UMC Logics (V3.2)

Franco Mazzanti (mazzanti@isti.cnr.it)

The structure of UMC temporal formulas

The logics supported by UMC is essentially the full modal/propositional mu-calculus, extended with higher level CTL/ACTL-like operators, and structured action expressions.

State Predicates

The simplest form of logic formula is a StatePredicate.
A StatePredicate can be either the formula "`true`", or the formula "`false`", or a predicate on the internal structure of a system configuration .
The formula "`true`" clearly holds for any state, the formula "`false`" never holds in any state.
A StatePredicate can also be relation between simple expressions involving attributes and valuers; such formulas hold for all the states in which the evaluation of the relation is true.
Examples: "`ASSERT(obj.objattr.intattr > 0)`"
"`ASSERT(obj3.x+1 > obj1.y+obj2.z)`"
The allowed relational operators are "**`<`**", "**`>`**", "**`=`**", "**`/=`**", "**`<=`**", "**`>=`**".
The "simple expression" can be an object attribute, a numeric value, or a sum of them.
The obj prefix of the attribute can be omitted is there system is constituted by a single active object. E.g. "`(x <= y)`" is a valid state predicate for a single object system.
"obj" is the name of one of the objects which compose the system.
"attr" is the name of one of the attributes of the object "obj". It can be also the special keyword "`queuesize`" which denotes the current length of the events queue of the object.
The keyword "`ASSERT`" can be omitted.

Boolean logical operators

Logic formulas can be composed with the usual boolean operators which have the classical meaning. The "`not`" operator has precedence over the others "`and`" "`or`" "`implies`" relational operators. Since no specific binding priority between the relational operators is defined, the boolean composition of formulas should not be ambiguous (and parenthesis should be explicitly used to disambiguate when needed).
"|" can be used as a shortcurt for "`or`", "`&`" for "`and`", "~" for "not", "`->`" for "`implies`".

Basic mu-calculus

The basic mu-calculus temporal operators are the box and diamond operators and the max and min fix point operators.

Diamond <>

The diamond operator `<action-expr>` `SUBFORMULA` allows to specify that a certain subformula should be true in at least one nextstate reachable in one step from the current

state. If an `action-expr` is specified, the nextstate should be reachable with a (one-step) system evolution which satisfies the `action-expr.`

Box []

The box operator `[action-expr] SUBFORMULA` allows to specify that a certain subformula should be true in all the nextstates reachable in one step from the current state, if any. If the current state has no successors, the diamond is still satisfied. If an `action-expr` is specified, the subformula should be true in all the nextstates reachable with a (one-step) system evolution which satisfies the `action-expr ,` if any such nextstates exist.
`[action-expr] SUBFORMULA` is equivalent to `not <action-expr> not SUBFORMULA`

Max and Min FixPoints

The power of mu-calculus is achieved through the two fixed point operators max and min. The fixedpoint identifier can only appear inside the fixedpoint body of the formula
In state s the formula: `max Z : FixedPointBody(Z)` is true if and only if the following infinite formula is true in s:

```
FixedPointBody(true)  and
FixedPointBody(FixedPointBody(true))  and
FixedPointBody(FixedPointBody(FixedPointBody(true))) and
...  and
```

In state s the formula: `min Z : FixedPointBody(Z)` is true if and only if the following infinite formula is true is s:

```
FixedPointBody(false) or
FixedPointBody(FixedPointBody(false))  or
FixedPointBody(FixedPointBody(FixedPointBody(false)))  or
 ... or
```

For example, the following formula: `max Z: <a> Z` holds in a state s1 if and only if there is a next state s2 reachable with an evolutions satisfying "`a`" in which `Z` holds again (i.e. if there exists an infinite path of evolutions satisfying "`a`" which starts from s1).

Of the two fixpoint operators only one might be considered primitive as the other one can be derived from the first. I.e:

```
max Z: FixpointBody(Z)                  is equivalent to:
not min Z:  not FixpointBody(not Z)

min Z: FixpointBody(Z)                   is equivalent to:
not max Z: not FixpointBody(not Z)
```

Even if monotonicity of a fixpoint formula is not strictly required, it is not advised to make use of non-monotonic formulas because their meaning in not necessarily intuitive.
E.g. the formula "`min Z: not Z`", holds for any state because its semantics is given by:
" `false or not false or not not false or ...`"
The above is equivalent to " `false or true or ...` " whose value is "`true`" .
It is likely that in future version of the tool the monotonicity of the formula will be required and checked.

The mu-calculus is an very powerful logic, and can be used to encode all the other CTL/ACTL like temporal constructs. However, because of its low level of abstraction, writing and understanding formulas written in pure-calculus can be a particularly hard task.
For this reason it is useful to explicitly define the other CTL/ACTL like modalities which allow a more natural expression of the system properties and a reasonably efficient way to evaluate them (the full mu-calculus is known to have a complexity which is exponential w.r.t. the alternation depth of the formula).

While the CTL /ACTL modalites will be described later here we only show a few (from simple to complex) properties which unfortunately cannot be espressed in ACTL-like modalities, and hence need the explicit use of fixpoint operators.

"*There exist an infinite path whose evolution steps all satisfy the action expression "b"*
The above can be encoded as: `max Z: (<b> Z )`

There is a reachable livelock (i.e. a loop of `tau` *evolutions) somewhere.*
The above can be encoded as: `EF max Y: <tau> Y`

There exist a path such thatan evolution which satisfies the action expression "a" occurs infinitely often.
The above can be encoded as: `max Z: min W: ((<a> Z ) or (<not a> W) )`

There exist a path such that the state predicate "p" is satisfied infinitely often by its states sequence.
The above can be encoded as: `max Z: min W: ((p and <> Z ) or ((not p) and <> W))`

There exist a path from certain point of which the state predicate "p" holds infinitely often and state predicate "q" does no longer hold.
The above can be encoded as:

```
EF  max Z:
       min W: ( (p and (not q) and <> Z)  or
                ((not p) and (not q) and <> W) )
```

For all paths, if an evolution which satisfies the action expression "a" is <u>possible</u> an infinite number of times, then an evolution which satisfies"a" is <u>done</u> an infinite number of times.
(i.e. does not exist a path such that from e certain point a *is not done anymore but nevertheles "a" is infinitely possible.*
The above can be encoded as:

```
not EF max Z:
         min W: ( ( (<a> true) and <not a> Z)  or
                  ( (not <a> true) and <not a> W ) )
```

Action Expressions

<u>Basic Action Expressions</u>

The basic form of action expression is eother the "`true`" or "`false`" predicate, or a basic action predicate about the actions performed by the system evloution.
In particular:
The action expression "`true`" is satisfied by all system evolutions.
The action expression "`false`" is not satisfied by any system evolution.

The action expression "obj:" is satisfied by any system evolution in which object "obj" is the one which evolves .
The actions expression "targetobj.event_id" is satisfied only by system evolution which generate the event event_id and send it to the object "targetobj".
The action expression "event_id" is satisfied only by those system evolutions which do generate an event "event_id", possibly with some parameters (notice that event_id might be just one of the events generated by the evolution).
The actions expression "event_id(args)" is satisfied only by system evolution which generate the event event_id event_id with the specified args, where args is a comma separated sequence of static values or "*" . E.g. the action expression "callop(*,3)" is satisfied by all evolution in which the signal or operation "callop" (which has two parameters) is being executed, and in with the second argument of the event has value "3".

Example1:

The actions expression "sourceobj:targetobj.event_id(args)" is satisfied only by system evolutions which object sourceobj generate a signal event_id with the specified args, being sent to object denoted by "targetobj".

Example2:

The action expression "event_id" is satisfied by the system evolutions labeled with any of the following labels: "event_id", "event_id(3)", "target!event_id", "target!event_id(0)", "first_event;target!event_id(1,2,3);other_events" .

Boolean Compositions of Action Expressions

...

An action expression can be a boolean composition of other action expressions.
The action expression "pred1 or pred2" is satisfied by the system evolutions which satisfy either pred1 or pred2.
The action expression "pred1 and pred2" is satisfied by the system evolutions which satisfy both pred1 or pred2.
The action expression "not pred1" is satisfied by the system evolutions which do not satisfy the action predicate pred1.

Observable Actions and Observation Modes

The concept of "observable action" is touched when we either make use of the "tau" action expression, or when the weak ACTL-like logical operators are used.
The action expression "tau" is satisfied only by the system evolutions which do not generate any observable action.
The activity of discarding an event from the events queue because it does not trigger any transitions can be considered an observable event if that is explicitly requested by the user (event "lostevent").
If an evolution does not perform any action it always satisfies the action expression "tau".

If an evolution does perform some action (assignements, signals or operation calls) the action is considered "observable" according to the users choice in terms of "observation modes".
In particular:
When the "white_box" observation mode is selcted, all actions are condidered observable.
When the "gray_box" observation mode is selcted, are condidered observable only signals and operation calls, and not the assignment actions.

When the "black_box" observation mode is selcted, are condidered observable only the signals (or operation calls) sent by the system to the outside of the model (i.e. not targeted to another active object of the model).
When the "custom" observation mode is selcted, the user should explicitly list the events and the object attribures of interest, thus making observable only certain assignments, signals and operation calls.
When the "selective" observation mode is selected, the user should explicitly list the objects of interest, thus making observable only the signals and operation calls received and generated by them, and the assignments made by them.
When the "interactions" observation mode is selected, the user should explicitly list the objects of interest, thus making observable only the signals and operation calls exchanged between any two objects of the list.

Evolution predicates

Evolution predicates are a form of action expression which describes a relation beween the attributes (variables) of the source and target states (names ending with "'").
They have the form:
<target_var>"'" <relop> <source_var_expr>
where <relop> is one of "**<", ">", "=", "/=", "<=", ">=".**

Examples are:
"(x' >= x)", "(obj1.x' /= obj1.obj2.x+1)", "(obj1.obj2.x' = 0)"

ACTL-LIKE temporal operators

Next EX AX ET AT

The Exists Next operator "EX {action-expr} subformula" holds in the current state if (and only if) there exists a transition from the current state, whose label satisfies the action-predicate, which leads to state in which the subformula holds.
It is exactly equivalent to the basic diamond operator "<action-expr> SUBFORMULA"

The Always Next operator "AX {action-expr} subformula" holds in the current state if (and only if) all the transitions which exit from the current state have a label which satisfies the action-predicate, and lead to state in which subformula holds. Moreover the current state must not be final (i.e at least a transition must exist).

"AX {action-expr} subformula" is equivalent in basic mu-calculus to:

```
(<action-expr> true) and
  (not <action-expr> not subformula) and
    (not <not action-expr> true)
```

"ET subformula" is equivalent to "EX {tau} subformula"
"AT subformula" is equivalent to "AX {tau} subformula"
"EX subformula" is equivalent to "EX {true} subformula"
"AX subformula" is equivalent to "AX {true} subformula"

Beware: in the original ACTL definition the action expression `{true}` in the contex of the Next operators had the meaning of "any observable action", while here it has the meaning of "any action (either observable or not)".

Eventually (alias Future)

The formula: "`EF Formula`" holds in a state s1 if and only if `Formula` holds in s1 or in one of the states reachable in one or more steps from s1.
The formula: "`AF Formula`" holds in a state s1 if and only if `Formula` holds in s1 or in one of the states reachable in one or more steps from s1, for all the possible paths starting from s1.

"`EF Formula`" is equivalent to its dual "`not AG not Formula`" and is equivalent to the fix point formula: "`min Z: (Formula or <>Z)`".

"`AF Formula`" is equivalent to its dual "`not EG not Formula`" and is equivalent to the fix point formula: "`min Z: (Formula or AX Z)`"

Globally

The formula: "`EG Formula`" holds in a state s1 if and only if `Formula` holds in s1 and in all the states reachable in one or more steps from s1, along at least one path starting from s1.
The formula: "`AG Formula`" holds in a state s1 if and only if `Formula` holds in s1 for all the states reachable in one or more steps from s1 (along any path).

"`EG Formula`" is equivalent to its dual "`not AF not Formula`" and is equivalent to the fix point formula: "`min Z: (Formula and <> Z)`".

"`AG Formula`" is equivalent to its dual "`~EF ~Formula`" and is equivalent to the fix point formula: "`min Z: (Formula and AX Z)`".

Until

The formula "`E[ Formula1 U Formula2 ]`" holds in a state s1 if and only if `Formula2` holds in s1 or if `Formula2` holds in a state s2 reachable in one or more steps from s1, and `Formula1` holds in all the intermediate states along the path from s1 to s2 (s1 included, s2 excluded).
"`E[ Formula1 U Formula2 ]`" is equivalent to the fix point formula:
"`min Z: ( Formula2 or (Formula1 and <> Z ))`"

The formula "`A[ Formula1 U Formula2 ]`" holds in a state s1 if and only if `Formula2` holds in s1 or if, for all paths p starting from s1: `Formula2` holds in a state $s2_p$ reachable in one or more steps from s1 and, `Formula1` holds in all the intermediate states along the path from s1 to $s2_p$ (s1 included, $s2_p$ excluded).
"`A[ Formula1 U Formula2 ]`" is equivalent to the fix point formula:
"`min Z: ( Formula2 or (Formula1 and AX Z ))`"

The formula "`E[ Formula1 {act} U Formula2 ]`" holds in a state s1 if and only if `Formula2` holds in s1 or if `Formula2` holds in a state s2 reachable in one or more steps from s1 performing only unobservable actions (satsfying `tau`) or actions satisfying `act`, and

`Formula1` holds in all the intermediate states along the path from s1 to s2 (s1 included, s2 excluded).

`"E[ Formula1 {act}U Formula2]"` is equivalent to the fix point formula:

```
"min Z: ( Formula2 or (Formula1 and <act or tau> Z ))"
```

The formula `"A[ Formula1 { act} U Formula2 ]"` holds in a state s1 if and only if `Formula2` holds in s1 or if, for all paths p starting from s1: `Formula2` holds in a state $s2_p$ reachable in one or more steps from s1 performing only unobservable actions (satsfying `tau`) or actions satisfying `act`, and `Formula1` holds in all the intermediate states along the path from s1 to $s2_p$ (s1 included, $s2_p$ excluded).

`"A[ Formula1 U Formula2 ]"` is equivalent to the fix point formula:

```
"min Z: ( Formula2  or (Formula1 and AX {act or tau} Z ))"
```

The formula `"E[ Formula1{act1} U {act2}Formula2 ]"` holds in a state s1 if exists at least one path p starting from s1 made by a (possibly empty) sequence of unobservable evolutions (satisfying `tau`) or evolutions whose label satisfies `act1` followed by a conclusive evolution whose label satisfies `act2` which leads to a state $s2_p$ in which holds `Formula2`, and in all the intermediate states along the path from s1 to $s2_p$ (s1 included, $s2_p$ excluded) `Formula1` holds.

`"E[ Formula1 { act }U { act2 }Formula2 ]"` is equivalent to the fix point formula:

```
"min Z: Formula1 and ((<act2> Formula2) or <act1 or tau> Z})"
```

The formula `"A[ Formula1 { act1 }U { act2 } Formula2 ]"` holds in a state s1 if and only if all paths p starting from s1 are made by a (possibly empty) sequence of unobservable evolutions (satisfying `tau`) or evolutions whose label satisfies `act1` followed by a conclusive evolution whose label satisfies `act2` which leads to a state $s2_p$ in which holds `Formula2`, and in all the intermediate states along the path from s1 to $s2_p$ (s1 included, $s2_p$ excluded) `Formula1` holds.

`"A[ Formula1 { act1 } U { act2 } Formula2 ]"` is equivalent to the fix point formula:

```
min Z: ( Formula1 and
         (not FINAL) and
         ([act2 and (not tau) and (not act1)] Formula2) and
         ([(act1 or tau) and (not act2)] Z )  and
         ([(act1 or tau) and act2] (Formula2 or Z) ) and
         ([not act1) and (not act2) and (not tau)] false)
```

So far, the above are the only currently supported flavours of "until" operators.
Further flavours are sometimes used in other logics , as for example versions of "weak until", or the "release" operator which is the dual of the normal "until".

Weak Diamond <<act>>

The formula: "<<act>> Formula" holds in a state s1 if and only if Formula holds in s1 or in one of the states reachable in one or more steps from s1.performing only internal actions (satisfying tau) or actions satisfying act.
act is not allowed to contain "tau" as subexpression.

Notice that in the original ACTL the syntax of this temporal operator was "<act> Formula" (now used for the basic mu-calculus diamond operator). From this point of view this operations consitutes a "backword incompatibility" between UMC and the original ACTL.

Weak Box [[act]]

The formula " [[act]] Formula " holds in state s1 if the formula " Formula " holds.in all the states (if any) reachable from s1 performing an evolution whose label satisfies act, possibly after a finite sequence of unobservable evolutions (satisfying tau).
act is not allowed to contain "tau" as subexpression.
"[[act]] Formula" can be translated as:
"max Z: ((~EX {tau} ~Z) & ~(EX {act} ~Formula))" and is equivalent to:
~< act > ~Formula

Notice that in the original ACTL the syntax of this temporal operator was "[act] Formula" (now used for the basic mu-calculus box operator). From this point of view this operations consitutes a "backword incompatibility" between UMC and the original ACTL.

UMC LOGIC GRAMMAR

-- umc formulas

```
FORM ::=
    state-predicate
   | and-sequence
   | or-sequence
   | negation
   | implication
   | basic-mu-formula
   | ctl-like-formula
```

-- state predicates

```
state-predicate ::=
   "true"
   | "false"
   | simple-expr
        [ "<", ">", "=", "/=", "<=", ">=" ] simple-expr

simple-expr ::=
    basic-expr
    | basic-expr "+" basic-expr

basic-expr ::=
    [obj"."]attr{"."attr}
    | value
```

-- boolean logical operators

```
and-sequence ::=  FORM "and" FORM { "and" FORM }
or-sequence ::=   FORM "or" FORM { "or" FORM }
negation ::=  "not" FORM
implication ::=  FORM "->" FORM
enclosing ::=  "(" FORM")"
```

-- basic mu-calculus

```
basic-mu-formula ::=
   maxfix  | minfix | mu-box | mu-diamond | fixid

maxfix  ::=  "max" VARID ":" FORM
minfix  ::=  "min" VARID ":" FORM
mu-box  ::=  "<" [action-expr] ">" FORM
mu-diamond  ::=  "[" [action-expr] " ]" FORM
fixid  ::=  VARID
```

-- action expressions

```
action-expr ::=
   "true"
  | "false"
  | "tau",
  | action-expr  "and"  action-expr {"and" action-expr}
  | action-expr  "or"  action-expr  {"or" action-expr}
  | "not" action-expr
  | [source":"] [target"."] [event] [args]
  | "("action-expr")"
  | "("obj"."]attr{"."attr}"'"
      ["<", ">", "=", "/=", "<=", ">="]
         simple-expr ")"
```

-- ACTL like formulas

EX FORM

AX FORM
(not FINAL and [true] FORM

E[FORM1 **U** FORM2**]**
min Z:(FORM2 or (FORM1 and
<true> Z))

A[FORM1 **U** FORM2**]**
min Z:(FORM2 or (FORM1 and
(not FINAL) and
[true] Z))

EF FORM
min Z: (FORM or <true> Z)

AF FORM
min Z:(FORM or not FINAL and [true] Z))

EG FORM
max Z: (FORM and (FINAL or <true> Z)

AG FORM
max Z: (FORM and (FINAL or [true] Z))

EX {action-expr**}** FORM *-- strict*
<action-expr> FORM

AX {action-expr**}** FORM *-- strict*
(not FINAL) and
([not action-expr] false) and
[action-exp] FORM

E[FORM1 **{**action-expr**} U** FORM2**]** *-- weak on action-expr*
min Z:(FORM2 or (FORM1 and
<tau or action-exp> Z))

A[FORM1 **{**action-expr**} U** FORM2**]** *-- weak on action-expr*

```
min Z:(FORM2 or (FORM1 and
                 (not FINAL) and
                 ([not (tau or action-expr)] false) and
                 [tau or action-exp] Z))
```

E[FORM1 **{**action-expr1**}** **U** **{**action-expr2**}** FORM2**]**
--weak on first action-expr

```
min Z:(FORM1 and
       ((<action-exp2> FORM2) or
        <tau or action-exp1> Z)))
```

A[FORM1 **{**action-expr1**}** **U** **{**action-expr2**}** FORM2**]**
--weak on first action-expr

```
min Z:(FORM1 and
       ((not FINAL) and
        ([not action-exp2 and(action-expr1 or tau)] Z) and
        ([action-exp 2 and not (action-expr1 or tau)] FORM2) and
        ([action-exp 2 and (action-expr1 or tau)] (FORM2 or Z) and
         [not action-exp2 and not (action-expr1 or tau)] false))
```

<<action-expr**>>** FORM *-- weak before action-expr*

```
min Z: ((<action-expr> FORM) or <tau> Z )
```

[[action-expr**]]** FORM *-- weak before action-expr*

```
min Z: (([action-expr] FORM) and [tau] Z)
```

More Shortcuts
FINAL