The MMLD-GROUND modelling language

Alban Grastien

September 2023

This document explains the modelling language MMLD-GROUND that was developed at NICTA (Australia) and used for model based diagnosis purposes. MMLD-GROUND is a grounded version of the MMLD (Marmelade) language that is not presented here. The grounded language is a restricted sublanguage of the Marmelade language where each variable is explicitly defined.

1 Syntax

The syntax of the MMLD-GROUND is given on Fig. 1.

2 Semantics

A system is defined as a *network* of interconnected *components*. Each component is described explicitly, and not as an instantiation of a generic component type.

Each component is a transition system described through *state variables*, *events*, and symbolic *transitions* (i.e., the transitions are not defined explicitly between states but through their interactions with the variables). The state of the network is then essentially the value of each variable of each component, plus the value of (continuous) clock variables, as explained later.

A component has a unique name given in the header of the component's definition. It has a set of (state) variables, which are partitioned into *dependent variables* (introduced by the keyword dvar) and *non-dependent variables*. A non-dependent variable is a variable whose value is modified exclusively via transitions; a dependent variable can change freely, with the one condition that the constraints (defined via keyword `constraint`) must be satisfied. For instance, assume that the component represents a room with a door and a window, and that there will be a draught only if both are open. This component will probably be modelled via three variables: `door_open`, `window_open`, and `has_draught`. The latest will be a dependent variable, with the constraints `(has_draught = true) => (door_open = true)` and `(has_draught = true) => (window_open = true)`. Note that, in this example, the door and window being both open does not imply that there is a draught; furthermore, the room may flip from being draughty/non-draughty without the other variables changing. The domain of a variable is either a set of names or a set of integers defined as a range.

Each component has a number of events defined as a name. Events from different components can be *synchronised*, e.g., through the following line:

```
synchronize comp1.event1, comp2.event2, comp3.event3;
```

Synchronised events always occur together: whenever event `event1` of component `comp1` occurs, both events `event2` and `event3` of components `comp2` and `comp3`, respectively, occur simultaneously. There cannot be a cycle of events such as `event1` of `comp1` generating `event2` of `comp2`, generating `event3` of `comp3`, generating `event4` of `comp1`.

Transitions explain how events and changes of state variables value interact. A transition has a name used mainly for debugging. There are three ways to trigger a transition:

```
network         ::= component+ (synchronisation | observable | initial)*

component       ::= 'component' ID '{' comp_def* '}'
comp_def        ::= var_def | event_def | conn_def | trans_def | const_def
var_def         ::= ('var' | 'dvar') ID ':' (interval | set) ';'
event_def       ::= 'event' ID ';'
conn_def        ::= 'connection' ID ':' ID = ID ';'
trans_def       ::= 'transition' ID rule_def* trigger_conds?
rule_def        ::= ID formula '->' a_or_events ';'
a_or_events     ::= a_or_event (',' a_or_event)*
a_or_event      ::= assignment | event
assignment      ::= ID ':=' expression
event           ::= ID
trigger_conds   ::= 'triggeredby' trigger_cond (',' trigger_cond)* ';'
trigger_cond    ::= event | ('[' FLOAT '..' FLOAT ']' formula)
const_def       ::= 'constraint' formula ';'

synchronisation ::= 'synchronize' ID '.' ID (',' ID '.' ID)+ ';'
observable      ::= 'observable' ID '.' ID ';'
initial         ::= ID '.' ID ':=' ('true' | 'false' | INT | ID) ';'

formula         ::= conjunction ('or' conjunction)*
conjunction     ::= atomic ('and' atomic)*
atomic          ::= (expression '=' v_expression)  | 'FALSE' | 'TRUE'
                  | '(' formula ')' | 'NOT' atomic
expression      ::= sexpr ('+' sexpr)*
sexpr           ::= INT | 'true' | 'false' | ID '.' ID | ID
v_expression    ::= INT | 'true' | 'false' | ID
interval        ::= '[' INT '..' INT ']'
set             ::= '{' ID (',' ID)* '}'
```

Figure 1: Syntax of the MMLD-GROUND language

1. A transition can be *spontaneous*, in which case there is no `triggeredby` attached to the transition.

2. A transition can be triggered by a condition being satisfied for a given amount of time. The condition is then attached a time interval `[min,max]` meaning that the transition may trigger as soon as the condition was satisfied for duration greater than `min` and will definitely trigger before the condition is satisfied for duration of `max`.

3. A transition can be triggered by the occurrence of a synchronised event produced by the occurrence of another transition on another component. The transition is simultaneous.

The `triggeredby` conditions are disjunctive (i.e., one of the conditions is sufficient to trigger the transition). The occurrence of a transition leads to the resolution of exactly one *rule* of the transition. For a rule to resolve, the formula before `->` must be satisfied; if multiple rules have their condition satisfied, any one rule can resolve. If no rule can resolve, the empty rule does (nothing happens); notice however that the empty rule resolve only if no other rule's condition is satisfied. (Notice that rules have a name which can be used for debugging purposes.) The resolution of a rule leads to a list of assignments (change of value of one of the component's variables) and the production of events (these events can then be synchronised with other events, leading to the simultaneous trigger of transitions on other components).

The network also specifies the *observable events*. When an observable event occurs, the observer sees this event. Similarly, one can specify the initial value of some state variables via the `initial` keyword.

Formulas are essentially logical conditions on the value of some state variables.