Languages in the MaC prototype implementation

MaC Research
University of Pennsylvania

1 Overview

This document decribes the languages, PEDL and MEDL, that are used in the MaC prototype implementation. Scripts written in these languages are used to automatically produce the *filter*, *event recognizer*, and *checker*, which are the three main components at run-time of a monitored system in the MaC framework. Before introducing these languages, we provide a quick overview of the MaC architechture. Details of the framework can be found in [KVBA+98, KVBA+99, LKK+99].

1.1 MaC Framework.

The MaC framework describes an approach to performing monitoring and checking of a running system with the express aim of ensuring that it is running correctly, with respect to a formal requirements specification. The run-time monitoring and checking (MaC) architecture consists of three components: *filter*, *event recognizer*, and *run-time checker*. The filter extracts low-level information (such as values of program variables and time when variables change their values) from the instrumented code. The filter sends this information to the event recognizer, which converts it into high-level events and conditions and passes them to the run-time checker.

Each event delivered to the checker has a timestamp, which reflects the actual time of the occurrence of the event. This enables one to monitor real-time properties of the system. Timestamps are assigned to events by the event recognizer based on the clock readings provided by the filter. The run-time checker checks the correctness of the system execution *thus far* according to a requirements specification of the system, based on the information it receives from the event recognizer, and on the past history.

The prototype implementation, for whose languages this is a manual, supports the monitoring of systems written in Java.

1.2 Motivations in Language Design.

As described above, the filter informs the monitor (event recognizer and checker combine) whenever an "interesting" state change occurs. The information from the filter falls into two natural classes: events and conditions.

Events occur instantaneously during the system execution, whereas conditions are information that hold for a duration of time. For example, an event denoting return from method `RaiseGate` occurs at the instant the control returns from the method, while a condition `(position == 2)` holds as long as the variable position does not change its value from 2. Distinction between events and conditions is very important in terms of what the monitor can infer about the execution based on the information it gets from the filter. For an event, the monitor can conclude that an event does not occur at any moment except when it receives an update from the filter. By contrast, once the monitor receives a message from the filter that variable `position` has been assigned the value 2, we can conclude that `position` retains this value until the next update.

This distinction between events and conditions is formalized in a logic that forms the basis for the semantics of both PEDL and MEDL. This logic is presented in the Appendix A.

Another factor that we must take into account while describing the semantics of the MaC languages, is how to evaluate expressions containing undefined variables. This is because, the requirements are described in terms of program variables which routinely go out of scope. We therefore choose to evaluate logical expressions over 3 values: **true**, **false**, and undefined (Λ). These ideas are formally described in the Appendix A, where the logic underlying the MaC languages is described.

2 Primitive Event Definition Language (PEDL)

PEDL is the language for writing monitoring scripts. The design of PEDL is based on the following two principles. First, we encapsulate all implementation-specific details of the monitoring process in PEDL scripts. Second, we want the process of event recognition to be as simple as possible. Therefore, we limit the constructs of PEDL to allow one to reason only about the current state in the execution trace. The name, PEDL, reflects the fact that the main purpose of PEDL scripts is to define primitive events of requirement specifications.

2.1 Structure of PEDL scripts

The BNF grammar is given in Figure 2. The PEDL script begins with the keyword `Monscr` followed by the name of the script, and ends with the keyword `End`. The PEDL script itself contains the list of events and conditions that are "exported", the list of program variables and methods that are need to be monitored, and the description of how low level program variable values are mapped to higher level events and conditions.

The list events (and conditions), whose truth and falsity is reported by the event recognizer to the checker, are listed after the keywords `export event` (and `export condition`). The definition of these events (and conditions) is then described later in the script. The monitored objects and methods are declared after the keyword `monobj` and `monmeth`, respectively.

2.2 Monitored Entities.

PEDL scripts can refer to any object of the target system. This means that declarations of monitored entities are by necessity specific to the implementation language of the system. In the current prototype which is based on Java, values of fields of an object, as well as of local variables of a method, and method calls can be monitored. Examples of monitored entities' declarations are given in Section 4.

2.3 Defining Conditions.

Primitive conditions in PEDL, are constructed from boolean-valued expressions over the monitored variables. An example of such condition is

```
condition TooFast =
     Train.Position().speed > 100.
```

In addition to these, we have primitive condition `InM(f)`. This condition is true as long as the execution is currently within method `f`. Complex conditions are built from primitive conditions using boolean connectives. Details of how to interpret these condition expressions is given in Appendix A.

2.4 Defining Events.

The primitive events in PEDL correspond to updates of monitored variables, and calls and returns of monitored methods. Each event has an associated timestamp and may have a tuple of values.

The event `update(x)` is triggered when variable x is assigned a value. The value associated with this event is the new value of `x`. Events `StartM(f)` and `EndM(f)` are triggered when control enters to and return from method `f`, respectively. The value associated with `StartM` is a tuple containing the values of all arguments. The value of an event `EndM` is a tuple that has the return value of the method, along with the values of all the formal parameters at the time control returns from the method. Besides these three, we have one other primitive event which is `IoM(f)`. This is also triggered when control returns from a method `f`, but has as its value a tuple that contains the return value of the method, and the values of the arguments *at the time of*

```
<Monitoring Script> ::= Monscr <IDENTIFIER> <Statements> End

<Statements> ::= ( <Statement> [;] )*
<Statement>  ::=   <Event Declaration>
                 | <Condition Declaration>
                 | <Monitored Object Declaration>
                 | <Monitored Method Declaration>
                 | <Event Definition>
                 | <Condition Definition>

<Event Declaration>     ::= export event <Identifier List>
<Condition Declaration> ::= export condition <Identifier List>
<Identifier List>       ::= <IDENTIFIER> ( , <IDENTIFIER> )*

<Monitored Object Declaration> ::= monobj <Type> <Object Identifier>
                                                 ( , <Object Identifier> )*
<Monitored Method Declaration> ::= monmeth <Result Type> <Method Identifier>

<Event Definition> ::= event <IDENTIFIER> = <Event Expression>
<Event Expression> ::=   <Simple Event Expression>
                       | <Event Expression> && <Event Expression>
                       | <Event Expression> || <Event Expression>
                       | <Event Expression> when <Condition Expression>

<Condition Definition> ::= condition <IDENTIFIER> = <Condition Expression>
<Condition Expression> ::=   <Simple Condition Expression>
                           | defined ( <Condition Expression> )
                           | ! <Condition Expression>
                           | <Condition Expression> && <Condition Expression>
                           | <Condition Expression> || <Condition Expression>


<Simple Event Expression> ::=   <IDENTIFIER>
                              | start ( <Conditional Expression> )
                              | end ( <Conditional Expression> )
                              | update ( <Object Identifier> )
                              | StartM ( <Method Identifier> )
                              | EndM ( <Method Identifier> )
                              | IoM ( <Method Identifier> )

<Simple Condition Expression> ::=   <IDENTIFIER>
                                  | InM ( <Method Identifer> )
                                  | <Boolean Expression>

<Boolean Expression> ::= Java boolean expressions
<Object Identifier>  ::= Java object identifer with class and package names
                         separated by periods
<Method Identifier>  ::= Java method name along with the types of the formal
                         arguments
<Type>               ::= Java Types
<Result Type>        ::= void | <Type>
```

Figure 1: BNF Grammar for PEDL

method invocation. This event allows one to look at the input-output behavior of a method, and is needed if one wants to *program check* some numerical computation. Notice that event `IoM(f)` is the only event to violate our second design principle, namely that the operation of the event recognizer is to be based only on *the current state*.

All the operations on events defined in the logic (see Appendix A) can be used to construct more complex events from these primitive events. In PEDL, we also have two attributes `time` and `value`, defined for events. As mentioned in Section A, events have associated with them attribute values, and the time of their occurrence, and these can be accessed using the attributes `time` and `value`. `time(e)` gives the time of the last occurrence of event `e`, while `value(e)` gives the value associated with `e`, provided `e` occurs. `time(e)` refers to the time on the clock of the monitored system (which may be different from the clock of the monitor) when this event occurs.

3 Meta Event Definition Language (MEDL)

The safety requirements that need to be monitored are written in a language called MEDL. Like PEDL, MEDL is also based on the logic for events and conditions, described in Appendix A. Primitive events and conditions in MEDL scripts are imported from PEDL monitoring scripts; hence the language has the adjective "meta".

3.1 Structure of MEDL script.

The BNF grammar for MEDL is decribed in Figure 3. The requirements are specified between the keywords `Reqspec` and `End`. The list of events and conditions that the checker expects to be informed about are listed in `<Event Declaration>` and `<Condition Declaration>`; this list of event and condition identifiers is specified after the keyword `import`. Apart from these "imported" events and confitions, the MEDL script also contains the definitions of other events and conditions which may be used to define the properties that need to be satisfied. These are described in `<Event Definition>` and `<Condition Definition>`. Auxiliary variables declared after `var` can be used to record certain aspects of the current execution. The safety properties of the system are specified after the keyword `property`, while alarms are described after `alarm`. The final thing in any MEDL script are the rules for the updation of auxiliary variables which are given in `<Guards>`. These are given as guarded commands, triggered by some event.

3.2 Auxiliary Variables.

The logic described is Section A has a limited expressive power. For example, one cannot count the number of occurrences of an event, or talk about the ith occurrence of an event. For this purpose, MEDL allows the user to define auxiliary variables, whose values may then be used to define events and conditions. Auxiliary variables must be of one of the basic types in Java. Updates of auxiliary variables are triggered by events. For example,

```
RaisingGate -> {t = time (RaisingGate);}
```

records the time of occurrence of event `RaisingGate` in the auxiliary variable `t`. Expression

```
e1 -> {count_e1 = count_e1 + 1;}
```

counts occurrences of event `e1`. A special auxiliary variable `currentTime` can be used to refer to the current time of the system. It is set to be the timestamp of the last message received from the filter.

3.3 Defining events and conditions.

The primitive events and conditions in MEDL are those that are defined in PEDL. Besides these, primitive conditions can also be defined by boolean expressions using the auxiliary variables. More complex events and conditions are then built up using the various connectives described in Section A. These events and conditions are then used to define safety properties and alarms.

```
<Requirement Script> ::= Reqscr <IDENTIFIER> <Statements> <Guards> End

<Statements> ::= ( <Statement> [;] )*
<Statement>  ::=   <Event Declaration>
                 | <Condition Declaration>
                 | <Axilliary Variable Declaration>
                 | <Event Definition>
                 | <Condition Definition>
                 | <Safety Property Definition>
                 | <Alarm Definition>

<Guards>  ::= ( <Guard> )*
<Guard>   ::= <IDENTIFIER> -> { <Updates> }
<Updates> ::= <Update> ( ; <Update> )* ;
<Update>  ::= <IDENTIFIER> = <Expression>

<Event Declaration>               ::= import event <Identifier List>
<Condition Declaration>           ::= import condition <Identifier List>
<Auxilliary Variable Declaration> ::= var <Primitive Type> <Identifier List>
<Identifier List>                 ::= <IDENTIFIER> ( , <IDENTIFIER> )*

<Event Definition>     ::= event <IDENTIFIER> = <Event Expression>
<Condition Definition> ::= condition <IDENTIFIER> = <Condition Expression>
<Safety Property Definition>
                       ::= property <IDENTIFIER> = <Condition Expression>
<Alarm Definition>     ::= alarm <IDENTIFIER> = <Event Expression>

<Event Expression> ::=   <Simple Event Expression>
                       | <Event Expression> && <Event Expression>
                       | <Event Expression> || <Event Expression>
                       | <Event Expression> when <Condition Expression>

<Condition Expression> ::=   <Simple Condition Expression>
                           | defined ( <Condition Expression> )
                           | ! <Condition Expression>
                           | <Condition Expression> && <Condition Expression>
                           | <Condition Expression> || <Condition Expression>
                           | [ <Event Expression> , <Event Expression> )

<Simple Event Expression> ::=   <IDENTIFIER>
                              | start ( <Conditional Expression> )
                              | end ( <Conditional Expression> )

<Simple Condition Expression> ::=   <IDENTIFIER>
                                  | <Boolean Expression>

<Boolean Expression> ::= Java boolean expressions
<Expression>         ::= Java expressions
<Primitive Type>     ::= Java primitive types
```

Figure 2: BNF grammar for MEDL

```
class GateController {
    public static final int GATE_UP    = 0;
    public static final int GATE_DOWN  = 1;
    public static final int IN_TRANSIT = 2;
    int gatePosition;
    public void open()  { ... }
    public void close() { ... }
    ...
};
```

Figure 3: Implementation of the gate controller

3.4 Safety Properties and Alarms.

The correctness of the system is described in terms safety properties and alarms. Safety properties are conditions that must *always* be true during the execution. Alarms, on the other hand, are events that must never be raised. Note that all safety properties [MP92] can be described in this way. Also observe that alarms and safety properties are complementary ways of expressing the same thing. The reason we have both of them is because some properties are easier to think of in terms of conditions, while others are easier to think of in terms of alarms.

4 Example

We illustrate the use of PEDL and MEDL using a simple but representative example. The example is inspired by the railroad crossing problem, which is routinely used as an illustration of real-time formalisms [HD96]. The system is composed of a gate that can open and close, taking some time to do it, trains that pass through the crossing, and a controller that is responsible for closing the gate when a train approaches the crossing and opening it after it passes. The common specification approach is to assume an upper bound on the time necessary for the gate to open or close. In reality, however, mechanical malfunctions may result in unexpectedly slow operation of the gate. A timely detection of such a violation lets the train engineer stop the train before it reaches the crossing. In this example, we monitor the controller of the gate, using the requirement that the gate is down within 30 seconds after signal *CloseGate* is sent, unless signal *OpenGate* is sent before the time elapses. Precisely, we check that if there is a signal *CloseGate*, not followed by either signal *OpenGate* or completion of gate closing, is present in the execution trace, then the time elapsed since that signal is less than 30.

Figure 3 shows a fragment of the gate controller implemented as a Java class. The state of the gate is represented as variable `gatePosition`, which can assume constant values `GATE_UP`, `GATE_DOWN`, or `IN_TRANSIT`. The controller controls the gate by means of methods `open()` and `close()`. For simplicity, we assume that there is only one instance of class `GateController` in the system.

We need to observe calls to methods `open()` and `close()`, and the state of the gate. The following PEDL script introduces high-level events `OpenGate`, `CloseGate` and condition `Gate_Down`.

```
Monscr RRCPedl

  export event OpenGate, CloseGate;
  export condition Gate_Down;

//Monitored Methods
  monmeth void GateController.open();
  monmeth void GateController.close();

//Monitored variables
  monent int  GateController.gatePosition;
```

```
//Definition of conditions
  condition Gate_Down =
       (GateController.gatePosition ==
        GateController.GATE_DOWN);

//Definition of Events
  event OpenGate =
       StartM(GateController.open());
  event CloseGate =
       StartM(GateController.close());

End
```

The correctness requirement for the gate is given in the MEDL script below. The time of the last occurrence of event `CloseGate` is recorded by the auxiliary variable `lastClose`. The requirement uses the events and conditions imported from the monitoring script and states that if there was a `CloseGate` event at the time when the gate was not down, which was not followed by either event `OpenGate` or condition `Gate_Down` becoming true, then the time allotted for gate closing has not elapsed yet.

```
Reqspec RRCMedl

  import event OpenGate, CloseGate;
  import condition Gate_Down;

//Declaration of auxilliary variables
  var float lastClose;
  var float currentTime;

//Safety properties
  property GateClosing =
         [ CloseGate when !Gate_Down,
           OpenGate || start(Gate_Down) )
         => lastClose + 30 > currentTime;

//Rules for updating auxilliary variables
  CloseGate ->
         {lastClose = time(CloseGate); }

End
```

References

[HD96] C. Heitmeyer and D. Mandrioli, *Eds. Formal Methods for Real-Time Systems.* Number 5 in Trends in Software. John Wiley & Sons, 1996.

[KVBA+98] Moonjoo Kim, Mahesh Viswanathan, Hanêne Ben-Abdallah, Sampath Kannan, Insup Lee, and Oleg Sokolsky. A framework for run-time correctness assurance of real-time systems. Technical Report MS-CIS-98-37, University of Pennsylvania, 1998.

[KVBA+99] Moonjoo Kim, Mahesh Viswanathan, Hanêne Ben-Abdallah, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Formally specified monitoring of temporal properties. In *To appear at ECRTS'99*, 1999.

[LKK+99] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M.Viswanathan. Runtime assurance based on formal specifications. In *To appear at PDPTA'99*, 1999.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

[c_k primitive]	$\mathcal{D}^t_M(c_k) = L_C(s_i, c_k)$, where $\tau(s_i) \leq t$ and for all s_j $(j > i)$ $\tau(s_j) > t$	
[defined]	$\mathcal{D}^t_M(\text{defined}(c)) = \begin{cases} \textbf{true} & \text{if } \mathcal{D}^t_M(c) \neq \Lambda \\ \textbf{false} & \text{otherwise} \end{cases}$	
[pair]	$\mathcal{D}^t_M([e_1, e_2]) = \begin{cases} \textbf{true} & \text{if there exists } t_0 \leq t \text{ such that } M, t_0 \models e_1 \text{ and for all} \\ & t_0 \leq t' \leq t,\ M, t' \not\models e_2 \\ \textbf{false} & \text{otherwise} \end{cases}$	
[negation]	$\mathcal{D}^t_M(!c) = \begin{cases} \textbf{true} & \text{if } \mathcal{D}^t_M(c) = \textbf{false} \\ \Lambda & \text{if } \mathcal{D}^t_M(c) = \Lambda \\ \textbf{false} & \text{if } \mathcal{D}^t_M(c) = \textbf{true} \end{cases}$	
[disjunction]	$\mathcal{D}^t_M(c_1 \| c_2) = \begin{cases} \textbf{true} & \text{if } \mathcal{D}^t_M(c_1) \text{ or } \mathcal{D}^t_M(c_2) \text{ is } \textbf{true} \\ \textbf{false} & \text{if } \mathcal{D}^t_M(c_1) = \mathcal{D}^t_M(c_2) = \textbf{false} \\ \Lambda & \text{otherwise} \end{cases}$	
[conjunction]	$\mathcal{D}^t_M(c_1 \&\& c_2) = \mathcal{D}^t_M(!(!c_1 \| !c_2))$	
[implication]	$\mathcal{D}^t_M(c_1 \Rightarrow c_2) = \mathcal{D}^t_M(!c_1 \| c_2)$	

Figure 4: Denotation for conditions

A Logic for Events & Conditions

Syntax. We assume a countable set $\mathcal{C} = \{c_1, c_2, \ldots\}$ of primitive conditions. For example, in the monitoring script language (Section 2), these primitive conditions will be Java boolean expressions built from the values of the monitored variables. In the requirements description language (Section 3) these will be conditions that were recognized by the event recognizer and sent to the run-time checker.

We also assume a countable set $\mathcal{E} = \{e_1, e_2, \ldots\}$ of primitive events. When an event occurs, it can have an attribute value, which is an element of a set $\mathcal{S}_{e_i}$. For example, `StartM(RaiseGate)` is a primitive event in the monitoring script language, which is present at the start of method `RaiseGate` and whose attribute value is the tuple of values of all the parameters with which this method is called. The primitive events in the requirements description language are those that are reported by the event recognizer.

The logic has two sorts: conditions and events. The syntax of conditions (C) and events (E) is as follows:

$$\begin{array}{lll} \langle C \rangle & ::= & c \mid \text{defined}(\ \langle C \rangle\ ) \mid [\ \langle E \rangle\ ,\ \langle E \rangle\ ) \\ & & \mid\ !\ \langle C \rangle \mid \langle C \rangle\ \&\&\ \langle C \rangle \mid \langle C \rangle\ \|\ \langle C \rangle \\ & & \mid \langle C \rangle \Rightarrow \langle C \rangle \\ \langle E \rangle & ::= & e \mid \text{start}(\ \langle C \rangle\ ) \mid \text{end}(\ \langle C \rangle\ ) \\ & & \mid \langle E \rangle\ \&\&\ \langle E \rangle \mid \langle E \rangle\ \|\ \langle E \rangle \\ & & \mid \langle E \rangle \text{ when } \langle C \rangle \end{array}$$

Semantics. The models for this logic are sequences of worlds, similar to those used for linear temporal logic. Each world has a description of the truth values of primitive conditions and occurrences of primitive events. More formally, a model M is a tuple (S, τ, L_C, L_E), where $S = \{s_0, s_1, \ldots\}$, τ is a mapping from S to the time domain (which could be integers, rationals, or reals), L_C is a total function from $S \times \mathcal{C}$ to $\{\textbf{true}, \textbf{false}, \Lambda\}$, and L_E is a partial function from $S \times \mathcal{E}$ to $\mathcal{D}_e$. Intuitively, L_C assigns to each state the truth values of all the primitive conditions; since we interpret conditions over a 3-valued logic, the truth value of primitive conditions can be **true**, **false** or Λ (undefined). Similarly, in each state s, $L_E(s, e)$ is defined for each event e that occurs at s and gives the value of the primitive event e. The mapping τ defines the time at each state, and it satisfies the requirement that $\tau(s_i) < \tau(s_j)$ for all $i < j$, i.e., the time at a later state is greater.

In order to define what we mean by a condition c being true in model M at time t ($M, t \models c$), we need to define what we mean by its denotation ($\mathcal{D}^t_M(c)$). This is defined in Figure 4. Using this we define the meaning of $M, t \models c$, and of an event e occurring in a model M at time t ($M, t \models e$). The formal definition is given in Figure 5[1].

[1] Notice, that the definition of $\mathcal{D}^t_M$ refers to the definition of $\models$, and vice versa. However, the definitions are well-defined.

$M, t \models c$	iff	$\mathcal{D}_M^t(c) = \mathbf{true}$
$M, t \models e_k$ (e_k primitive)	iff	there exists state s_i such that $\tau(s_i) = t$ and $L_E(s_i, e_k)$ is defined.
$M, t \models \text{start}(c)$	iff	$\exists s_i$ such that $\tau(s_i) = t$ and $M, \tau(s_i) \models c$ and $M, \tau(s_{i-1}) \not\models c$. i.e., start($c$) occurs when condition c changes from false or undefined, to true.
$M, t \models \text{end}(c)$	iff	$\exists s_i$ such that $\tau(s_i) = t$ and $M, \tau(s_i) \models !c$ and $M, \tau(s_{i-1}) \not\models !c$. i.e., end($c$) occurs when condition c changes from true or undefined, to false.
$M, t \models e_1 \vert\vert e_2$	iff	$M, t \models e_1$ or $M, t \models e_2$.
$M, t \models e_1$ && e_2	iff	$M, t \models e_1$ and $M, t \models e_2$.
$M, t \models e$ when c	iff	$M, t \models e$ and $M, t \models c$. i.e., event e occurs when condition c is true.

Figure 5: Semantics of events and conditions.

As stated before, we interpret conditions over three values, **true**, **false**, and Λ (undefined). The denotation of a primitive condition, c at time t is given by c's truth value in the last state before time t. The predicate defined(c) is true whenever the condition c has a well-defined value, namely, **true** or **false**. The denotation of negation ($!c$), disjunction ($c_1||c_2$) and conjunction ($c_1\&\&c_2$) are interpreted classically whenever c, c_1 and c_2 take values **true** or **false**; the only non-standard cases are when these take the value Λ. In these cases, we interpret them as follows. Negation of an undefined condition is Λ. Conjunction of an undefined condition with **false** is **false**, and with **true** is Λ. Disjunction is defined dually; disjunction of undefined condition and **true** is **true**, while disjunction of undefined condition and **false** is Λ. Implication ($c_1 \Rightarrow c_2$) is taken to $!c_1||c_2$.

For primitive events, once again, the truth value is given by the labels on the states. Conjunction ($e_1\&\&e_2$) and disjunction ($e_1||e_2$) defined classically; so $e_1\&\&e_2$ is present only when both e_1 and e_2 are present, whereas $e_1||e_2$ is present when either e_1 or e_2 is present.

There are some natural events associated with conditions, namely, the instant when the condition becomes **true** (start(c)), and the instant when the condition becomes **false** (end(c)). Notice, that the event corresponding to the instant when the condition becomes Λ can be described as end(defined(c)). Also, any pair of events define an interval of time, so forms a condition $[e_1, e_2)$ that is **true** from event e_1 *until* event e_2. Finally, the event (e when c) is present if e occurs at a time when condition c is **true**.

Notice that every condition can be identified with the events corresponding to when it becomes **true**, when it becomes **false** and when it becomes Λ. This is the reason why the languages in the MaC framework, are called "event definition languages".