ":" *T0* ["among" "{" *E01* "..." *E02* "}"]
"," ... ","
Xn ":" *Tn* ["among" "{" *En1* "..." *En2* "}"]
"in"
R
"end" "choice"

is the generalized choice regular formula, which denotes the choice among several alternative transition sequences depending whether data variables belong or not to certain domains. It is satisfied by a sequence of LTS transitions iff for each 0 <= *i* <= *n* there exists at least a value *vi* of type *Ti* in the domain delimited by the values of *Ei1* and *Ei2* (if present) such that the sequence satisfies the regular formula *R* in which all occurrences of variables *X0*, ..., *Xn* are substituted with the values *v0*, ..., *vn*, respectively. The optional expressions *Ei1* and *Ei2* must be of type *Ti* for 0 <= *i* <= *n*. Only the types **bool** and **nat** are allowed currently as *Ti*s.

All data variables exported by *R* are also exported to the enclosing formula (regardless of whether or not they are identical to some of the variables *X0*, ..., *Xn*).

"while" *F* "do"
R
"end" "while"

is the initial condition loop regular formula, which denotes the repetition of a regular sub-sequence as long as its source state satisfies a certain state formula. It is satisfied by a sequence of LTS transitions iff this sequence consists of the concatenation of 0 or more sub-sequences such that the source state of each sub-sequence satisfies *F* and each sub-sequence satisfies *R*.

None of the data variables exported by *R* is exported to the enclosing formula, since none of these variables will be initialized when the initial condition loop formula is satisfied by an empty sequence (whose source state does not satisfy *F*).

"repeat"
R
"until" *F* "end" "repeat"

is the final condition loop regular formula, which denotes the repetition of a regular sub-sequence until its target state satisfies a certain state formula. It is satisfied by a sequence of LTS transitions iff this sequence consists of the concatenation of 1 or more sub-sequences such that the target state of each sub-sequence satisfies *F* and each sub-sequence satisfies *R*.

All data variables exported by *R* are also exported to the enclosing formula, since *R* is always satisfied by at least one sub-sequence of the current sequence (the body of the final condition loop formula is repeated at least once).

"for" *X* ":" *T* "from" *E1* "to" *E2* ["step" *E3*] "do"
R
"end" "for"

is the bounded loop regular formula, which denotes the repetition of a regular sub-sequence depending on the values taken by variable *X* in an interval. It is satisfied by a sequence of LTS transitions iff this sequence consists of the concatenation of 0 or more sub-sequences, such that each sub-sequence satisfies the regular formula *R* for the current value of variable *X* (which may occur or not in *R*). The expressions *E1*, *E2*, and *E3* must be of type *T*. At the first iteration of the loop, *X* is initialized with the value of *E1*. At each sub-sequent iteration, *X* is incremented either by the value of *E3* if the optional clause "step" is present, or by 1 otherwise. The loop terminates

when *X* becomes strictly greater than the value of *E2*.

None of the data variables exported by *R* is exported to the enclosing formula, since none of these variables will be initialized when the bounded loop formula is satisfied by an empty sequence.

The type *T* of the iteration variable can be currently the **nat** type only.

"loop" ["(" *X0* ":" *T0* ":=" *E0* "," ... "," *Xn* ":" *Tn* ":=" *En* ")"]
[":" "(" *X'0* ":" *T'0* "," ... "," *X'm* ":" *T'm* ")"]
"in"
R
"end" "loop"

is the general loop regular formula, which denotes the repetition of a regular sub-sequence satisfying *R* depending on the values of the optional input parameters *X0*, ..., *Xn*. These parameters are initialized at the start of the loop with the values of expressions *E0*, ..., *En*, which must be of types *T0*, ..., *Tn*, respectively. Upon termination of the loop, the optional output parameters *X'0*, ..., *X'm* are assigned appropriate values and are exported to the enclosing formula.

An iteration of the loop is triggered when the evaluation of *R* on a sub-sequence of the current sequence leads to the evaluation of a "continue" subformula of *R* (see below), which must assign values to the input parameters *X0*, ..., *Xn*. The loop terminates when the evaluation of *R* on a subsequence of the current sequence either does not lead to the evaluation of a "continue" subformula (in this case the loop must not have output parameters), or it leads to the evaluation of an "exit" subformula of *R* (see below), which must assign values to the output parameters *X'0*, ..., *X'm*.

None of the data variables exported by *R* is exported to the enclosing formula, since none of these variables will be initialized when the general loop formula is satisfied by an empty sequence.

"continue" ["(" *E0* "," ..."," *En* ")"]

is the continuation regular formula, which denotes the general loop repetition. It can occur only in the scope of a "loop" regular formula. If present, the optional expressions *E0*, ..., *En* must be of the same types *T0*, ..., *Tn* as the input parameters *X0*, ..., *Xn* of the immediately enclosing "loop" formula. The continuation formula is always satisfied by an LTS sequence and triggers an iteration of the immediately enclosing "loop" formula, the input parameters of which are assigned the values of the expressions *E0*, ..., *En*, respectively.

"exit" ["(" *E0* "," ... "," *Em* ")"]

is the termination regular formula, which denotes the general loop termination. It can occur only in the scope of a "loop" regular formula. If present, the optional expressions *E0*, ..., *Em* must be of the same types *T'0*, ..., *T'm* as the output parameters *X'0*, ..., *X'm* of the immediately enclosing "loop" formula. The termination formula is always satisfied by an LTS sequence and triggers the termination of the immediately enclosing "loop" formula, the output parameters of which are exported to the enclosing formula after being assigned the values of the expressions *E0*, ..., *Em*, respectively.

"export" "(" *X0* ":" *T0* ":=" *E0* "," ... "," *Xn* ":" *Tn* ":=" *En* ")"

is the exporting regular formula, which assigns the values of expressions *E0*, ..., *En* to variables *X0*, ..., *Xn* and exports them to the enclosing formula. The expressions *E0*, ..., *En* must be of types *T0*, ..., *Tn*, respectively. The exporting formula is satisfied by any LTS sequence. It is an abbreviation of the following "loop" formula:

```
"loop" "(" X0 ":" T0"," ... "," Xn ":" Tn ")" "in"
   "exit" "(" E0 "," ... "," En ")"
"end" "loop"
```

Note: The "let" regular formula (see above) also assigns values to variables, but these variables are visible only in the regular subformula of the "let" formula and are not exported to the enclosing formula.

"(" *R* ")"

a sequence of LTS transitions satisfies this formula iff it satisfies *R*. Parentheses are useful for imposing an evaluation order of subformulas different from the order given by the associativity and precedence of operators.

STATE FORMULAS

A *state formula* is a logical formula built from boolean operators, modalities, fixed point operators, and data-handling constructs inspired from functional programming languages. The syntax of *MCL* state formulas is defined by the following grammar:

F **::=** *E*

| **"true"**

| **"false"**

| **"not"** *F*

| *F1* **"or"** *F2*

| *F1* **"xor"** *F2*

| *F1* **"and"** *F2*

| *F1* **"implies"** *F2*

| *F1* **"equ"** *F2*

| "<" *R* ">" *F*

| "<<" *R* ">>" *F*

| "[" *R* "]" *F*

| "[[" *R* "]]" *F*

| "<" *R* ">" "@"

| "<<" *R* ">>" "@"

| "[" *R* "]" "-|"

| "[[" *R* "]]" "-|"

| *Y* ["(" *E0* "," **...** "," *En* ")"]

| **"mu"** *Y* ["(" *X0* **":"** *T0* **":="** *E0* "," **...** ","
Xn **":"** *Tn* **":="** *En* ")"]
"." *F*

```
| "nu" Y [ "(" X0 ":" T0 ":=" E0 "," ... ","
           Xn ":" Tn ":=" En ")" ]
  "." F

| "exists" X0 ":" T0 [ "among" "{" E01 "..." E02 "}" ]
       "," ... ","
       Xn ":" Tn [ "among" "{" En1 "..." En2 "}" ]
  "." F

| "forall" X0 ":" T0 [ "among" "{" E01 "..." E02 "}" ]
       "," ... ","
       Xn ":" Tn [ "among" "{" En1 "..." En2 "}" ]
  "." F

| "let" X0 ":" T0 ":=" E0 "," ... "," Xn ":" Tn ":=" En "in"
    F
  "end" "let"

| "if" F0 "then"
    F'0
  [ "elsif" F1 "then"
    F'1
   ...
   "elsif" Fn "then"
    F'n
   "else"
    Fn+1 ]
  "end" "if"

| "case" E "in"
    P0 [ "where" E0 ] "->" F0
    ...
  "|" Pn [ "where" En ] "->" Fn
  "end" "case"

| "(" F ")"
```

Syntactically, all binary operators on state formulas are left-associative. The unary operators "**not**", "<"...">", "<<"..."<<", "["..."]", "[["..."]]", "mu", "nu", "exists", and "forall" have the highest precedence, followed by "**and**", followed by "**or**" and "**xor**", followed by "**implies**", followed by "**equ**".

The minimal and maximal fixed point operators "mu" and "nu" act as binders for the propositional variables *Y* in a way that is similar to quantifiers in first-order logic. In each meaningful "**mu** *Y* **(**...**) .** *F*" or "**nu** *Y* **(**...**) .** *F*" formula, *Y* is assumed to have free occurrences inside *F*.

State formulas must satisfy the following two syntactic conditions:

- *Syntactic monotonicity* [Koz83] means that in each fixed point formula "**mu** *Y* **(**...**) .** *F*" or "**nu** *Y* **(**...**) .** *F*", free occurrences of the propositional variable *Y* in *F* may appear only under an even number of negations and/or left-hand sides of implications.

- *Alternation-freeness* [EL86] means that each fixed point formula "**mu** *Y* **(**...**) .** *F*" cannot contain free occurrences of propositional variables *Y'* defined by "**nu**" operators, and each fixed point formula "**nu** *Y* **(**...**) .** *F*" cannot contain free occurrences of propositional variables *Y'* defined by

"**mu**" operators. When checking this condition on a formula, strong possibility (resp. necessity) modalities whose regular subformulas contain an iteration operator, and weak possibility (resp. necessity) modalities are interpreted as "hidden" minimal (resp. maximal) fixed point operators. Note that the state formulas corresponding to infinite looping and saturation operators do not satisfy the alternation-freeness condition (see REMARKS below).

A state formula defines a predicate over the states of the LTS. The semantics of *MCL* state formulas is described informally below.

E

a state of the LTS satisfies a boolean expression *E* iff *E* evaluates to **true**.

"true"

a state of the LTS always satisfies this formula.

"false"

a state of the LTS never satisfies this formula.

"not" *F*

a state of the LTS satisfies this formula iff it does not satisfy *F*.

F1 "or" *F2*

a state of the LTS satisfies this formula iff it satisfies *F1* or it satisfies *F2*.

F1 "xor" *F2*

a state of the LTS satisfies this formula iff it satisfies exactly one of *F1* and *F2*.

F1 "and" *F2*

a state of the LTS satisfies this formula iff it satisfies both *F1* and *F2*.

F1 "implies" *F2*

a state of the LTS satisfies this formula iff it does not satisfy *F1* or it satisfies *F2*.

F1 "equ" *F2*

a state of the LTS satisfies this formula iff either it satisfies both *F1* and *F2*, or neither of them.

"<" *R* ">" *F*

is the possibility modality. It is satisfied by a state of the LTS iff there is some transition sequence going out of this state that satisfies the regular formula *R* and leads to a state satisfying the state formula *F*.

The evaluation of *F* on the target state of the transition sequence is carried out in a context in which all data variables exported by *R* are initialized with the corresponding values extracted from the sequence. If there is no transition sequence satisfying *R*, then the whole possibility modality is false and *F* is not evaluated at all.

All data variables exported by *R* are visible in *F*, but none of them is exported outside the whole possibility modality.

"<<" *R* ">>" *F*

is the weak possibility modality. It is satisfied by a state of the LTS iff there is some transition sequence going out of this state that weakly satisfies the regular formula *R* and leads to a state satisfying the state formula *F*.

The evaluation of *F* on the target state of the transition sequence is carried out in a context in which all data variables exported by *R* are initialized with the corresponding values extracted from

the sequence. If there is no transition sequence weakly satisfying *R*, then the whole weak possibility modality is false and *F* is not evaluated at all.

All data variables exported by *R* are visible in *F*, but none of them is exported outside the whole weak possibility modality.

The regular formula *R* must not contain any occurrence of the **tau** action formula.

"[" *R* "]" *F*

is the necessity modality. It is satisfied by a state of the LTS iff for each transition sequence going out of this state, if this sequence satisfies the regular formula *R*, then it must lead to a state satisfying the state formula *F*.

The evaluation of *F* on the target state of each transition sequence is carried out in a context in which all data variables exported by *R* are initialized with the corresponding values extracted from that sequence. If there is no transition sequence satisfying *R*, then the whole necessity modality is true and *F* is not evaluated at all.

All data variables exported by *R* are visible in *F*, but none of them is exported outside the whole necessity modality.

"[[" *R* "]]" *F*

is the weak necessity modality. It is satisfied by a state of the LTS iff for each transition sequence going out of this state, if this sequence weakly satisfies the regular formula *R*, then it must lead to a state satisfying the state formula *F*.

The evaluation of *F* on the target state of each transition sequence is carried out in a context in which all data variables exported by *R* are initialized with the corresponding values extracted from that sequence. If there is no transition sequence weakly satisfying *R*, then the whole weak necessity modality is true and *F* is not evaluated at all.

All data variables exported by *R* are visible in *F*, but none of them is exported outside the whole weak necessity modality.

The regular formula *R* must not contain any occurrence of the **tau** action formula.

"<" *R* ">" "@"

is the infinite looping formula. It is satisfied by a state of the LTS iff there is some transition sequence going out of this state and consisting of an infinite concatenation of sub-sequences that satisfy the regular formula *R*.

None of the data variables exported by *R* is exported outside of the infinite looping formula.

"<<" *R* ">>" "@"

is the weak infinite looping formula. It is satisfied by a state of the LTS iff there is some transition sequence going out of this state and consisting of an infinite concatenation of sub-sequences that weakly satisfy the regular formula *R*.

None of the data variables exported by *R* is exported outside of the weak infinite looping formula.

The regular formula *R* must not contain any occurrence of the **tau** action formula.

"[" *R* "]" "-|"

is the finite saturation formula. It is satisfied by a state of the LTS iff for each transition sequence

going out of this state, if this sequence consists of a concatenation of sub-sequences that satisfy the regular formula *R*, then the sequence must be finite.

None of the data variables exported by *R* is exported outside of the finite saturation formula.

"[[" *R* "]]" "-|"

is the weak finite saturation formula. It is satisfied by a state of the LTS iff for each transition sequence going out of this state, if this sequence consists of a concatenation of sub-sequences that weakly satisfy the regular formula *R*, then the sequence must be finite.

None of the data variables exported by *R* is exported outside of the weak finite saturation formula.

The regular formula *R* must not contain any occurrence of the **tau** action formula.

Y ["(" *E0* "," ... "," *En* ")"]

is a call of the propositional variable *Y*. It can occur only in the scope of a fixed point formula defining *Y*. If present, the optional expressions *E0*, ..., *En* must be of the same types *T0*, ..., *Tn* as the parameters *X0*, ..., *Xn* of the corresponding fixed point formula. The propositional variable call formula is satisfied by a state of the LTS iff this state belongs to the solution *Y* of the corresponding fixed point equation, evaluated by assigning the values of the expressions *E0*, ..., *En* to the parameters *X0*, ..., *Xn*, respectively.

"mu" *Y* ["(" *X0* ":" *T0* ":=" *E0* "," ... ","
Xn ":" *Tn* ":=" *En* ")"]
"." *F*

is the parameterized minimal fixed point formula defining the propositional variable *Y*. The expressions *E0*, ..., *En* must be of types *T0*, ..., *Tn*, respectively. The formula is satisfied by a state of the LTS iff this state belongs to the minimal solution of the fixed point equation *Y* (*X0*, ..., *Xn*) = *F*, evaluated by assigning the values of the expressions *E0*, ..., *En* to the parameters *X0*, ..., *Xn*, respectively. The parameters *X0*, ..., *Xn* are visible only in *F* and are not exported outside the minimal fixed point formula.

Intuitively, a parameterized minimal fixed point formula characterizes finite subgraphs contained in the LTS. The parameters enable one to perform arbitrary computations during a forward traversal of the subgraphs.

"nu" *Y* ["(" *X0* ":" *T0* ":=" *E0* "," ... ","
Xn ":" *Tn* ":=" *En* ")"]
"." *F*

is the parameterized maximal fixed point formula defining the propositional variable *Y*. The expressions *E0*, ..., *En* must be of types *T0*, ..., *Tn*, respectively. The formula is satisfied by a state of the LTS iff this state belongs to the maximal solution of the fixed point equation *Y* (*X0*, ..., *Xn*) = *F*, evaluated by assigning the values of the expressions *E0*, ..., *En* to the parameters *X0*, ..., *Xn*, respectively. The parameters *X0*, ..., *Xn* are visible only in *F* and are not exported outside the maximal fixed point formula.

Intuitively, a parameterized maximal fixed point formula characterizes infinite subgraphs contained in the LTS. The parameters enable one to perform arbitrary computations during a forward traversal of the subgraphs.

"exists" *X0* ":" *T0* ["among" "{" *E01* "..." *E02* "}"]
"," ... ","
Xn ":" *Tn* ["among" "{" *En1* "..." *En2* "}"]
"." *F*

is the existential quantification over variables *X0*, ..., *Xn*. It is satisfied by a state of the LTS iff there exists a combination of values *v0*, ..., *vn* belonging to the types *T0*, ..., *Tn* (possibly restricted to the optional domains delimited by the values of *E01*, *E02*, ..., *En1*, *En2* if present) such that this state satisfies the state formula *F* in which all occurrences of variables *X0*, ..., *Xn* are substituted with the values *v0*, ..., *vn*, respectively. The optional expressions *Ei1* and *Ei2* must be of type *Ti* for 0 <= *i* <= *n*. Only the types **bool** and **nat** are allowed currently as *Ti*s.

The variables *X0*, ..., *Xn* are visible only in *F* and are not exported outside the existential quantification formula.

Note: The existential quantifier is not, strictly speaking, a primitive operator of *MCL*. It can be seen as an abbreviation of the disjunction operator. Assuming that the domain of variable *Xi* is **{** *vi0*, ... *vimi* **}** for 0 <= *i* <= *n*, the existential quantification formula is equivalent to the disjunction below:

F (*v00*, ..., *vn0*) or ... or *F* (*v00*, ..., *vnmn*)
or ... or
F (*v0m0*, ..., *vn0*) or ... or *F* (*v0m0*, ..., *vnmn*)

where *F* (*v0j*, ..., *vnk*) denotes the state formula *F* in which all occurrences of variables *X0*, ..., *Xn* are substituted with the values *v0j*, ..., *vnk*, respectively. In practice, the usage of the existential quantifier may yield much more concise formulations of properties than its equivalent disjunctive formulation.

"forall" *X0* ":" *T0* ["among" "{" *E01* "..." *E02* "}"]
"," ... ","
Xn ":" *Tn* ["among" "{" *En1* "..." *En2* "}"]
"." *F*

is the universal quantification over variables *X0*, ..., *Xn*. It is satisfied by a state of the LTS iff for every combination of values *v0*, ..., *vn* belonging to the types *T0*, ..., *Tn* (possibly restricted to the optional domains delimited by the values of *E01*, *E02*, ..., *En1*, *En2* if present), this state satisfies the state formula *F* in which all occurrences of variables *X0*, ..., *Xn* are substituted with the values *v0*, ..., *vn*, respectively. The optional expressions *Ei1* and *Ei2* must be of type *Ti* for 0 <= *i* <= *n*. Only the types **bool** and **nat** are allowed currently as *Ti*s.

The variables *X0*, ..., *Xn* are visible only in *F* and are not exported outside the universal quantification formula.

Note: The universal quantifier is not, strictly speaking, a primitive operator of *MCL*. It can be seen as an abbreviation of the conjunction operator. Assuming that the domain of variable *Xi* is **{** *vi0*, ... *vimi* **}** for 0 <= *i* <= *n*, the universal quantification formula is equivalent to the conjunction below:

F (*v00*, ..., *vn0*) and ... and *F* (*v00*, ..., *vnmn*)
and ... and
F (*v0m0*, ..., *vn0*) and ... and *F* (*v0m0*, ..., *vnmn*)

where *F* (*v0j*, ..., *vnk*) denotes the state formula *F* in which all occurrences of variables *X0*, ..., *Xn* are substituted with the values *v0j*, ..., *vnk*, respectively. In practice, the usage of the universal quantifier may yield much more concise formulations of properties than its equivalent conjunctive formulation.

"let" *X0* ":" *T0* ":=" *E0* "," ... "," *Xn* ":" *Tn* ":=" *En* "in"
F
"end" "let"

is the variable definition state formula, which declares and initializes data variables. It is satisfied by a state of the LTS iff this state satisfies the state formula *F* in which all occurrences of variables *X0*, ..., *Xn* are substituted with the values of the expressions *E0*, ..., *En*, respectively. Each expression *Ei* must be of type *Ti* for 0 <= *i* <= *n*.

The variables *X0*, ..., *Xn* are visible only in *F* and are not exported outside the variable definition formula.

"if" *F0* "then"
F'0
["elsif" *F1* "then"
F'1
...
"elsif" *Fn* "then"
F'n
"else"
Fn+1]
"end" "if"

is the conditional state formula, which denotes the branching according to certain state formulas. It is satisfied by a state of the LTS iff either this state satisfies *F0* and it also satisfies *F'0*, or this state satisfies *F1* (if present) and it also satisfies *F'1*, ..., or this state satisfies *Fn* (if present) and it also satisfies *F'n*, or this state satisfies *Fn+1* (if present).

All state formulas *F0*, ..., *Fn* occurring as conditions of the branches must be propositionally closed (i.e., they cannot contain free occurrences of propositional variables, but may contain free occurrences of data variables) in order to ensure the syntactic monotonicity condition for the whole *MCL* formula.

The branches "**elsif**" and "**else**" are optional; if they are all absent and the state does not satisfy *F0*, then this state satisfies the conditional formula. In other words, the following equality holds:

"if" *F0* "then" *F'0* "end if"

=

"if" *F0* "then" *F'0* "else" "true" "end" "if"

"case" *E* "in"
P0 ["where" *E0*] "->" *F0*
...
"|" *Pn* ["where" *En*] "->" *Fn*
"end" "case"

is the selection state formula, which denotes the selection between several alternatives depending on whether the value *v* of *E* matches or not certain patterns. It is satisfied by a state of the LTS iff this state matches one of the branches 0, ..., *n* of the selection, in this order. A state matches a branch *i* iff the following conditions hold:

- *v* matches the pattern *Pi*;

- the boolean expression *Ei* (if present) evaluates to true in a context in which all variables declared in *Pi* are replaced with the corresponding values extracted from *v*;

- the state satisfies *Fi* in the same context.

If the value of *E* does not match any of the patterns *P0*, ..., *Pn*, then the selection formula is true.

"(" *F* ")"

a state of the LTS satisfies this formula iff it satisfies *F*. Parentheses are useful for imposing an evaluation order of subformulas different from the order given by the associativity and precedence of operators.

An LTS satisfies a state formula *F* iff its initial state *s0* satisfies *F*.

REMARKS

When writing complex formulas containing many operators (especially when mixing regular and boolean operators), it is safer to use parentheses to enclose subformulas whenever being in doubt about the relative precedence of the operators. Otherwise, the tool may parse and evaluate the formulas in a way different from the user's intentions, leading to erroneous results that may be quite difficult to track down.

Not all operators defined above are primitive constructs of the logic. The boolean operators "**`false`**", "**`and`**", "**`implies`**", and "**`equ`**" can be expressed in terms of "**`true`**", "**`or`**", and "**`not`**" in the usual way. The diamond and box modalities are dual:

[*R* **]** *F* **= not <** *R* **>** **not** *F*
[[*R* **]]** *F* **= not <<** *R* **>> not** *F*

The same holds for minimal and maximal fixed point operators (only parameterless versions are illustrated below):

nu *Y* **.** *F* **= not mu** *Y* **.** **not** *F* **(not** *Y***)**

where *F* **(not** *Y***)** denotes the syntactic substitution of *Y* by **not** *Y* in *F*.

The infinite looping operator and the finite saturation operator are opposites:

< *R* **> @** **= not [** *R* **]** **-|**
<< *R* **>> @ = not [[** *R* **]] -|**

The modalities containing regular formulas can be translated in terms of boolean operators, fixed point operators, and modalities containing only action formulas (see [MS03,MT08] for details).

The infinite looping and finite saturation operators correspond to fixed point formulas belonging to the mu-calculus fragment of alternation depth two [EL86]. In practice, this means that one can write formulas "**<** *R* **> @**" in which the regular subformula *R* contains iteration operators. This feature is supported by **evaluator4**, but was not available in **evaluator3**, which accepted only "**<** *R* **> @**" formulas containing iteration-free regular subformulas *R*.

evaluator4 handles the equivalent fixed point formulations of infinite looping and saturation operators, which can be used directly instead of these operators:

< *R* **> @** **= nu** *Y* **.** **<** *R* **>** *Y*
<< *R* **>> @** **= nu** *Y* **.** **<<** *R* **>>** *Y*
[*R* **]** **-|** **= mu** *Y* **.** **[** *R* **]** *Y*
[[*R* **]] -| = mu** *Y* **.** **[[** *R* **]]** *Y*

Moreover, in the fixed point formulas equivalent to infinite looping and saturation operators, **evaluator4** accepts propositional variables *Y* parameterized by data values (see FAIRNESS PROPERTIES below).

For efficiency reasons, when using fixed point operators, it is recommended to put the recursive call of the propositional variable at the rightmost place in the formula (as in all fixed point formulas shown above). This reduces both the evaluation time and the size of the diagnostic generated for the formula.

A fixed point formula "**mu**" *X* "**.**" *F* or "**nu**" *X* "**.**" *F* is *unguarded* [Koz83] if *F* contains at least one free occurrence of *X* which is not preceded (not necessarily immediately) by a modality. The evaluation of an unguarded formula on an LTS may yield a BES with cyclic dependencies between variables even if the LTS is acyclic.

Note that a state formula containing regular modalities with nested star operators may yield after translation an unguarded mu-calculus formula. For example, in the following formula:

```
< A1** . A2 > true =
       mu X1 . (< A2 > true or mu X2 . (X1 or < A1 > X2)
```

the free occurrence of *X1* is not preceded by any modality, and hence the formula is unguarded.

Unguarded occurrences of propositional variables can always be eliminated from a mu-calculus formula, at the price of an increase in size [Koz83,Mat02].

EXAMPLES OF TEMPORAL PROPERTIES

MCL allows to express concisely various interesting properties. The most useful classes of temporal properties are illustrated below.

SAFETY PROPERTIES

Informally, a *safety* property specifies that "something bad never happens." Typical safety properties are those forbidding "bad" execution sequences in the LTS. These properties can be naturally expressed using box modalities containing regular formulas. For instance, mutual exclusion can be characterized by the following formula:

```
[ true* . "ENTER !1" . (not "LEAVE !1")* . "ENTER !2" ] false
```

which states that every time process 1 enters its critical section (action **"ENTER !1"**), it is impossible that process 2 also enters its critical section (action **"ENTER !2"**) before process 1 has left its critical section (action **"LEAVE !1"**).

Note that this formula does not make any assumption about the fact that a process enters/leaves several times its critical section, i.e., the formula does not forbid sequences of the form **"ENTER !1 . ENTER !1 . LEAVE !1 . LEAVE !1"**.

The above formula can be made parametric w.r.t. the number of processes in the system, by using action predicates equipped with data variables:

```
[ true* . { ENTER ?m:Nat } . (not { LEAVE !m })* .
     { ENTER ?n:Nat where m <> n } ] false
```

where the values of **m**, **n** captured by the action predicates **{** *ENTER* ?*m*:Nat **}** and **{** *ENTER* ?*n*:Nat **}** are propagated to the enclosing formula in order to ensure that a process *n* different from *m* cannot enter its critical section before process *m* has left it (action predicate **{** *LEAVE* !*m* **}**).

Regular formulas equipped with counters provide a useful means for describing safety properties depending on data. The formula below expresses (part of) the safety of a *n*-place buffer:

```
[ (INPUT . (not OUTPUT)*) { n + 1 } ] false
```

by forbidding the existence of a sequence containing more than *n* insertions of elements in the buffer (action *INPUT*) without any deletions of elements in between (action *OUTPUT*).

A more precise formulation of the above property can be obtained by using a fixed point operator parameterized by a counter *c*, which stores the number of elements (initially 0) currently present in the buffer:

```
nu Buffer (c:Nat := 0) . (
       [ INPUT ] ((c < n) and Buffer (c + 1))
       and
       [ OUTPUT ] ((c > 0) and Buffer (c - 1))
       and
       [ not (INPUT or OUTPUT) ] Buffer (c)
   )
```

The number of elements *c* in the buffer is equal to the difference between the number of elements inserted and deleted from the buffer, and for a *n*-place buffer *c* must belong to 0..*n*.

Other typical safety properties are the *invariants*, expressing that every state of the LTS satisfies some "good" property. For example, deadlock freedom can be expressed by the formula below:

```
[ true* ] < true > true
```

stating that every state has at least one successor. Alternately, this formula may be expressed directly using a fixed point operator:

```
nu X . (< true > true and [ true ] X)
```

but less concisely than by using a regular formula.

The "**natset**" type is useful for expressing the occurrence of a set of actions in any order. For instance, the fact that natural numbers inserted in a bag (initially empty) can be retrieved in any order can be expressed by the *MCL* formula below:

```
nu Bag (b:NatSet := empty) . (
       [ { PUT ?n:nat } ] Bag (insert (n, b))
       and
       [ { GET ?n:nat } ] ((n isin b) and Bag (remove (n, b)))
       and
       [ not ({ PUT ... } or { GET ... }) ] Bag (b)
   )
```

Here the action predicates **{** *PUT* ?*n*:Nat **}** and **{** *GET* ?*n*:Nat **}** denote the insertion and retrieval of a natural number into/from the bag, respectively.

LIVENESS PROPERTIES

Informally, a *liveness* property specifies that "something good eventually happens." Typical liveness properties are *potentiality* assertions (i.e., expressing the reachability on a sequence) and *inevitability* assertions (i.e., expressing the reachability on all sequences).

Potentiality assertions can be directly expressed using diamond modalities containing regular formulas. For instance, the following formula:

```
< true* . { PUT ?n:Nat } . true* . { GET !n } > true
```

states that there exists a sequence that passes (after 0 or more transitions) through a *PUT n* action for some natural number *n*, and then leads (after 0 or more transitions) to a *GET n* action.

Regular formulas allow to express succinctly complex potentiality assertions, such as the formula below:

```
< true* . SEND . (true* . RETRY) { 0 ... max } .
    true* . RECV > true
```

stating that there exists a sequence leading (after 0 or more transitions) to a *SEND* action, possibly followed by a sequence of at most *max RETRY* actions (possibly separated by other actions) and leading (after 0 or more transitions) to a *RECV* action.

Inevitability assertions can be expressed using fixed point operators. For instance, the following formula:

```
mu X . (< true > true and [ not START ] X)
```

states that all transition sequences starting at the current state lead to *START* actions after a finite number of steps.

Similarly, temporal properties containing both safety and liveness aspects can be expressed by combining box modalities and inevitability operators. For example, the *response* property stating that every emission of a message must be inevitably followed in the future by the reception of the same message can be expressed by the *MCL* formula below:

```
[ true* . { SEND ?n:Nat } ]
      mu X . (< true > true and [ not { RECV !n } ] X)
```

Note how variable *n* is assigned in the box modality by capturing the value of a message sent (action predicate **{** *SEND* ?*n*:Nat **}**) and is used later in the body of the fixed point formula (action predicate **{** *RECV* !*n* **}**).

FAIRNESS PROPERTIES

These are similar to liveness properties, except that they express reachability of actions by considering only *fair* execution sequences. One simple notion of fairness that can be easily encoded in *MCL* is the "fair reachability of predicates" defined by Queille and Sifakis [QS83]: a sequence is fair iff it does not infinitely often enable the reachability of a certain state without infinitely often reaching it. For instance, the following formula specifies that after every message emission (action SEND), all fair execution sequences will lead to the reception of the message (action RECV) after a finite number of steps:

```
[ true* . SEND . (not RECV)* ] < (not RECV)* . RECV > true
```

Intuitively, the formula above considers the sequences following the *SEND* action by "skipping" the circuits of the LTS that do not contain *RECV* actions: it states that from every state of such a circuit, there is still a finite sequence leading to a *RECV* action.

More elaborated forms of fairness can be expressed by specifying the absence of *unfair* execution sequences, which can be characterized using the infinite looping operator. For example, the formula *MCL* below expresses that after process *i* has requested access to a resource, it cannot be indefinitely preempted by another process *j*:

```
[ true* . { REQUEST ?i:Nat } ]
    not < (not { GRANT !i })* . { REQUEST ?j:Nat where j <> i } .
          (not { GRANT !i })* . { GRANT !j } > @
```

This formula can be expressed more concisely by propagating the negation in front of the infinite looping operator and using the saturation operator:

```
[ true* . { REQUEST ?i:Nat } ]
    [ (not { GRANT !i })* . { REQUEST ?j:Nat where j <> i } .
      (not { GRANT !i })* . { GRANT !j } ] -|
```

The existence of complex cycles can be specified using the fixed point formulation of the infinite looping operator, in which the propositional variable has data parameters. The formula below expresses the existence of a cycle on which the pairs emission-reception of messages **n=0..4** occur in this order:

```
nu Y (n:Nat := 0) .
    < true* . { PUT !n } . true* . { GET !n } >
      Y ((n + 1) % 5)
```

Other, more elaborated examples of fairness properties can be found in [MT08,MS10].

ACTION PREDICATES

The use of action formulas (and, in particular, of regexps) may be of considerable help when dealing with

actions having the same gate but different values in the offers. For instance, the following formula:

< true* . 'SEND !1.*' and not 'SEND !1.*!2.*' > true

states the potential reachability of an action having the gate *SEND* followed by the value 1, possibly followed by other values different from 2.

Moreover, action formulas combined with modalities allow to express invariants over actions (i.e., action formulas that must be satisfied by all transition labels of the LTS). For instance, the following formula:

[true* . { RECV ?src:Nat ?dest:Nat where src <> dest }] false

states that all message receptions have different source and destination fields. Another way of formulating this property is by using regexps on character strings:

```
[ true* .
  not ('RECV !.* !.*' and 'RECV !\(.*\) !\1')
] false
```

Note the use of the UNIX regular expression construct '\(\)' allowing to match a portion of a string and to reuse it later in the same regexp.

MACROS AND LIBRARIES

evaluator4 allows to define and use macros for temporal operators parameterized by action and/or state formulas. This feature is particularly useful for constructing reusable libraries encoding various temporal operators of other logics translatable in regular alternation-free mu-calculus (like CTL and ACTL). The *macro-definitions* have the following syntax:

```
"macro" M "(" P1"," ..."," Pn ")" "="
        <text>
   "end_macro"
```

The above construct defines a macro *M* having the parameters *P1*, ..., *Pn* and the body *<text>*, which is a string of alpha-numeric characters (normally) containing occurrences of the parameters *P1*, ..., *Pn*. For example, the following macro-definition:

```
macro EU_A (F1, A, F2) =
       mu X . ((F2) or ((F1) and < A > X))
   end_macro
```

encodes the "Exists Until" operator of ACTL, which states that there exists a sequence of transitions leading to a state satisfying F2 such that all intermediate states satisfy F1 and all intermediate labels satisfy A.

The calls of a macro *M* have the following form:

```
M "(" <text1>"," ..."," <textn> ")"
```

where the arguments *<text1>*, ..., *<textn>* are strings. The result of the call is the body <text> of the macro *M* in which all occurrences of the parameters *Pi* have been syntactically substituted with the arguments *<texti>*, for all i between 1 and n. For example, the following call:

```
EU_A (true, not "SEND", < "RECV" > true)
```

expands into the formula below:

```
mu X . ((< "RECV" > true) or ((true) and < not "SEND" > X))
```

A macro is visible from the point of its definition until the end of the formula. Macros may be overloaded: several macros with the same name, but different arities, may be defined in the same scope.

Various macro-definitions (typically encoding the operators of some particular temporal logic) can be grouped into files called *libraries*. These files may be included in the source program using the following command:

"library"
<*file0***.mcl>","** ...**","** **<***filen***.mcl>**
"end_library"

At the compilation of the program, the above construct is syntactically replaced with the contents of the files <*file0*.**mcl**>, ..., <*filen*.**mcl**>, placed one after the other in this order. For example, the following command:

library *actl*.**mcl end_library**

is syntactically replaced with the contents of the file *actl*.**mcl,** which implements the ACTL operators.

The included files are searched first in the current directory, then in the directory referenced by $CADP/src/xtl. Multiple inclusions of the same file are silently discarded.

EXPRESSIVENESS

MCL enables direct and succinct encodings of "pure" branching-time logics like *CTL* (Computation Tree Logic) [CES86] or *ACTL* (Action-based CTL) [DV90], as well as of regular logics like *PDL* (Propositional Dynamic Logic) [FL79] or *PDL-delta* [Str82].

The infinite looping operator, whenever it is applied to a regular subformula containing iteration operators, belongs to the mu-calculus fragment of alternation depth two [EL86]. It is able to express the existence of complex cycles (containing regular sub-sequences) in the LTS, which cannot be expressed using the other operators of *MCL* because they belong to the alternation-free fragment of the modal mu-calculus. In particular, the infinite looping operator can express the existence of accepting cycles in Büchi automata, which underlies the classical verification procedure for *LTL* (Linear Time Logic) [CGP00].

Therefore, *MCL* strictly subsumes both *CTL* and *LTL*, since these two logics are not comparable w.r.t. their expressive power (each of them is able to express properties that the other one cannot). *MCL* also syntactically subsumes *PDL-delta*, which was shown to be more expressive than *CTL*∗ [Wol82].

When dealing with finite state LTS models, the presence of data-handling constructs does not, strictly speaking, increase the expressiveness of *MCL* because one can instantiate all parameters present in an *MCL* formula based on the finite set of values contained in the transition labels of the LTS. However, in practice the data-handling constructs lead to significant simplifications and reductions in size of the formulas, thus facilitating the specification activity and reducing the risk of errors.

MODEL CHECKING COMPLEXITY

The dataless part of *MCL* has an efficient on-the-fly model checking algorithm, with a space and time complexity linear in the size of the formula (number of operators) and the size of the LTS model (number of states and transitions). Despite the fact that it belongs to the mu-calculus fragment of alternation depth two (which has theoretically a quadratic-time model checking complexity [EL86]), the infinite looping operator can be evaluated in linear-time using the algorithm proposed in [MT08]. The evaluation of *CTL* and *PDL-delta* operators, which cover the quasi-totality of practical needs, stores in memory only the states, and not the transitions of the LTS.

Note: The linear-time model checking complexity obtained for *PDL-delta* does not imply a similar result for *LTL* or *CTL*∗, since the translations of these logics in *PDL-delta* are not guaranteed to be succinct.

The evaluation of fixed points having parameters of infinite types (e.g., **nat**, **string**, etc.) may diverge when the number of fixed point variable instances is unbounded. Therefore, parameterized fixed points should be used with the same care as recursive functions in programming languages (note however that cycles *Y* (*v0*, ..., *vn*) -> ... -> *Y* (*v0*, ..., *vn*) do no harm, since BES resolution algorithms can handle cyclic dependencies between variables). The evaluation of all extended regular operators involving counters is guaranteed to converge, because it always creates a finite number of fixed point variable instances, bounded by the values of counters and/or the number of LTS states.

BIBLIOGRAPHY

[CES86]
E. M. Clarke, E. A. Emerson, and A. P. Sistla. "Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications." ACM Transactions on Programming Languages and Systems, v. 8, no. 2, p. 244-263, 1986.

[CGP00]
E. M. Clarke, O. Grumberg, and D. Peled. "Model Checking." MIT Press, 2000.

[DV90] R. De Nicola and F. W. Vaandrager. "Action versus State based Logics for Transition Systems." Proceedings Ecole de Printemps on Semantics of Concurrency, LNCS v. 469, p. 407-419, 1990.

[EL86] E. A. Emerson and C-L. Lei. "Efficient Model Checking in Fragments of the Propositional Mu-Calculus." Proceedings of the 1st LICS, p. 267-278, 1986.

[FL79] M. J. Fischer and R. E. Ladner. "Propositional Dynamic Logic of Regular Programs." Journal of Computer and System Sciences, no. 18, p. 194-211, 1979.

[Gar89] H. Garavel. Chapter 9 of "Compilation et verification de programmes LOTOS." PhD thesis, Universite Joseph-Fourier Grenoble, 1989. Available from http://cadp.inria.fr/publications/Garavel-89-b.html

[Koz83] D. Kozen. "Results on the Propositional Mu-Calculus." Theoretical Computer Science, v. 27, p. 333-354, 1983.

[Mat98a]
R. Mateescu. "Verification des proprietes temporelles des programmes paralleles." PhD Thesis, Institut National Polytechnique de Grenoble, April 1998. Available from http://cadp.inria.fr/publications/Mateescu-98-a.html

[Mat98b]
R. Mateescu. "Local Model-Checking of an Alternation-Free Value-Based Modal Mu-Calculus." Proceedings of the 2nd International Workshop on Verification, Model Checking and Abstract Interpretation VMCAI'98, 1998. Available from http://cadp.inria.fr/publications/Mateescu-98-b.html

[Mat02] R. Mateescu. "Local Model-Checking of Modal Mu-Calculus on Acyclic Labeled Transition Systems". Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'02, LNCS v. 2280, p. 281-295, 2002. Full version available as INRIA Research Report RR-4430. Available from http://cadp.inria.fr/publications/Mateescu-02.html

[Mat06] R. Mateescu. "CAESAR_SOLVE: A Generic Library for On-the-Fly Resolution of Alternation-Free Boolean Equation Systems." Springer International Journal on Software Tools for Technology Transfer (STTT), v. 8, no. 1, p. 37-56, 2006. Full version available as INRIA Research Report RR-5948. Available from http://cadp.inria.fr/publications/Mateescu-06-a.html

[MS03] R. Mateescu and M. Sighireanu. "Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus." Science of Computer Programming, v. 46, no. 3, p. 255-281, 2003. Available

from http://cadp.inria.fr/publications/Mateescu-Sighireanu-03.html

[MS10] R. Mateescu and W. Serwe. "A Study of Shared-Memory Mutual Exclusion Protocols using CADP." Proceedings of the 15th International Workshop on Formal Methods for Industrial Critical Systems FMICS'10, LNCS v. 6371, p. 180-197, 2010. Available from http://cadp.inria.fr/publications/Mateescu-Serwe-10.html

[MT08] R. Mateescu and D. Thivolle. "A Model Checking Language for Concurrent Value-Passing Systems." Proceedings of the 15th International Symposium on Formal Methods FM'08, LNCS v. 5014, p. 148-164, 2008. Available from http://cadp.inria.fr/publications/Mateescu-Thivolle-08.html

[QS83] J-P. Queille and J. Sifakis. "Fairness and Related Properties in Transition Systems - A Temporal Logic to Deal with Fairness." Acta Informatica, v. 19, p. 195-220, 1983.

[Str82] R. S. Streett. "Propositional Dynamic Logic of Looping and Converse." Information and Control, v. 54, p. 121-141, 1982.

[Wol82] P. Wolper. "A Translation from Full Branching Time Temporal Logic to One Letter Propositional Dynamic Logic with Looping." Unpublished manuscript, 1982.

SEE ALSO

evaluator(LOCAL), **evaluator3**(LOCAL), **evaluator4**(LOCAL), **mcl**(LOCAL), **mcl3**(LOCAL), **regexp**(LOCAL)

Additional information is available from the CADP Web page located at http://cadp.inria.fr

Directives for installation are given in files **$CADP/INSTALLATION_***.

Recent changes and improvements to this software are reported and commented in file **$CADP/HISTORY**.

BUGS

Please report bugs to Radu.Mateescu@inria.fr