SMEDL and PEDL Language Definition

UPenn

Feb 2016

1 Overview of SMEDL and PEDL

SMEDL is a language specifying the runtime monitors of the system. The central concept of SMEDL is a monitoring object, or object for short. An object can be an abstraction of a system object, whose state is tracked by the monitoring system. Alternatively, an object can be an abstract entity that represents interactions between multiple system objects. An object can have a state, which reflects history of its evolution. Objects can be created dynamically during an execution of a system.

As the system evolves, multiple instances of the same object may be created. To distinguish between instances, a part of the object state is designated as the object identity and remains immutable during the object's lifetime. Objects in SMEDL can have their state changed by event arrivals.

The second concept of SMEDL is an event. SMEDL events are instantaneous occurrences that ultimately originate from observations of the system execution. Events are decorated with a set of attributes, which are formal parameters of the event. An event occurrence carries a set of actual parameters, which are values assigned to formal parameters of the event.

At runtime, each object instance produces a separate checker. Events are delivered to the checkers corresponding to objects that match the values of event parameters. Thus, a single event can be processed by multiple checkers.

Each SMEDL object includes an interface, which declares events that are associated with this object, object state, which can include references to other objects, and the specification of object behaviors. We adopt a scenario-based behavior specification style, where each scenario describes a partial behavior, and the complete behavior is obtained by composing scenarios.

This document will describe the syntax and semantics of SMEDL and PEDL.

2 Syntax Description of SMEDL

2.1 Basics

2.1.1 Lexical elements

- **Identifiers**. The identifiers are sequences of characters representing type, variable and other names in the SMEDL(also the PEDL) specification. The legal characters of identifiers are digit from 0 to 9, alphabets from a to z and A to Z and underscore '_'. Identifiers are case sensitive and the first character of identifiers are alphabet only.The BNF representation of Identifiers are shown below.

  identifier ::= `[a-zA-Z][A-Za-z0-9_]*`

- **Numbers**. In SMEDL, float and integer numbers are supported, which conform the rule in Java and C. The BNF representation of Numbers are shown below.

  integer ::= `[-+]?[0-9]+`
  float ::= `[-+]?[0-9]*.?[0-9]+`

- **keywords**. The keywords are defined below. The meaning of key words will be introduced in detail in the following sections.

```
#import     object    identity     state
events     internal   exported   imported
true        false       null      atomic
scenarios   error       when
```

- **Special tokens** The following strings are defined as special tokens:

```
:    ;    (    )    [   ]
{    }    ,    .    +   -
*    /    &&   ||   >   <
>=   <=   ==   !=   !
```

2.1.2 Types

Primitive types are integers, floating point numbers, strings, and container objects such as sets. The opaque type is a primitive type, on which only the equality operation is defined. Composite types are either objects defined in

SMEDL or Java objects treated as data types. Methods of Java objects can be used as operations in expressions.

2.1.3 Term and Expression

Term is the syntax element forming the expressions. In SMEDL, there are three kinds of term: atomic term, term with trailer and expressions rounded by parentheses. Atomic term includes integer number, float number, literal value such as 'true', 'false' and 'null' and identifiers. Identifier can be trailed with brackets, parentheses and dot plus identifier which can respectively correspond to the array, function call and reference to a variable in the function. As for expressions,arithmetic expressions, relational and logical expressions are supported in SMEDL.

```
term ::= ('+'|'-'|'~'|'!')* atom{trailer}
|('+'|'-'|' ~')* '(' arith_expr ')'
atom ::= integer | float | identifier | 'true' |
'false' | 'null'
expression ::= and_expr {'||' and_expr}+ | and_expr
and_expr ::= sub_expr {'&&' sub_expr}+ | sub_expr
sub_expr ::= '!(' expression ')' | '(' expression ')'
| comp_expr
comp_expr ::= arith_expr {('>' | '<' | '>=' | '<=' |
'==' | '!=') arith_expr}+ | '(' comp_expr ')' |
arith_expr
arith_expr ::= term {('+' | '-' | '*' | '/' | '%')
term}+ | term
trailer ::= '[' [expression] ']' | '('
[expression_list] ')' | '.' identifier {trailer}*
expression_list ::= expression {',' expression}* | ()
```

2.2 Monitoring Object Definitions

Object definitions are given by the following grammar. Scenario definitions, referenced in the grammar, are discussed below. The optional import part is similar to the import in Java. Files containing the helper functions are needed to be imported into the SMEDL specification so that these functions can be called. Each object contains a set of identity parameters, a set of state variables, partitioned into object identity and data state, a non-empty set of events that are delivered to the object, and a non-empty set of scenarios. The set of state variables can be empty. If identity parameters are absent, the object is static and cannot be explicitly instantiated. Static objects have only one instance.

```
object ::=
{'#import' identifier}*
'object' identifier
  ['identity' parameter_declaration+]
  ['state' variable_declaration+]
  'events'
    { 'internal' event_definition_list
    | 'exported' event_definition_list}*
    'imported' event_definition_list
    { 'imported' event_definition_list
    | 'internal' event_definition_list
    | 'exported' event_definition_list]}*
  'scenarios'
    {scenario_definition}+
parameter_declaration ::= type identifier {',' identifier}* ';'
variable_declaration ::= type identifier {',' identifier}* ';'
event_definition ::= ['error'] identifier ['(' parameter_list ')']
parameter_list ::= type {',' type}* | ()
```

Variable declarations for identity and state variable are lists of typed identifiers. Event definitions are partitioned into imported, exported and internal events, and can be decorated with lists of typed parameters. The set of imported events in an object cannot be empty. Otherwise, the object cannot be used to monitor system behaviors, because it would not have any inputs to observe. Exported events are generated by the object itself. The set of declared exported events can be empty. Internal events are events which cannot be observed nor exported by the monitor. Moreover, each object has a default exported event that represents a monitoring violation. Declared exported events can be used as means of supplying more detailed information about the observed behavior, or as means of coordination between monitors of different SMEDL objects.

Note that the error qualifier can be used only in the exported list of event definitions. That is, error events are never observed, but are always produced by a monitor. Assignments in an action cannot modify the values of identity parameters in an object instance.

2.3 Scenario

Each monitor can specify multiple scenarios and in each scenario, multiple event traces can be defined. The definition related to the scenario is given below. In each trace, the first identifier and the last identifier respectively correspond to the initial and the final state of the trace. Within the trace, a sequence of event is defined and each of event is divided by '→'. Intuitively, a scenario

waits for the specified event to arrive. When it arrives, scenario execution is triggered. An event may be followed by an action that changes the values of state variables of the object and can raise events. The type of action includes the update of state variables, the raise of an event, the instantiation of another monitor object or the call of a helper function defined in the code. If present, the optional action is executed, and the scenario moves into the state where it waits for the next event. Moreover, the condition can be specified on the event by the "when" expression. In each trace, an "else" branch can be specified. If any of the "when" expression in the trace is evaluated to false, the trace will transit into the "else" branch.

States in a scenario may be named, or they may be anonymous. Named states are used in the composition within a scenario. Scenario definitions allow us to specify two events immediately following each other, without a named state in between. This is a notational convenience that allows us to avoid naming states that we are not going to refer to. In every such case, we syntactically introduce a unique default state into the scenario definition.

A scenario may be designated atomic, meaning that the object cannot engage in any other scenario until the end of the sequence of events in the scenario is reached. Otherwise, scenarios are interleaved, in that an incoming event may trigger execution of a scenario while another scenario may be partially executed. An event may be used in multiple scenarios. When such an event arrives, all scenarios that are ready to accept this event are triggered.

```
scenario_definition ::= ['atomic'] identifier ':'
{trace_definition}+
trace_definition ::= identifier '->' {step_definition
'->'}+ identifier ['else' [action] '->' identifier]
step_definition ::= event_instance [action]
event_instance ::= expression ['when' expression]
action ::= '{' nonempty_action_item_list '}'
nonempty_action_item_list ::= action_item {';' action_item}*

action_item ::= state_update | raise_stmt |
instantiation_stmt | call_stmt
state_update ::= target ('++' | '--')| target '='
expression
target ::= (* To be improved *) identifier
raise_stmt ::= 'raise' identifier ['(' expression_list
')']
instantiation_stmt ::= 'new' identifier ['('
state_update_list ')']
state_update_list ::= state_update {',' state_update}* | ()

call_stmt ::= target '(' expression_list ')'
```

3 Syntax Description of PEDL

PEDL is used to describe the instantiation of the checkers and events of the lowest level(raised by the instrumented code when the monitored program is being executed). The basic definition of PEDL syntax such as lexical elements, types, term and expressions are the same as SMEDL.

3.1 Object Definitions

The object definitions are given by the following grammar. Each PEDL specification can contain multiple monitor and event instantiations. For the instantiation of monitors, all parameters needs to be assigned with initial values, shown in the *structure* and *field* part. The instantiation of events are shown in the following section.

```
object ::= 'object' identifier
           'events'
             {new_mon}+
             {event_def}+
new_mon ::= 'new' identifier ['(' identifier_list
            ')'] '=' 'create' structure
structure ::= identifier '{' field* '}'
field ::= identifier '=' expression ';'
```

3.2 Events

The grammar of event instantiations are shown below. Note that the elements "state_update_list","expression_list","state_update","raise_stmt", "nonempty_action_item_list", "instantiation_stmt" are the same as in the SMEDL specification. The first identifier represents the monitor which receives the events and the following identifier_list contains the actual parameter of the monitor. The second identifier represents the name of the event, followed by the state update list which initializes the parameter of the events. In PEDL, the event can be triggered by the update of certain values or the calling of certain functions. The expression after 'when' represents the condition under which the event can occur.

```
event_def ::= identifier '(' identifier_list
').'identifier '(' [state_update_list] ')' '='
{'update(' expression_list ')' | 'call('
expression_list ')'} ['when' expression]
```

4 Abstract Syntax and Semantics of SMEDL and PEDL

This section describes the abstract syntax of SMEDL and PEDL, based on which the informal semantics of SMEDL will be described.

4.1 Abstract Syntax of SMEDL

object ::= <name, identityParameters, stateVariables,
eventDeclarations, scenarios>.
identityParameters :: = Set of typed parameters.
stateVariables :: = Set of typed variables.
eventDeclarations ::= <importedEvents, exportedEvents,
internalEvents>.
importedEvents ::= Set of eventSig.
exportedEvents ::= Set of eventSig.
internalEvents ::= Set of eventSig.
eventSig ::= <name, parameterTypes>.
parameterTypes ::= List of types.
scenarios ::= Set of ScenarioDefinition.

ScenarioDefinition ::= <name, atomic, states, initialState,
triggeringEvents, traces,errorState>.
atomic ::= {true,false}.
states ::= Set of state.
state ::= name.
triggeringEvents ::= Set of event.
event ::= <name, parameters>.
parameters ::= List of variables.
traces ::= <transitions, elseTerm>.
transitions ::= set of transition.
transition ::= <originState, TargetState, event, condition, actions>.
originState ::= state.
targetState ::= state.
condition ::= bool expression.
actions ::= <StateVariableUpdate, HelpFunctionCalls, RaisingEvents>.
StateVariableUpdate ::= $stateVariables \rightarrow values$.
RaisingEvents ::= <outstandingEvents, ExportedEvents>.
elseTerm ::= <actions, elseState>.

4.2 Semantics for Single Non-parametric monitor

4.2.1 Informal Description

The monitoring object (object) is responsible for monitoring systems. It is a 5-tuple <name, identityParameters, stateVariables, eventDeclarations, scenarios> The name represents the type of the object which means that there could be multiple monitoring object instances created in the system. The identityParameters is the set of parameters that identify the instance of the object. If identityParameters is empty, the object is static which means there can be only one instance of object in the system. Note that once parameters are assigned in the instantiation, their values cannot be changed. The stateVariables is the set of variables belonging to the object which can be used and updated during the execution of the monitoring.

EventDeclarations is to specify the events which are imported, exported or internally used. There are three kinds of events. The imported events are the events raised by the code instrumented in the monitored programs or other monitors, which are mainly used for the triggering of the transition in scenarios. The exported events are the events to be triggered in the object, which can then be imported by other monitors. The internal events can be only seen within the object so that the use or the triggering both happen in the scenario of this object. The event definition is a two-tuple of <name, parameterTypes> where name is the identity and parameterTypes is a list of types of the parameters of the event. In SMEDL, there are three ways of triggering the event. First two are from the instrumented code. One is the update of a variable in the program. The other is the calling of a function in the code. The third way is to "raise" it during the execution of the monitor.

In the context of SMEDL, "the execution of the monitor" means the execution of scenarios in the object. There can be multiple scenarios defined in the object, each one of which is like a state machine, defined as a eight-tuple of <name,atomic,states,triggeringEvents,traces, errorState>. Name is the identity of the scenario. Atomic is a property of the scenario, indicating that if the scenario is atomic, the execution of the scenario cannot be interrupted by other scenarios. A set of states are defined in the scenario in which the initialState is the state the scenario is in at the beginning of the execution. The triggeringEvents are to used to trigger the transition between the states. The traces is a set of traces. Each trace contains a set of transitions and an "elseTerm". The states in each transition need to be in the states of the scenario. The transition has a orginState, a targetState, an event that triggers the transition. Moreover, there may be a condition that enables the transition. Actions of the transition can be the calling of helper functions, the update of the state variables or the raising of events. "ElseTerm" is used to deal with the situation that the event e can trigger the event but the corresponding condition is not satisfied. For each trace, if any of the transition to be triggered is not satisfied the condition, this scenario will transit into the state defined in the "elseTerm" and execute the actions. "ErrorState" is an implicit state for each scenario. There are two kinds

of error in SMEDL. One is explicit error specified in the scenario by the user. Another error is implicit such as the unexpected order of event arriving. The SMEDL runtime will transit the current scenario into the errorState.

The actions is a set of action statements to be executed after the event is triggered. There are three kinds of statements:1) update of a state variable; 2) call a helper function; 3) raise of an event. Usually, events only used within the object are defined as internal events while events which can be used by other monitors will be defined as exported events(an exported event can be used to trigger the transition within the same monitor instance and when it happens, the raised exported event will be seen as the internal event of this monitor instance). Here we only consider the raise of internal event. SMEDL uses the composition transition for describing the execution of multiple scenarios. A composite transition is a collection of scenario transitions, at most one from each scenario, starting from the current state of each scenario. The initial transition is triggered by an external event. One or multiple scenario can receive this event and begin executing the corresponding actions defined in the event. Within the actions, there may exist several raise of internal events. The composite transition can be obtained by the calculation of the transitive closure of transitions receiving internal events. Note that it is not allowed that transitive closure contains more than one transition per scenario. Other actions such as update of state variables or calls of helper functions in the composite transition are independent. When all actions finish execution, scenarios involved will simultaneously transit into the new state. Moreover, when an event can be triggered in the scenario but the current state of that scenario is not the origin state of that transition, an implicit error will happen due to the possibility of the out of order of the event arrivals.

4.2.2 Formal Description

This section will introduce the semantics of a single monitor in a formal way. The notations in the abstract syntax will be used and some new notations will be introduced first. Note that to introduce an element of a tuple, the "." notation is used. For instance, to infer all exported events of a monitor A, the notation "object(A).eventDeclarations.exportedEvents" is used.

1. Some notations

- $AllStates_A$: the set of all states in the monitor A. Without losing the generality, it is assumed that states in different scenarios have different names. Note that if an "elseTerm" is defined in a trace, the corresponding "elseState" will be in the set.
- $Events_A$ denotes the set of all events in the monitor A.
- $ExpEv_A$ denotes the set of events that will be exported by the monitor A.

- $InterEv_A$: outstanding events that can be triggered in the monitor A. Note that events in $ExpEv_A$ may also be in $InterEv_A$.
- $StateVars_A$ denotes the set of state variables in the monitor A.
- Sce_A denotes the set of scenarios in the monitor A.
- $SceState_A$: $Sce_A \rightarrow AllStates_A$. $SceState_A$ denotes the current mapping from scenarios to corresponding states of the scenario of the monitor A.
- $DataState_A$: $Vars_A \rightarrow V$. $DataState_A$ denotes the current mapping from the state variables to corresponding values in the monitor A. Note that variables of different types have different ranges and V represents the set of all possible values.
- $Trans_A$: $AllStates_A \times Events_A \rightarrow_{(C,A)} AllStates$. $Trans_A$ denotes the set of all transition in the monitor A where (C, A) represents the condition(C) of the transition and the set of actions(A) to be executed of the transition.
- $SceEvent_A$: $Sce_A \rightarrow 2^{Events_A}$. $SceEvent_A$ denotes the mapping between the scenario and the set of outstanding events in that scenario.
- $Traces_A$ denotes the set of traces in the monitor A.
- $SceTra_A$: $Sce_A \rightarrow 2^{Traces_A}$. $SceTra_A$ denotes the mapping relation between the scenario and the set of traces in that scenario.
- $TracePart_A$: $Trans_A \rightarrow Traces_A$. $TracePart_A$ is the mapping relation of the transition and corresponding trace that the transition is in. Note that if a transition t is in a trace that has "elseTerm", a transition from $t.target$ to the corresponding "elseState" will be added where the trigger event is the trigger event of t, condition is the logic opposite of the condition of t and the actions are defined in the "elseTerm".

2. Configurations and Runtime Environment

To formally describe the semantics of the execution of the monitor, $Configuration$ is introduced first. Configuration is a four-tuple $< SceState, DataState, Ev, Sc >$ where $SceState$ illustrates which state each scenario is in; $DataState$ represents the valuation of each state variables; Ev is the set of events raised by the monitor(including the exported and internal events; Sc is the set of scenarios that have executed the transition during the execution. The execution and interaction of the monitors are driven by the triggering of events, which is controlled by the SMEDL runtime environment. In the following part, it is assumed that the delivering of the event to the corresponding scenario is done by the runtime environment.

3. Transition Rules

The transition of the monitor is triggered by the imported events defined in the monitor. Before each transition, the configuration of the monitor takes the form:

$$Conf =< SceState, DataState, \emptyset, \emptyset >$$

After an imported event e happens, the environment deliver it to the monitor:

$$Conf =< SceState, DataState, \{e\}, \emptyset, >$$

The triggering of e will drive the execution of scenario. All legal rules are defined below

a. normal rule

$$\frac{tr{:}s1\xrightarrow{e}s2, e\in Conf.E, tr\in Trans, \exists sc1, SceState(sc1){=}s1\wedge sc1\notin Conf.Sc, tr.condition{=}true}{<SceState[sc1{=}s1], DataState, Ev\cup\{e\}, Sc>\xrightarrow{e}<SceState[sc1{=}s2], DataState', Ev\cup Ev', Sc\cup\{sc1\}>}$$

The premise of applying the normal rule includes: 1) the triggering event e is in the $Conf.E$; 2) the transition tr triggered by e is in $Trans$; 3) There exists a scenario $sc1$ that the current state of $sc1$ is the origin state of tr; 4) $sc1$ is not in $Conf.Sc$; 3) The condition of the transition is satisfied. Satisfying all the conditions given above, the configuration $Conf$ will transit via e. The update of configuration includes(suppose the updated configuration is denoted as $Conf'$): 1) the state of $sc1$ transit to the target state of the transition and $sc1$ is put into $Conf'.Sc$; 2) $DataState$ updates according to the actions in the transition; 3) e is removed from $Conf.Ev$ and outstanding events that have been raised in the actions of the transition is added to the $Conf'.Ev$. Note that for all traces that have "elseTerm", corresponding transitions have been added to $Trans$, these transition are seen as the normal transition with the target state of $elseState$. On the other hand, if there is no "elseTerm" defined, the corresponding scenario will transit into the implicit $errorState$. The fail rule is defined below.

b. fail rule without elseTerm

$$\frac{tr{:}s1\xrightarrow{e}s2, e\in Conf.E, tr\in Trans, \exists sc1, SceState(sc1){=}s1\wedge sc1\notin Conf.Sc, tr.condition{=}false, TracePart(tr).elseTerm{=}null}{<SceState[sc1{=}s1], DataState, Ev\cup\{e\}, Sc>\xrightarrow{e}<SceState[sc1{=}errorState], DataState, Ev, Sc\cup\{sc1\}>}$$

By applying different rules, a configuration can have multiple possibilities to evolve. If two possibilities are compatible with each other, they can then be merged into one. The following rules involve the merge of different configurations.

c. merge rule

$$\frac{Conf\xrightarrow{e}Conf', Conf\xrightarrow{e}Conf''}{Conf\xrightarrow{e}(Conf'\diamond Conf'')\backslash Conf}\ Conf' \text{ and } Conf'' \text{ are compatible}$$

The operator $\diamond$ is used to merge the two possible update of the configuration. The $Conf$ after the backslash denotes the origin configuration that may be updated to the configuration $Conf'$ and $Conf''$ by applying the rule. To apply for the rule, the two possible updates of the configuration should be "compatible" with each other. Here the definition of compatibility between two configurations are give below. Note that the premise of compatibility is that two configurations should be derived by the same origin configuration and they all have the same domain with the origin configuration.

Definition 1: Compatibility. For two configuration $Conf_1$ and $Conf_2$ under the orign configuration $Conf$, they are *compatible* with each other if the following conditions are satisfied:

- $\forall sc \in Sce$, $Conf_1.SceState(sc)$ and $Conf_2.SceState(sc)$ satisfies at least one of the following conditions:
  1) $Conf_1.SceState(sc) = Conf_2.SceState(sc)$
  2) $Conf_1.SceState(sc) = Conf.SceState(sc)$
  3) $Conf_2.SceState(sc) = Conf.SceState(sc)$

- $\forall v \in StateVars$, $Conf_1.DataState(sc)$ and $Conf_2.DataState(sc)$ satisfies at least one of the following conditions:
  1) $Conf_1.DataState(v) = Conf_2.DataState(v)$
  2) $Conf_1.DataState(v) = Conf.DataState(v)$
  3) $Conf_2.DataState(v) = Conf.DataState(v)$

If $Conf'$ and $Conf''$ are compatible with each other given the definition above, they can be merged using the operator $\diamond$ under the origin configuration $Conf$. Here the definition of the operator $\diamond$ is given below.

Definition 2: Assuming the merge result is denoted as $Conf'$, the merge of $Conf_1$ and $Conf_2$ under the origin configuration $Conf$ follows the following rules:

- $\forall sc \in Sce$,

$$Conf'.SceState(sc) = \begin{cases} Conf.SceState(sc),\ if\ Conf_1.SceState(sc) = Conf_2.SceState(sc) \\ \qquad\qquad\qquad\qquad\qquad\qquad = Conf.SceState(sc) \\ Conf_1.SceState(sc),\ if\ Conf_1.SceState(sc) \neq Conf.SceState(sc) \\ Conf_2.SceState(sc),\ if\ Conf_2.SceState(sc) \neq Conf.SceState(sc) \end{cases}$$

- $\forall v \in StateVars$,

$$Conf'.DataState(v) = \begin{cases} Conf.DataState(v),\ if\ Conf_1.DataState(v) = Conf_2.DataState(v) \\ \qquad\qquad\qquad\qquad\qquad\qquad = Conf.DataState(v) \\ Conf_1.DataState(v),\ if\ Conf_1.DataState(v) \neq Conf.DataState(v) \\ Conf_2.DataState(v),\ if\ Conf_2.DataState(v) \neq Conf.DataState(v) \end{cases}$$

- $Conf'.Ev$=$Conf_1.Ev \cup Conf_2.Ev$

- $Conf'.Sc{=}Conf_1.Sc \cup Conf_2.Sc$

d. merge error rule

$\frac{Conf \xrightarrow{e} Conf', Conf \xrightarrow{e} Conf''}{Conf \xrightarrow{e} ImplicitError}$ $Conf'$ and $Conf''$ are not compatible

$$\frac{Conf \xrightarrow{e} Conf', Conf \xrightarrow{e} ImplicitError}{Conf \xrightarrow{e} ImplicitError}$$

The rules given above indicates that if there exists two configuration possibilities that are not compatible with each other, the monitor will transit into the state "ImplicitError".

After an imported event is dispatched to the monitor, the configuration will be updated by repeatedly applying the rules given above, until no further rule is available. The operation to the configuration after the transition is shown below. Note that the $Conf$ represents the update configuration.

- $\forall ev \in Conf.Ev$, if $ev \notin InterEv$, the transition succeeded and $Conf.Ev$ and $Conf.Sc$ are set to $\emptyset$ for the next cycle of execution.
- If $\exists ev \in Conf.Ev \wedge InterEv, \forall sce$, if $ev \in SceEvent(sce), Conf.SceState(sce) = errorState$
- $Conf = ImplicitError$

4.3 Semantics of Event Deliveries among Multiple Monitors

4.4 Semantics for Parametric Monitors

4.5 PEDL

object ::= <name, monitorInstantiations, eventInstantiations >
monitorInstantiations ::= Set of monitorInstantiation.
eventInstantiations ::= Set of eventInstantiation.
monitorInstantiation ::= <name, parametersOfMonitor, structure >.
parametersOfMonitor ::= List of identityParameters.
structure ::= <name, initializations>.
initializations ::= List of assignment.
assignment ::= <identifier, expression>.

eventInstantiation ::= <monitorInstance, event>.
monitorInstance ::= <name,parametersOfMonitor>.
event ::= <name, parameterIntializationOfEvent, triggerCondition, valueCondition>.
parameterIntializationOfEvent ::= List of assignment.
triggerCondition = update of a variable — calling of a function.
valueCondition ::= bool expression.

PEDL specification is used for the guide of instrument of the target system (manually or automatically). One PEDL object can contain a set of monitor instantiations and a set of event instantiations. In the 3-tuple of instantiation of monitor, parametersOfMonitor needs to correspond to the identity parameters in the SMEDL specification. The structure contains the name of this instance, the initialization of identity parameters and state variables of the monitor.

Event instantiation is determined by the event and the instance of monitor that imports the event. To determine which monitor instance will be import the event, the name and the parameters of the monitor are needed, as shown in the tuple of monitorInstance. The instantiation of event needs the actual parameters of the event, the triggering condition and the value condition. Usually, the parameters of the event will be instantiated from the variables of the target program or the calculation on those variables. The triggering condition is the update of a variable or the function call of the target program. The value condition is a boolean expression over the variables of the target program.