Monterey Phoenix

System and Software Architecture Modeling Language (version 2.0)

Mikhail Auguston
Naval Postgraduate School
Monterey, California, USA
maugusto@nps.edu
wiki site: http://wiki.nps.edu/display/MP

Extended BNF notation

```
(* P *) – P repeated zero or more times
(+ P +) – P repeated one or more times
(* P * E) – P repeated zero or more times, separated by E
(+ P + E) – P repeated one or more times, separated by E
(A | B | C)  - alternative
[ A ]  - optional A
```

Lexics

```
'ttt'  - terminal string
Id     - identifier
Spaces, eol, and tabs are not syntactically significant
comments /* … */ and // … eol are allowed in the MP source text.
Note. // … eol comments are not yet available
```

Some conventions

The trace derivation determines the MP semantics. Leftmost trace derivation for schemas, roots, and composite events is performed top-down and from left to right using the event grammar rules. Composition operations act like a "crosscutting" derivation rules that may add new events and basic relations to the trace under derivation.

The *trace space* is the set of events and binary relations between them emerging as a result of derivation process. It is produced by grammar rules and composition operations.

```
For a schema
SCHEMA S
ROOT R1: …;
ROOT R2: …;
…
ROOT Rn: …;
```

Schema's S event trace derivation follows an implicit grammar rule

```
S: { R1, R2, … , Rn }
```

and encompasses the composition operations interlacing the root rules and defined in the BUILD blocks attached to the schema and to other event grammar rules.

Any root or included schema's instance (NEW schema_name;) should appear in the MP source code before its use in composition operation, so that the trace derivation from the grammar rules could be accomplished before application of composition operations on those traces. Reused schemas should be declared with the INCLUDE statement.

This is a draft document for MP2 beta-testers.
MP2 is still a work in progress.

Syntax rules for MP2 constructs not yet implemented are highlighted in red.

Syntax rules

```
(1) model:    schema(2)
                 (* ( assertion(32) | query (48)) ';' *)
;

(2) schema:   'SCHEMA' schema_name(3)
                 (* 'INCLUDE' schema_name(3) ';' *)
                 (* (  rule(4)                        |
                       composition_operation(20)  |
                       'NEW' schema_name(3)        )
                    ';'
                 *)
                 [ Build_block(31)]
;
```

Reused schemas should be announced with INCLUDE before the schema's name appears somewhere in NEW composition. The scope of the host schema is applied to the included schema.

Derivation of the schema's trace proceeds top-down and from the left to right. Composition operations outside the BUILD block are performed following the order of derivation, immediately after the root derivation and composition operations appearing in the schema before them were performed, and hence may speed up the trace derivation by rejecting the previously derived root traces before proceeding with the next roots. Operations within the BUILD block are performed after all the derivations in schema's body have been completed. Schema's BUILD block may also define attributes for the whole schema.

```
(3) schema_name: Id  -- schema's name is an identifier
;

(4) rule: ('ROOT' root_name(5)| event_name(8) ) ':'
                pattern_list(6)
                [ Build_block(31)]
;
```

-- absence of the ROOT keyword yields a composite event declaration

```
(5) root_name: Id     root name is an identifier
;

(6) pattern_list:    (* pattern_unit(7) *)
;

(7) pattern_unit:(   event_name(8)        |
                     alternative(9)       |
                     iteration(11)        |
                     iterator_plus(13)    |
```

```
                    set(14)              |
                    set_iterator(15)     |
                    set_iterator_plus(16)|
                    optional(17)         |
                '<|' pattern_list(6) when_clause(18) '|>'|
                    empty
                )
;
```

Alternative with a single branch yields just a sequence pattern.
Empty option is needed to balance probabilities assignment, e.g. in (A | B |) empty alternative has probability 1/3, but in [(A|B)] it will have probability ½.
ROOT event may be initially empty, and then assigned some contents via MAP
<| ... |> provides the scope for the WHEN clause, see(18) for details

(8) `event_name:  Id`
-- event name is an identifier, atomic or composite event name
`;`

(9)
```
alternative:
    '(' (+ [ probability(10)] (pattern_list(5) +'|') ')'
;
```
-- probability to select the alternative during the trace generation,
-- each selection is independent from the previous history
-- empty option is needed to balance probabilities assignment, e.g. in (A|B|) empty
-- alternative has probability 0.33333, but in [(A|B)] it will have probability 0.5;

(10)
```
probability: '<<' float_number  '>>'
;
```
-- float_number should be in the range 0.0 – 1.0,
-- sum of probabilities within the alternative should be 1.0
-- probabilities not provided explicitly, are assigned evenly to supplement the total sum to 1.0
-- probabilities are used only in the random trace generation mode

(11)
```
iteration:
    '(*' [ iteration_scope(12)] pattern_list(5)'*)'
;
```

(12)
```
iteration_scope: '<' [   natural_number '..' ]
                         natural_number      '>'
;
```
-- iteration range < min .. max > covered during trace generation
-- the default is < 0 .. scope >
-- < n > is the same as <n .. n>

(13)
```
iterator_plus:
        '(+' [ iteration_scope(12)] pattern_list(6)'+)'
```

```
;
-- the default is < 1 .. scope >
-- the lower limit can not be less than 1

(14)  set:       '{' (*  pattern_list(6) * ',') '}'
;
-- set implies only IN relation between the set and its elements
-- the default is  < 0 .. scope >
--  < n > is the same as <n .. n>

(15)  set_iterator:
           '{*' [ iteration_scope(12)] pattern_list(6) '*}'
;

(16)  set_iterator_plus:
           '{+' [ iteration_scope(12)] pattern_list(6) '+}'
;
-- the default is < 1 .. scope >
-- the lower limit can not be less than 1

(17)  optional: '[' [ probability(10)] pattern_list(6)']'
;
-- the default probability is 0.5

(18)  when_clause:    'WHEN' (+ when_unit(19) + ',')
;
-- the scope of WHEN clause determines the events which could be interrupted, see (7)

(19)  when_unit:
      [probability(10)] event_name(8) '==>' pattern_list(6)
     -- event_name is a non-root event
;
```

WHEN unit can be interrupted by another triggering event;
it will be included as alternative for each event (atomic, composed, or container) in the WHEN scope (with the probability, if defined; probabilities are prorated if needed);
triggering event should not appear within pattern list preceding WHEN unit in the WHEN block;

Semantics:

WHEN event defines an interruption of the trace within its scope and continuation with the WHEN event and its associated pattern list. This option may be computationally expensive and should be used with care, to specify for instance an event that may occur in the environment at any unpredictable time moment.

We need the concept of CUT(E) as a pattern specifying possible initial segments of event trace for the event pattern E, when E has been interrupted while generating its event trace.

1) CUT(A) = [A] if A is atomic event pattern
2) CUT(A) = [A: CUT(Body)] if A is a composite event A: Body
3) CUT(A B) = (CUT(A) | A CUT(B))
4) CUT((A | B)) = (CUT(A) | CUT(B))
5) CUT((<n..m> E *)) = (* <0..m-1> E *) CUT(E)*
6) CUT({A, B}) = { CUT(A), CUT(B) }
7) CUT({ <n..m> E *}) = {* <0..m> CUT(E) *}*
8) CUT((+ E +)) = (E *) CUT(E)*
9) CUT({+ E +}) = { CUT(E) *}*

The meaning of when-clause pattern can be defined as:

<| P WHEN <p1> A1 **==>** *A2, <p2> B1* **==>** *B2, ... |> =*
(P |
CUT(P)
((<p1> A1 CUT(A2) | <p2> B1 CUT(B2) | ...) *)*
(<p1> A1 A2 | <p2> B1 B2 | ...))
MP trace scope limit is applied to the induced iteration in WHEN definition as well.

Composition operations

(20)
```
composition_operation:
  (   shared_composition(21)          |
      coordinate_composition(23)      |
      'ENSURE' bool_expr(38)          )
;
```

(21)
```
shared_composition:
        (+
           (+ (root_name(5)| variable(25) ) + '|+|')
         + ',')
        'SHARE' 'ALL'
        (+  event_name(8) + ',')
;
```

A |+| B means exclusive union (partition), i.e. if A |+| B, C SHARE ALL D each event D in C should be also either in A or in B, but not in both. This construct is useful for specifying the use of critical shared resources. Usually architecture models don't address resource contention problems, assuming that resolution for it is provided by lower level design.

(22)
```
new_instance: 'NEW' ( schema_name(3) | event_name(8) )
;
```

NEW brings to the trace space a new instance of event. No duplication of this schemas root event is allowed. NEW cannot be applied to the included schema's roots or composite events.

(23)
```
coordinate_composition:
```

```
            'COORDINATE' ['<!>']
            (+ coordination_source(24) + ',')
            'DO'
            (+ ( variable(25) ':' new_instance(22)   |
                 add_relation(26)                    |
                 MAP_composition(29)                 |
                 shared_composition(21)              |
                 coordinate_composition(23)
               )
              ';' +)
            'OD'
;
```

COORDINATE is essentially a loop, synchronous or asynchronous, over one or more event sets of equal size (coordination sources), for each tuple of events selected from the coordination sources the composition operations within DO – OD are performed. The COORDINATE composition uses event selection patterns to specify subsets of traces that should be coordinated.

Synchronized composition requires that events selected in each coordinated trace are totally ordered (with respect to the transitive closure of PRECEDES), selected event sets should have the same number of elements, and the coordination follows this ordering (synchronous coordination). If any of selected event sets is not totally ordered, the synchronized coordination operation fails to produce a resulting trace.

Presence of <!> means asynchronous coordination. Selected sets may be totally ordered or not. But now the resulting merged traces will include all permutations of events from the coordination sources. This assumes that other constraints, like the partial ordering axioms are satisfied. Each permutation yields one potential instance of a resulting trace for the schema deploying this composition. Use of <!> may significantly increase the number of composed traces.

(24) `coordination_source:`

```
            variable(25) ':' selection_pattern(27)
            [ 'FROM' (root_name(5) | variable(25) | 'this') ]
;
```

'this' refers to the schema, root or composite event in the BUILD block of which it appears. Absence of FROM clause is equivalent to 'FROM this'

(25) `variable: $Id` *-- MP variable is an identifier preceded by '$'*

```
;
```

(26) `add_relation: 'ADD'`

```
        (+ variable(25)
              ('IN' | 'PRECEDES' | 'CONTAINS' | 'FOLLOWS')
           variable(25) + ',')
;
```

```
(27)  selection_pattern:
          ( event_name(8)                          |
            alternative_of_event_names(28)         )
;

(28)  alternative_of_event_names:
          '(' (+ event_name(8) + '|') ')'
;

(29)  map_composition:
          (+ 'MAP' map_unit(30) 'AS' map_unit(30) +',')
;
```

-- the purpose of map construct is to implement event aliasing, source and target events may be in the current or reused schema; this can be considered as a generalization of the SHARE ALL;

```
(30)  map_unit:
          ( event_name(8)
              ['FROM'( root_name(5) | variable(25) |'this')]|
            variable(25)                                      |
            root_name(5) ['FROM' (variable(25) | 'this')]     )
;
```

FROM clause absence is equivalent to 'FROM this'

Build Blocks

Build blocks describe event trace derivation activities performed when an instance of event is added to the tracespace (analog to the constructor method in OO programming languages).

```
(31)  Build_block:
          'BUILD' '{'
```

Event attribute is a binary relation added between the event associated with Build_block and other event or object (number, string, Boolean value)

```
          (* ( plain_attribute_declaration(33) |
               [ 'disj'] relation_name(32) ':'
               (+ ( event_name(8) | new_instance(22) ) +',')|
```

event name without NEW refers to an event instance already present in the derivation; 'disj' ensures that event instances brought from the already accomplished derivation are distinct from each other, NEW event instances are disjoint by definition.

```
                  variable(25) ':' new_instance(22)   |
```

creates a new instance of event (including the whole schema event) for use in the following composition operations.

```
                composition_operation(20)          )
            * ';')
            '}'
;
```

```
(32)  relation_name: Id
;
```

relations IN, CONTAINS, PRECEDES, FOLLOWS are predefined

Event plain attribute declarations

Event attributes are specified in BUILD block(31). Immutable attributes only in this version, i.e. for each event instance the value of event attribute is assigned only once, but in case of RAND(n..m) the value may be different for different instances of the same event type. No dependencies between attributes (synthesized or inherited attributes) in this version as well.

```
(33)  plain_attribute_declaration:
      (+ Id + ',') ':' type(34) ['=' initial_value(35) ]
;

(34)  type:
    ( 'int' | 'double' | 'string' | 'bool')
;

(35)  initial_value: (integer_number | float_number |
            string_constant | 'RAND' '(' interval(36) ')' |
            'true' | 'false' )
;

(36)  interval:  integer_number '..' integer_number
;

-- predefined attributes:
-- double clock_begin
-- double clock_end
-- double duration
-- string event_type
```

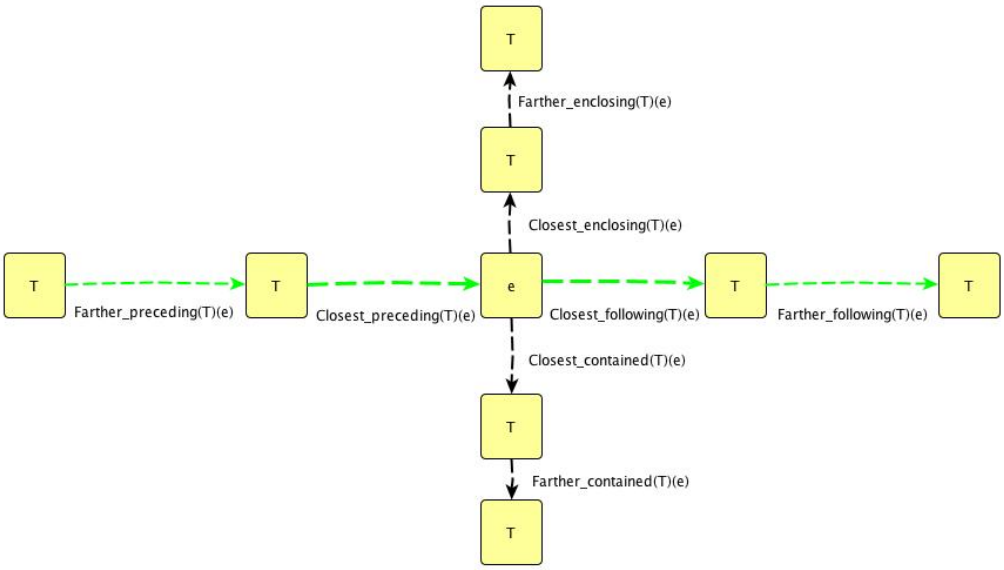
Assertions and Queries

Assertions

```
(37)  assertion:      'ASSERT'   Id    -- assertion name
                           bool_expr(38)
```

```
;

(38)  bool_expr: (    ('FOREACH' | 'EXISTS') variable(25) ':'
                          ( root_name(5) |
         event_name(8) 'FROM' (root_name(5)|variable(25))
                          )
                          bool_expr(38)                    |
                     bool_expr0(39)
                  )
;

(39)  bool_expr0:    bool_expr1 (* '->' bool_expr1 *)
;

(40)  bool_expr1:    bool_expr2 (* 'OR' bool_expr2 *)
;

(41)  bool_expr2:    bool_expr3 (* 'AND' bool_expr3 *)
;

(42)  bool_expr3:    (    bool_aggregate_op(59)     |

                          '(' bool_expr(38) ')'     |

                          'NOT' bool_expr3(42)      |

                          special_predicate (46)    |

                          ( event_instance(43)
                               ('PRECEDES'   | 'IN'   |
                                'PRECEDES*'  | 'IN*' )
                            event_instance (43)        )|

                          ( expression(53)
                            comparison_operation(43)
                            expression(53) )
                     )
;
-- need to separate IN for direct, and IN* for transitive closure, similarly for PRECEDES
and PRECEDES*

(43)  comparison_operation:
           ( '<' | '<=' | '==' | '!=' | '>=' | '>' )
;

(44)  event_instance:
```

```
                ( variable(25) |
                  event_location(45) '(' event_name(8) ')'
                          '(' variable(25) ')'
                          [ 'SUCH' 'THAT' bool_expr(38) ]
                     -- yields an event object, or NULL if such does not exist
                 )
;

(45)  event_location:
          ( 'Closest_enclosing'| 'Farthest_enclosing' |
            'Closest_contained'| 'Farthest_contained' |
            'Closest_preceding'| 'Farthest_preceding' |
            'Closest_following'| 'Farthest_following'   )
;
```



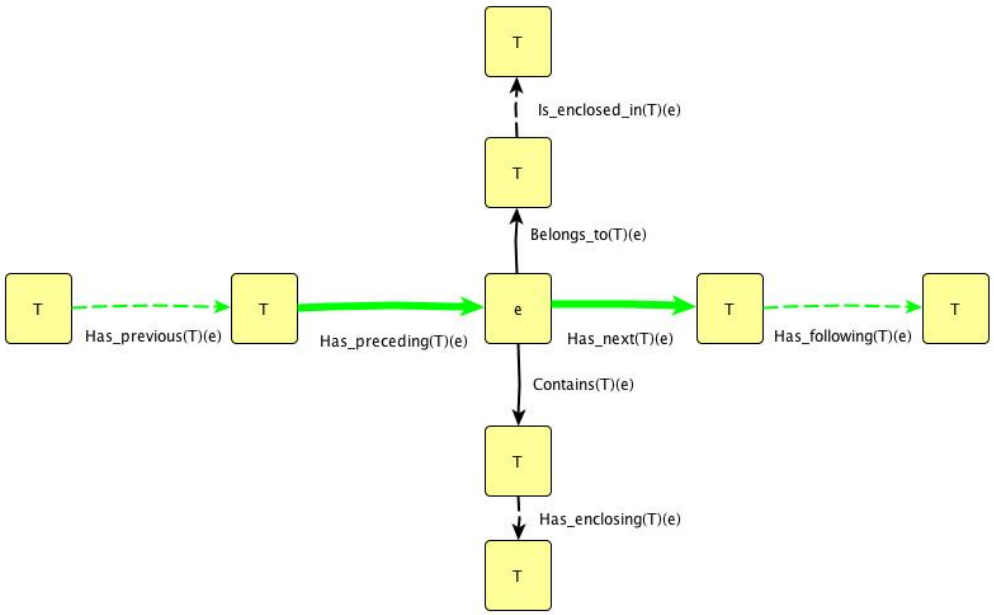

```
(46)  special_predicate:
          direction(47)  '(' event_name(8) ')'
                         '(' variable(25) ')'
                         [ 'SUCH' 'THAT' bool_expr (38)]
;

(47)  direction: (   'Is_enclosed_in' | 'Has_enclosing  |
                     'Has_previous'   | 'Has_following' |
```

-- these predicates are based on transitive closures of IN and PRECEDES**

```
                     'Belongs_to'    | 'Contains' |
                     'Has_preceding'| 'Has_next'    )
```

-- these predicates are based on direct IN and PRECEDES

;



Queries

-- a query is evaluated separately on each event trace;

```
(48)  query:
      (     SAY_clause(49)        |     global_query(50)     |
            'CHECK'     ( Id        |     -- Id is assertion name
                          bool_expr(38))
                  [ SAY_clause ]
                  [ 'ONFAIL' SAY_clause ]
      )
;

(49)  SAY_clause:     'SAY' '(' (+ expression(53) +) ')'
;
```

-- global queries are evaluated on the whole set of generated traces for a given MP mode, they are distinguished by the presence of $$TOTAL variable or global_estimate expression;

```
(50)  global_query:  SAY_clause_1
;

(51)  SAY_clause_1: 'SAY' '(' (+ (  global_estimate(52)|
                                    text_string |
                                    '$$TOTAL'       )
                                +) ')'
;
```

```
(52)  global_estimate: 'P?' '/' bool_expr (38)
;
```
-- returns the ratio of number of traces satisfying bool_expr to the total number of generated traces;

Semantics:

Queries are performed after the generation of an event trace or the whole set of traces within given scope has been completed (in exhaustive or random mode). Each query generates its own result.

CHECK A will search the set of generated event traces for the counterexample, a trace, which does not satisfy assertion A.

If operation within an expression cannot be completed, it yields NULL as a result.
NULL also represents Boolean constant False, whereas any value different from NULL represents Boolean constant True.

```
(53)  expression:  aggregate_op (56)      |  bool_expr (38)|
                   event_instance '.' event_attribute_name |
                -- event attributes may be used in
                -- expressions built from
                --  constants, metavariables
                -- operations  +, -, *, /, unary -,  ==, !=,  >, <, >=, <=,
                --  set-theoretical operations +(union), & (intersection),
                --  - (set difference), etc.

                special_function (54)|
                NULL |   -- empty set,
                         --  0 if used as an argument in an operation
                integer_number | float_number |
                string_constant
;

(54)  special_function:
      'Number_of' '(' event_name(8) ')'
           navigation_direction(55) '(' ( variable(25) |
                                          root_name    )')'
;

(55)  navigation_direction: (  'before'  |
                               'after'   |
                               'in'      |
                               'enclosing' )
;
```

(56) **aggregate_op:**

```
          [ operation(57) '/' ]
```

If operation is omitted, the result is a set of events

```
          '{'  [variable(25) ':'] search_pattern (58)
```

The scope of variable is that aggregate operation only.

```
          [ 'FROM' ( root_name(5)   |
                     variable(25)   |
                     event_path_attribute (60) ) ]
```

If FROM is omitted, the range assumed to be the whole event trace

```
          ['APPLY' expression ]
```

If APPLY is present, the aggregate operation returns multi-set of non-event values (integer, string, etc., depending on the type of expression.)

```
     '}'
;
```

(57) **operation:**

```
     (    'SUM'      |     -- numerical sum
          'TIMES'    |     -- numerical product
          'MAX'      |     -- numerical max
          'MIN'      |     -- numerical min
          'CARD'     |     -- number of elements in the resulting (multi-) set
          'AVG'      |     -- result of SUM divided by CARD
          'OR'       |     -- yields bool, requires APPLY bool_expr
          'AND'            -- yields bool, requires APPLY bool_expr
     )
;
```

(58) **search_pattern:**

```
     (    ( event_name(8) | 'ANY' ) |
          '(' (+ [variable(25) ':'] event_name(8) +)     ')' |
          '{' (+ [variable(25) ':'] event_name(8) +',') '}' |
          '(' (+ [variable(25) ':'] event_name(8) +'|') ')'
     )
     [ 'SUCH' 'THAT' bool_expr(38) ]
;
-- event_name may be atomic or composite event type;
-- ANY  matches any event;
-- Typical use:
-- Chain:  ( A B C )
-- Concurrent slice: { A, B, C }
-- Concurrent( v)  means that for each pair of events e1, e2 IN v
        NOT ( e1 PRECEDES e2 OR  e2 PRECEDES e1), although e1 IN e2 is OK
```

See Example 17.

-- Alternative: (A | B | C)
-- Local variables: ($x: A B $y: C) and then use them in the expression, like
-- OR/{ ($x: A B $y: C) APPLY ($y.time_end - $x.time_begin > 20) }
-- the '&&' bool_expr provides additional SUCH THAT constraint

(59) **bool_aggregate_op:** *-- is aggregate_op(56) with OR or AND operation*
;

(60) **event_path_attribute: variable(25) '.'**
(PREV_PATH |
NEXT_PATH |
PARENTS | DESCENDANTS
);
-- these attributes provide sets of events under PRECEDES and IN* relations with the event in variable. Used in (56)*

Abbreviations for some bool_aggregate_op

```
FORALL variable: search_pattern FROM source | bool_expr
```

is an abbreviation for

```
AND/{ variable: search_pattern FROM source APPLY bool_expr }
```

Correspondingly,

```
EXISTS variable: search_pattern FROM source | bool_expr
```

is an abbreviation for

```
OR/{ variable: search_pattern FROM source APPLY bool_expr }
```

Another abbreviation.

```
EXISTS variable: search_pattern FROM source
```

is an abbreviation for

```
CARD/{ variable: search_pattern FROM source } > 0
```

Examples of queries

1) SAY “number of send/receive” CARD/ { $x: (send | receive) FROM Task_A };

2) CHECK SUM/{ $y: send FROM Task_A APPLY $y.duration } > 100

3) CHECK NOT EXISTS {Generator_off, Radar_working};
-- slice, by default FROM extends to the whole trace