NAME

mcl, MCL – Model Checking Language version 3 (regular alternation-free mu-calculus)

DESCRIPTION

This manual page presents the version 3 of *MCL* (*Model Checking Language*), which is the temporal logic accepted as input by **evaluator3**(LOCAL).

This temporal logic is also known as "regular alternation-free mu-calculus". A description of the regular alternation-free mu-calculus can be found in the articles [MS03] and [Mat06], which also describe the verification methods implemented in versions 3.0 and 3.5 of EVALUATOR, respectively.

The regular alternation-free mu-calculus is an extension of the alternation-free fragment of the modal mu-calculus [Koz83, EL86] with action predicates and regular expressions over action sequences. In this setting, action labels are merely handled as character strings.

Note: There exists an extended version of this logic, able to express temporal properties involving data values; see the **mcl4**(LOCAL) manual page for details. This extended version is supported by **evaluator4**(LOCAL) but not by **evaluator3**(LOCAL).

Regular alternation-free mu-calculus allows direct encodings of "pure" branching-time logics like *CTL* [CES86] or *ACTL* [DV90], as well as of regular logics like *PDL* [FL79] or *PDL-delta* [Str82]. Moreover, it has an efficient model checking algorithm, with a linear-time complexity in the size of the formula (number of operators) and the size of the LTS model (number of states and transitions). The logic is built from three types of formulas, indicated in the table below.

Symbol	Description
A	action formula
R	regular formula
F	state formula

The BNF syntax and the informal semantics of these formulas are defined below. In the grammar, terminal symbols are written between double quotes. The axiom of the grammar is the **F** symbol.

Identifiers are built from letters, digits, and underscores (beginning with a letter or an underscore). Keywords must be written in lowercase. Comments are enclosed between '**(***' and '***)**'. Nested comments are not allowed. **evaluator3** is case-sensitive.

The formulas 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 noted *s1-a->s2*, indicates that the program from which the LTS has been generated can move from state *s1* to state *s2* by performing action *a*.

ACTION FORMULAS

An *action formula* is a logical formula built from basic action predicates and boolean connectives, according to the grammar below:

```
A     ::=  string
         |  regexp
         |  "true"
         |  "false"
         |  "not" A
```

```
|   A1 "or" A2
|   A1 "and" A2
|   A1 "implies" A2
|   A1 "equ" A2
```

A *string* is a sequence of zero or more characters, enclosed between double quotes ('**"**'), which denotes a label 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 '**#**'.

```
string ::= "(any char but end-of-line)*"
    | string1 "#" string2
```

A transition label of the LTS satisfies a *string* iff it is identical to the corresponding character string (obtained after concatenation whenever needed).

A *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 labels of the LTS. Regexp's can be concatenated using the binary operator '**#**'. Strings can be concatenated to regexp's, in which case they are implicitly converted into regexp's.

```
regexp ::= 'UNIX_regular_expression'
    | regexp1 "#" regexp2
    | string1 "#" regexp2
    | regexp1 "#" string2
```

A label of the LTS satisfies a *regexp* if it matches the corresponding *UNIX_regular_expression* (obtained after concatenation whenever needed).

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

The boolean operators have the usual semantics: a label of the LTS always satisfies "**true**"; it never satisfies "**false**"; it satisfies "**not** *A*" iff it does not satisfy *A*; it satisfies "*A1* **or** *A2*" iff it satisfies *A1* or it satisfies *A2*; it satisfies "*A1* **and** *A2*" iff it satisfies both *A1* and *A2*; it satisfies "*A1* **implies** *A2*" iff it does not satisfy *A1* or it satisfies *A2*; it satisfies "*A1* **equ** *A2*" iff either it satisfies both *A1* and *A2*, or none of them.

REGULAR FORMULAS

A *regular formula* is a logical formula built from action formulas and the traditional regular expression operators, according to the grammar below:

```
R ::=  A
    |  "nil"
    |  R1 "." R2
    |  R1 "|" R2
    |  R "?"
    |  R "*"
    |  R "+"
```

where "**nil**" is the empty operator, "**.**" is the concatenation operator, "**|**" is the choice operator, "**?**" is the option operator, "*" is the transitive and reflexive closure operator, and "**+**" is the transitive closure operator.

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

Note: In early versions of **evaluator3**(LOCAL), the "**|**" operator had a higher precedence than "**.**". To ensure that "old" MCL version 3 regular formulas are interpreted by the current version of **evaluator3**(LOCAL) according to their original intended meaning, it is recommended to add parentheses at appropriate places. For example, an "old" MCL version 3 regular formula "R1 | R2 . R3" should be rewritten as

"(R1 | R2) . R3" to maintain its original meaning, otherwise the current version of **evaluator3**(LOCAL) would parse it as "R1 | (R2 . R3)".

A regular formula *R* denotes a sequence of (consecutive) LTS transitions such that the word obtained by concatenating their labels belongs to the regular language defined by *R*.

The regular operators have the following semantics: a sequence of LTS transitions satisfies *A* iff it has the form *s1-a->s2*, where the label *a* satisfies the formula *A*; it satisfies "**nil**" iff it is empty (i.e., it contains no transition); it satisfies "*R1* **.** *R2*" iff it is the concatenation of two sequences satisfying *R1* and *R2*, respectively; it satisfies "*R1* **|** *R2*" iff it satisfies *R1* or it satisfies *R2*; it satisfies "*R* **?**" iff it is either empty, or it satisfies *R*; it satisfies "*R* ∗" iff it is the concatenation of zero or more sequences satisfying *R*; it satisfies "*R* **+**" iff it is the concatenation of one or more sequences satisfying *R*.

STATE FORMULAS

A *state formula* is a logical formula built from boolean, modal, and fixed point operators, according to the grammar below:

```
F     ::=  "true"
         |  "false"
         |  "not" F
         |  F1 "or" F2
         |  F1 "and" F2
         |  F1 "implies" F2
         |  F1 "equ" F2
         |  "<" R ">" F
         |  "[" R "]" F
         |  "<" R ">" "@"
         |  "[" R "]" "-|"
         |  X
         |  "mu" X "." F
         |  "nu" X "." F
```

where "**<** *R* **>** *F*" and "**[** *R* **]** *F*" are the possibility and necessity modal operators, "**<** *R* **>** **@**" is the infinite looping operator, "**[** *R* **]** **-|**" is the saturation operator, "**mu** *X* **.** *F*" and "**nu** *X* **.** *F*" are the minimal and maximal fixed point operators, and *X* is a *propositional variable*.

Syntactically, all binary operators on state formulas are left-associative. The "**not**", "**< >**", "**[ ]**", "**mu**", and "**nu**" operators have the highest precedence, followed by "**and**", followed by "**or**", followed by "**implies**", followed by "**equ**". The fixed point operators act as binders for the variables *X* in a way similar to quantifiers in first-order logic. In each meaningful "**mu** *X* **.** *F*" or "**nu** *X* **.** *F*" formula, *X* is supposed to have free occurrences inside *F*. State formulas are assumed to be *syntactically monotonic* (i.e., in each fixed point formula "**mu** *X* **.** *F*" or "**nu** *X* **.** *F*", free occurrences of *X* in *F* may appear only under an even number of negations and/or left-hand sides of implications) and *alternation-free* (i.e., without mutually recursive minimal and maximal fixed point variables).

The boolean operators have the usual semantics: a state of the LTS always satisfies "**true**"; it never satisfies "**false**"; it satisfies "**not** *F*" iff it does not satisfy *F*; it satisfies "*F1* **or** *F2*" iff it satisfies *F1* or it satisfies *F2*; it satisfies "*F1* **and** *F2*" iff it satisfies both *F1* and *F2*; it satisfies "*F1* **implies** *F2*" iff it does not satisfy *F1* or it satisfies *F2*; it satisfies "*F1* **equ** *F2*" iff either it satisfies both *F1* and *F2*, or none of them.

The modal operators have the following semantics: a state of the LTS satisfies "**<** *R* **>** *F*" iff there is (at least) one transition sequence starting at the state, satisfying *R*, and leading to a state satisfying *F*; it satisfies "**[** *R* **]** *F*" iff all transition sequences starting at the state and satisfying *R* are leading to states satisfying *F*.

The infinite looping and saturation operators have the following semantics: a state of the LTS satisfies "**<** *R* **> @**" iff there is a transition sequence starting at the state and consisting of an infinite concatenation of sequences satisfying *R*; it satisfies "**[** *R* **] -|**" iff all transition sequences starting at the state and consisting of a concatenation of sequences satisfying *R* are finite.

The fixed point operators have the following semantics: a state satisfies "**mu** *X* **.** *F*" iff it belongs to the minimal solution of the fixed point equation $X = F(X)$, and it satisfies "**nu** *X* **.** *F*" iff it belongs to the maximal solution of the same equation, where the propositional variable *X* denotes a set of LTS states. Intuitively, minimal (resp. maximal) fixed point operators allow to characterize finite (resp. infinite) tree-like patterns in the LTS.

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

Note: When writing complex formulas containing many operators (especially when mixing regular and boolean operators), it is safer to use parenthesis to enclose subformulas whenever being in doubt about the relative priorities 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.

Note: 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*

The same holds for minimal and maximal fixed point operators:

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

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

The saturation operator is the negation of the infinite looping operator:

[*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, by recursively applying the identities below:

< **nil** **>** *F* **=** **<** **false*** **>** *F*
< *R1* **.** *R2* **>** *F* **=** **<** *R1* **>** **<** *R2* **>** *F*
< *R1* **|** *R2* **>** *F* **=** **<** *R1* **>** *F* **or** **<** *R2* **>** *F*
< *R***?** **>** *F* **=** **<** **nil** **|** *R* **>** *F*
< *R** **>** *F* **= mu** *X* **. (**F **or** **<** *R* **>** *X***)**
< *R***+** **>** *F* **=** **<** *R* **.** *R** **>** *F*

where *X* is a "fresh" propositional variable (the corresponding identities for box modalities are obtained by duality).

The infinite looping operator is equivalent to the maximal fixed point formula below:

< *R* **> @ = nu** *X* **. <** *R* **>** *X*

where *X* is a "fresh" propositional variable.

Note: Early versions of **evaluator3**(LOCAL) accepted only alternation-free formulas, meaning that infinite looping operators "**<** *R* **>** **@**" were not allowed to contain "∗" or "**+**" operators in their regular formulas *R*. The current version of **evaluator3**(LOCAL) accepts regular formulas with "∗" or "**+**" in infinite looping operators, which are now able to characterize complex cycles in the LTS (e.g., generalized Buchi accepting cycles). An example of formula accepted by the current version of **evaluator3**(LOCAL) but not expressible in alternation-free mu-calculus is the following:

```
< true* . "A" > @
```

This formula is equivalent (by applying the identities above) to a fixed point formula of alternation depth 2:

```
nu X . mu Y . (< "A" > X or < true > Y)
```

Although the mu-calculus fragment of alternation depth 2 has in general a quadratic-time model checking complexity in the size of the LTS, the alternation depth 2 formulas resulting from the translation of infinite looping operators "**<** *R* **>** **@**" containing "∗" or "**+**" operators in their regular formulas *R* have a linear-time model checking complexity in the size of the LTS [MT08].

Note: Early versions of **evaluator3**(LOCAL) accepted the syntax "**@** **(** *R* **)**" for the infinite looping operator. This syntax is now obsolete, but still accepted by **evaluator3**(LOCAL) for backward compatibility. It is recommended to use the new syntax "**<** *R* **>** **@**", which is closer to the syntax of possibility modalities and reflects more intuitively the existence of an infinite sequence, terminated by a loop ("@") in a finite state LTS.

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.

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

The regular alternation-free mu-calculus allows to express concisely various interesting properties. The most useful classes of temporal properties are illustrated below.

SAFETY PROPERTIES

Informally, a safety property expresses 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* . "OPEN !1" . (not "CLOSE !1")* . "OPEN !2" ] false
```

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

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.

LIVENESS PROPERTIES

Informally, a liveness property expresses 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* . "GET !0" > true
```

states that there exists a sequence leading to a "GET !0" action after performing zero or more transitions. Regular formulas allow to express succinctly complex potentiality assertions, such as the formula below:

```
< true* . "SEND" . (true* . "ERROR")* . true* . "RECV" > true
```

stating that there exists a sequence leading (after zero or more transitions) to a "SEND" action, possibly followed by a sequence of "ERROR" actions (possibly separated by other actions) and leading (after zero 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.

FAIRNESS PROPERTIES

These are similar to liveness properties, except that they express reachability of actions by considering only *fair* execution sequences. One notion of fairness that can be easily encoded in the logic 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 expresses 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 cycles of the LTS that do not contain "RECV" actions: it states that from every state of such a cycle, there is still a finite sequence leading to a "RECV" action.

ACTION PREDICATES

The use of action formulas (and, in particular, of regexp's) may be of considerable help when dealing with LOTOS 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 and the value of the first offer equal to 1, possibly followed by other offers with 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* .
     not ('RECV !.* !.*' and 'RECV !\(.*\) !\1')
  ] false
```

states that all message receptions (actions "RECV !source !dest") have different source and destination fields. The UNIX regular expression construct '**\(\)**' enables to match a portion of a string and to re-use it later in the same regexp.

Note: 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.

MACROS AND LIBRARIES

evaluator3 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 program. The 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 content 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.

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.

[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.

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

[Mat98] 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

[Mat02] R. Mateescu. "Local Model-Checking of Modal Mu-Calculus on Acyclic Labeled Transition Systems". Proceedings of 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

[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.

SEE ALSO

evaluator(LOCAL), **evaluator3**(LOCAL), **evaluator4**(LOCAL), **mcl**(LOCAL), **mcl4**(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