# Formal Semantics of SMEDL

Teng Zhang

University of Pennsylvania, Philadelphia PA 19104, USA,
{tengz}@seas.upenn.edu

**Abstract.** SMEDL is a specification language for hybrid runtime verification(RV). The central concept of SMEDL is a monitoring object which can be an abstraction of a system object or an abstract entity that represents interactions between multiple system objects. Each monitor specification contains a set of EFSMs with shared state variables. The monitor receives events from the target program by instrumentation or from other monitors which trigger the transitions of state machines within the monitor and then produces events to other monitors. A tool has been implemented using Python to generate C code from the SMEDL specification [1]. However, to guarantee the correctness of this code, it is desirable to formalize the semantics of SMEDL. This report proposes an operational semantics for the single monitor of SMEDL, which has been encoded in the Coq platform. Based on the semantics, several interesting properties such as the termination and determinism are defined and proved in Coq.

**Keywords:** Runtime Verification, SMEDL, Formal Semantics, Coq

## 1 Introduction

As software systems become increasingly complicated, ensuring their reliability has emerged as a major research topic. Runtime verification(RV) [2] is a lightweight but powerful verification technique for correctness monitoring of critical systems. The objective of RV is to check if a run of the system satisfies or violates certain properties. A run can be abstracted as a finite or infinite sequence of events either from the actual execution of the system or the logging information. Designed and implemented based on the behaviors of the system or the properties to be checked, the monitor constantly receives the events it is interested in and check if the run has deviated from the desired behaviors. Various of approaches have been proposed among which *synchronous* monitoring [3] and *asynchronous* monitoring [4] are broadly used. Synchronous monitoring blocks the execution of the system being monitored until validity of an observation is confirmed, ensuring that potentially hazardous behavior is not propagated to the system environment. However, synchronous monitoring incurs high execution overhead for the target system, and less critical properties may not require such strict guarantees. On the other hand, asynchronous monitoring may allow for checking the properties with less overhead for the target system, but as the

system continues its execution while checking is performed, it may not be suitable for some critical properties. Furthermore, synchronous monitoring may not be suitable for distributed systems. In many practical cases, it is desirable to combine the two approaches to get the benefits of both and reduce effects of drawbacks.

In [1], we have proposed a tool for the construction and deployment of hybrid monitoring. SMEDL is the corresponding domain specific language for describing monitoring architecture and single monitors. Taking the form of a set of EFSMs, each monitor can be an abstraction of a system object or an abstract entity that represents interactions between multiple system objects. Execution within a monitor is synchronous while the communication among monitors is asynchronous, which allows us to monitor properties on multiple time scales and levels of criticality. The tool has been implemented in Python and is being used to design actual monitoring systems. In this paper, we will give an operational semantics for single monitor of SMEDL and encode it into the Coq platform. Based on the formal semantics, some interesting properties from both the level of language and level of monitor specification can be proved.

The report is organized as follows. Section 2 gives an overview of SMEDL. Section 3 introduces the abstract syntax of SMEDL and its Coq expressions. Section 4 defines the operational semantics of SMEDL based on the synchronous composition of state machines. Then in Section 5, we will give definitions of two properties, termination and determinism, and provide the idea of how to prove them. Section 6 concludes the paper and presents the future work.

## 2 Overview of SMEDL

### 2.1 SMEDL concepts

A SMEDL monitoring system can be divided into four parts: a target system, a monitoring specification, a code generator and runtime checkers, as illustrated in Fig. 1. A target system is the system in which the properties are checked during the runtime. The SMEDL specification contains a set of monitoring objects and an architecture description that captures patterns of communication between them. Each monitoring object can be identified by a list of parameters whose values are fixed when the object is instantiated. Internal state can be maintained in an object to reflect the history of its evolution. Events in SMEDL are instantaneous occurrences that ultimately generated from observations of the execution of target system, which can be pure or carry data through a list of attributes. They can also be raised by monitors in response to other events. Raised events can be delivered to other monitors for checking or serve as alarms. Due to the fact that monitors are connected to the target system through events, a SMEDL specification is independent from the system implementation such that it does not need to be changed as long as the specification of the target system remains the same. Instead, only the definition of events in terms of observations on the target system may be modified. Each monitoring object is converted to executable code by the SMEDL code generator and is instantiated as runtime

checkers multiple times with different parameters, either statically during the target system startup or dynamically at runtime, in response to system events such as the creation of a new thread in the target system. During the compilation, the architecture description is used to provide the communication patterns. Because this paper focuses on the semantics of a single monitor, the principle of instantiation of monitor instances and the definition of architecture description language will not be presented.

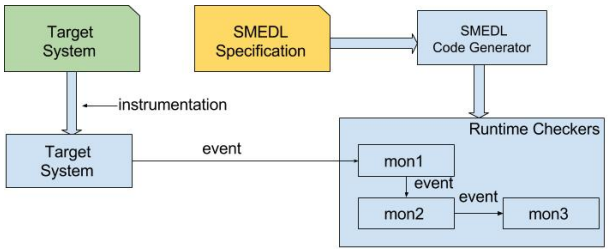


**Fig. 1.** SMEDL overview

### 2.2 Informal definition of the monitoring object

A SMEDL monitoring object consists of a collection of *scenarios*, a set of state variables and a set of event declarations. State variables are used to record the state of the monitor during the execution, which may be written or read by state machines. Each scenario is an extended finite state machine (EFSM) of which the transitions are performed in response to the arrival of events. There are three types of events specified in the event declarations: *imported*, *exported* and *internal*. Imported events, which are responsible for triggering the execution of the monitor, are raised from the target system or other monitors; exported events are raised within the monitor during the transitions of state machines and are then sent to other monitors; and internal events are also used to trigger the transition but they can only be seen and processed within the monitor. In the state machine, each transition is labeled with a triggering event. Additionally, a guard expression may be attached to the transition, which may be a predicate over state variables or event attributes, or a call to a helper function. Moreover, a set of actions can be defined in each transition in which each action is either an update to the state variable, or a statement that raises an event.

Listing 1.2 gives an example of a monitor specification for checking the behavior of a robot simulator and counting the number of moves the robot has taken to retrieve a target [1]. The monitor has two scenarios: $Main$ and $Explore$. $Main$ is used to check whether the robot has found the target in its view and begun to retrieve it. $Explore$ is used to monitor the behavior of robot. Two imported events, *view* and *drive*, are directly received from the robot simulator. $View$ is

sent to the monitor whenever the view of the robot has been updated. If the target is in the robot's view, the monitor raises the internal event *found*, indicating that the robot has found the target. *Drive* is triggered whenever the robot is trying to move. If the helper function *check_retrieved* returns true, the exported event *retrieved* is raised carrying the value of state variable *move_count*, the number of moves that the robot has taken to retrieve the target. Otherwise, *move_count* is increased by 1.

**Listing 1.1.** SMEDL specification for ExplorerMon

```
object ExplorerMon
state
  int mon_x, mon_y, move_count;
events
 imported view(pointer),drive(int,int,pointer),count();
 internal found();
 exported retrieved(int);
scenarios
  Main:
    Explore -> found() -> Retrieve
    Retrieve -> retrieved(cnt) -> Explore
  Explore:
   Look->view(view_pointer) when contains_object(view_pointer){raise found();}->Move
                else -> Move
   Move -> drive(x, y, map) when check_retrieved(map,x,y)
    { raise retrieved(move_count); mon_x = x; mon_y = y; move_count = 0; } -> Look
      else {mon_x = x;mon_y = y; move_count=move_count+1;} -> Look
}
```

A monitor can be treated informally as a black box with input (imported events) and output (exported events). When an imported event is sent into the monitor, it triggers the transitions of the monitor. After finishing its execution, raised exported events is sent to other monitors and the monitor waits for the next arrival of an imported event. This process is defined as a *macro-step*, which cannot be interrupted by the environment. The macro-step is constructed by the synchronous composition of transitions. One or more state machines are triggered to execute the transition by the imported event. Actions of these transitions may raise internal or exported events, which further trigger transitions of other state machines. The process will stop when no more transition is enabled. Then, the monitor finishes the macro-step by outputting the exported events. Note that during each macro-step, each state machine can execute its transition at most once to ensure the termination of the monitor. The formal definition of semantics will be defined in section 4.

## 3 Abstract Syntax of SMEDL

This section presents the abstract syntax of SMEDL: supported data type and definition of expressions will be introduced first; then the state machine will be defined; and finally the definition of the monitoring object will be given. All syntax will be given using Coq [5] expression.

### 3.1 Language Primitives

**Data Type.** At the language level, data types supported by SMEDL include integer, float, string, pointer and opaque. The first three types correspond to

int, double and string in Java. Pointer is a reference to a more complicated data structure, such as arrays, which are not supported at the language level. Data with type pointer is usually used in the helper function implemented in the target language of the SMEDL code generator. Opaque is a data type whose structure is not known by SMEDL, which can also be treated as a reference.

**Expression.** SMEDL supports arithmetic, logical, comparison and atom expressions. The inductive definition of expression in Coq is given below. Note that atom expression in SMEDL includes literal values of integer, float and string, null pointer and variable names.

```
Inductive expr : Set :=
  | EOr : expr -> expr -> expr
  | EAnd : expr -> expr -> expr
  | EEq  : expr -> expr -> expr
  | ENeq : expr -> expr -> expr
  ....
  | EPlus  : expr -> expr -> expr
  | EMinus : expr -> expr -> expr
  ...
  | EAtom  : atom -> expr
  .
```

**Event declaration.** Events used in the monitor are to be declared in the specification. Each event is identified by a unique event name with its type and a list of types of its attributes, defined below. Note that *eventType* is an enumeration over *Internal*, *Imported* and *Exported*; and *typ* denotes the set of supported data types. Once declared in the specification, events can be used to trigger the transition in the state machine when they are sent to the monitor or raised in actions.

```
Record event_definition : Set := {
  eventType   : eventType;
  eventName     : string;
  attributeTypes : list typ;
}.
```

### 3.2 State machine

State machine is a 5-tuple $\langle name, states, initialState, alphabet, transitions \rangle$. *Name* and *states* are respectively the identity and the set of states of the machine. *InitialState*, an element in *states*, is the initial state when the monitor begins running. The *alphabet* is the set of events that can trigger the transitions of the machine.

```
Record stateMachine : Set := {
  scenarioId : string;
  states: scenario_state;
  initialState: scenario_state;
  alphabet : list event_definition;
  transitions: list transition
}.
```

Transition is a four-tuple $\langle originState, targetState, stepEvent, actions \rangle$ where *stepEvent* is the instance of event whose attributes are bound to actual values. In Coq, it is typed with *event_instance*, defined below. Apart from the element *event* to denote the corresponding event declaration, *eventArgs* is defined as the list of local variable names. These variables are bound to actual values and can be referenced within the transition, including the guard condition and actions. *EventGuard* is an optional expression guarding the transition. *Actions* is a set of action statements to be executed right after the transition and it can be divided into two categories: updating state variables and raising events. Note that transitions defined in the state machine are complete, meaning that for any state $s$ in states and any event $e$ in the alphabet, there is a corresponding transition. For any non-interesting transition, the *targetState* is the implicit *errorState*.

```
Record transition : Set := {
  originState : scenario_state;
  stepEvent : event_instance;
  actions : list action;
  targetState : scenario_state
}.

Record event_instance : Set := {
  event: event_definition;
  eventArgs     : list string;
  eventGuard : option expr
}.
```

### 3.3 Monitoring object

A monitor is a 5-tuple $\langle objectId, identities, variables, events, scenarios \rangle$, where *objectId* represents the type of the monitor; *identities* is the set of typed parameters that identify the instance of the monitor. *Variables* is the set of state variables. *EventDeclarations* specifies the signature of events used in the monitor, which have been defined above. *Scenarios* is the set of state machines in the monitor. The Coq definition is given below. Note that *typ_env* is a list of product $atom * typ$, representing the corresponding relation between identifiers and types.

```
Record monitor : Type := {
  objectId   : string;
```

```
  identities : typ_env;
  variables  : typ_env;
  eventDeclarations     : list event_definition;
  scenarios  : list stateMachine
}.
```

## 4 An Operational Semantics of SMEDL

Based on the abstract syntax given in section 3, this section will propose an operational semantics of SMEDL. First, the evaluation function for the expression is given in Coq expression. Then, the semantic rules for constructing the *macro-step* will be proposed.

### 4.1 Evaluation of expressions

In Coq, evaluation of expressions is defined as a fixpoint function, part of which is shown below. To achieve a correct evaluation, the value of each operand should comply with the type requirement of the expression. For example, in the evaluation of plus operation, two operands should be typed with integer or float. In the Coq definition, sub expressions x and y are evaluated first and the returned values are respectively n1 and n2. Monad operator $< num-$ ensures that n1 and n2 are numbers. If not, the evaluation process fails. Moreover, each expression is evaluated under a certain variable scope. In SMEDL, there are two levels of scope: the monitor level and the transition level. State variables are defined at the monitor level such that they can be used or updated in all transitions of all state machines. At the transition level, the triggering event has a list of local bound variables representing the actual parameters of the event, which can be used in the expressions defined in it. The first parameter of the evaluation function is the *environment* of the expression, denoting the valuation of each variable in the scope. During the evaluation, the value of the variable should be fetched from the environment. If a variable out of the scope is used in the expression, the evaluation process fails.

```
Fixpoint evalMonad (env:value_env) (e:expr): ErrorOrResult :=
match e with
|EOr x y => b1 <bool- (evalMonad env x);
                   b2 <bool- evalMonad env y;
                   retBool ((b1 || b2))
...
| EPlus x y => n1 <num- evalMonad env x;
                    n2 <num- evalMonad env y;
                    match n1, n2 with
                    | (inl i), (inl j)  => retInt (i+j)
                    |  (inl i), (inr q) => retFloat ((inject_Z i + q))
                    | (inr p), (inl j) => retFloat ((p + inject_Z j))
```

```
                    | (inr p), (inr q) =>  retFloat (p + q)
                   end
...
|EAtom x => match x with
    | AIdent i => match getValue  (AIdent i) env with
                        | None => Error "Scope error"
                        |  Some (inl (inl (inl z))) => retInt (z)
                        | Some (inl (inl (inr q))) => retFloat (q)
                        | Some (inl (inr s)) => retStr (s)
                        | _ => Error "Type error"
                        end
    | AInt z => retInt (z)
    | AFloat q =>  retFloat (q)
    | AString s => retStr (s)
    | ANull => retInt (0)
    | ABool b => if b then retInt (1) else retInt (0)
    end
| _ => Error "Type error"
end.
```

### 4.2 Definition of semantic rules

This section proposes the formal semantic rules of the execution of single monitor through which the macro-step is constructed based on the current state of the monitor and an imported event. First, *configuration* is introduced to describe the dynamic state of the monitor. Then, semantic rules are defined to manipulate the transitions between the configurations.

Configuration is a four-tuple $< MState, DataState, Ev, Sc >$ where *MState* denotes the mapping from state machines to their respective current states; *DataState* is a type-correct valuation of the state variables; *Ev* is the set of pending events raised by the monitor or the environment, which are used to trigger the transitions inside the monitor or sent to other monitors after the monitor has finished the current macro-step; and *Sc* is the set of state machines that have executed the transition within the current macro-step. The Coq definition of configuration is defined below. Note that the pending events are of type *raisedEvent* in which each attribute of the event has been bound to a value.

```
Record configuration : Type :={
  datastate : value_env;
  controlstate : scenario_env;
  raisedevents :  list raisedEvent;
  exportedevents: list raisedEvent;
  finishedscenarios: list scenario
}.

Record raisedEvent : Type :={
```

```
  eventDefinition: event_definition;
  eventArguments     : list (range_typ)
}.
```

Essentially, a macro-step is constructed by chaining a series of consecutive *micro-steps*. Each micro-step is the synchronous composition of a set of transitions on the state machines with the same triggering event, constructed by applying *basic rule* and *synchrony rule*. Then, the chaining of micro-steps will be performed by applying the *chain merge rule*.

**Basic rule.** The basic rule is applied to a state machine when a transition of that machine is enabled by a pending event. Assume $Conf$ denotes the configuration before applying the rule and $Conf'$ denotes the configuration after applying the rule, the rule is defined below.

$$\frac{tr{:}s1 \xrightarrow{e\{a\}}_c s2, valid(tr, Conf, M), Conf' = updateConfig(Conf, tr)}{Conf \xrightarrow{e} Conf'}$$

$Tr$ is the enabled transition, in which $s1$ is the origin state; $s2$ is the target state; $e$ is the triggering event; $a$ is the set of actions defined in the transition and $c$ is the guard condition of the transition. *Valid* is the function testing on the validity of $tr$ under the configuration $Conf$, defined below.

$$valid(tr, Conf, M) = (e \in Conf.Ev) \wedge (tr \in TS_M) \wedge (\exists mh, Conf.MState(mh) = s1 \wedge mh \notin Conf.Sc) \wedge (tr.condition = true)$$

To enable $tr$, following conditions need to be satisfied: 1) $tr$ is in the set of transitions in the monitor; 2) the triggering event is in the set of pending events; 3) the state machine is at the source state of $tr$; 4) the state machine has not finished its execution; and 5) the guard of $tr$ is satisfied.

Once the above condition is satisfied, $tr$ is taken and $Conf$ is updated, defined below.

$$updateConfig(Conf, tr) =< Conf.SceState[mh = s2], DataState', (e\ is\ InternalEvents\ or\ ExportedEvents?(Conf.Ev \setminus \{e\}) : Conf.Ev) \cup Ev', Conf.Sc \cup \{mh\} >$$

The update of configuration includes: 1) the state machine transitions to *s2* and is put into the set of executed machines; 2) *DataState* is updated according to the actions in the transition; 3) $e$ is removed from the set of pending events; 4) events raised in the actions of the transition are added to the pending events. $Conf \xrightarrow{e} Conf'$ is called *configuration update transition*, which is a three-tuple $< originConf, targetConf, event >$ where $originConf$ is the origin configuration of the transition, $targetConf$ is the target configuration and $event$ is the triggering event. The Coq definition of basic rule is defined below.

```
Definition constructConfig (sce: scenario) (re: raisedEvent)
    (conf : configuration): Configuration:=
 let state := (getScenarioState sce (controlstate conf)) in
 let e :=  findEventInstance re (datastate conf)
  (transEvMap (Some state , Some (eventDefinition re))) in
 let ex_env := createValueEnv (eventArgs e)
   (eventParams (eventDefinition re)) (eventArguments re) in
   if valid (extendEnv (ex_env) (datastate conf)) e sce conf
      then let stp:= (getStep conf sce e) in
        updateConfig extend_env re e sce stp conf
   else
      ...
.
```

**Synchrony rule.** Because basic rules are applied to the state machines, multiple transitions may be obtained with the same source configuration and the same triggering event. In synchrony rule, all target configurations of these transitions will be merged into one configuration. First, the merging of the two configurations is defined below.

**Definition : Merging of Configuration.** 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:

- $\forall mh \in stateMachines$,

$$MState(mh)_{Conf'} = \begin{cases} MState(mh)_{Conf1}\ , MState(mh)_{Conf_1} = MState(mh)_{Conf_2} \\ MState(mh)_{Conf_1}\ , MState(mh)_{Conf_1} \neq MState(mh)_{Conf} \\ MState(mh)_{Conf_2}\ , MState(mh)_{Conf_2} \neq MState(mh)_{Conf} \end{cases}$$

- $\forall v \in StateVariables$,

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

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

Note that to avoid the update conflicts on state variables, actions of any configuration transitions *tr1* and *tr2* to be merged must not update the same state variable. Otherwise, the synchrony rule fails.

The synchrony rule is given below. *Confs* is the set of all target configurations obtained from the basic rule with the source configuration *Conf* and event *e*. Function *MergeAll* will merge all configurations in *Confs* to construct a new configuration.

$$\frac{Confs=\{Conf_i | Conf \xrightarrow{e} Conf_i\}}{Conf \xrightarrow{e} MergeAll(Confs)}$$

The Coq definition of synchrony rule is defined below, where *combineFunc* is the function responsible for merging the list of configurations obtained by the function *constructConfigList*.

```
Definition synchrony  (conf conf' :configuration)
(event:raisedEvent) : Prop :=
In event (raisedevents conf) /\ ~(conflict event conf) /\ combineFunc event conf = conf'.

Definition combineFunc  (e: raisedEvent) (conf: configuration):
Configuration :=
let lst := constructConfigList (dom (controlstate conf)) e conf in
innerCombineFunc' conf None lst'.
```

The new generated configuration by the synchrony rule can also be applied to the basic rule such that the application of basic rule and synchrony rule are performed back and forth until no more configurations can be generated. These configuration transitions have different origin configuration or triggering event, to which the chain merge rule will be applied.

**Chain merge rule.** The objective of chain merge rule is to hide all inner transitions and obtain the set of possible macro-steps, defined below.

$$\frac{Conf \xrightarrow{e1} Conf', Conf' \xrightarrow{e2} Conf''}{Conf \xrightarrow{e1} Conf''} \quad e1 \in ImportedEvents \wedge e2 \in OtherTriggeringEvents$$

The input of the chain merge rule is two configuration transitions, which can be combined when the following conditions are satisfied: target configuration of the $tr1$ is the origin configuration of $tr2$; and the triggering event of $tr1$ is the imported event. The result transition takes the origin transition of $tr1$ and the target configuration of $tr2$ with the imported event as the triggering event. The rule is defined inductively in Coq, shown below. The basic case is that a configuration transition obtained by the synchrony rule is the starting point of a chain merge rule when the triggering event is an imported event (which indicates that the source configuration is an initial configuration). The inductive case describes the hiding of the internal transition.

```
Inductive chainMerge:
   configuration -> configuration-> raisedEvent ->  Prop :=
| basic: forall conf conf' event, (importedEvent (eventDefinition event))
->  synchrony conf conf' event -> chainMerge conf conf' event
| chain: forall conf conf'' conf' event1 event2, chainMerge conf conf' event1
-> synchrony conf' conf'' event2-> chainMerge conf conf'' event1.
```

If there is no error, the chain merge rule is applied to the configurations repeatedly until a final configuration is reached. The definition of a final configuration is given below. First, a final configuration needs to be a *correct* configuration. Informally speaking, a configuration is correct if: 1) the number of state

machine is greater than zero; 2) each state machine can only appear once in the set of executed state machines; 3) there is no duplication in state machines; 4) each executed state machine is always a state machine of the monitor. Then, either the set of pending event is empty or all pending events are exported events which cannot be used to trigger the execution of any state machines.

```
Definition finalConfig (conf:configuration):Prop:=
correct_configuration conf /\ ((length (raisedevents conf) = 0)
\/ (forall e  , In e (raisedevents conf) -> (~importedOrInternalEvent (eventDefinition e)
/\  (forall sce, In sce (dom (controlstate conf))
/\ ~ (In sce (finishedscenarios conf))
-> transEvMap ((getScenarioState sce (controlstate conf)),
Some (eventDefinition e))  = nil))))).
```

Given the semantic rules, there are two issues needed to be answered. One is that although the definition is given above, we are not sure the construction of a macro-step will terminate: from any initial configuration, a final configuration will finally be reached. Another issue is that there are multiple ways of generating micro-steps and chaining them such that multiple final configurations may be constructed. However, to accurately monitor the target system, the behavior of the monitor should be deterministic. Next section will discuss these two properties.

## 5 Properties of SMEDL and Monitors

This section further explores two important properties of the monitor, *termination* and *determinism*. Termination means that a macro-step always terminates if the monitor finishes its whole step or an error is raised when there is a static or runtime error. We will prove that termination is a language-level property. The semantics of SMEDL ensures the termination of the monitor. Determinism states that given a certain state of the monitor and an imported event, the monitor always transitions into the same state and outputs the same set of exported events. In contrast to termination, determinism is a property which can be decided during the execution. A sufficient condition for determinism will be presented and the structure of the proof will be proposed.

### 5.1 Termination

Each monitor works in a reactive way. An imported event is sent to the monitor to trigger the execution of the monitor. State machines in the monitor either transition based on the imported event or the internal events raised in the monitor, or stays at the same state if no event can trigger the transition. Once there is no more available transition, the monitor finishes the execution and outputs the raised exported events. During the execution, runtime error may happen when there are conflicts with updating state variables or pending internal events

cannot be handled by the state machines, resulting in the abnormal exit of the monitor. However, each step always terminates. To prove termination, we first give the definition of a *stuck* configuration using Coq, indicating that if a configuration *conf* has no pending events that can trigger any transition in the monitor, it is a stuck configuration. Apparently, a final configuration is a stuck configuration, which has been proved in Coq.

```
Definition stuckConfig (conf:configuration):Prop:=
forall e , In e (raisedevents conf) -> ~(exists conf', synchrony conf conf' e).
```

The termination theorem is defined below, which shows that for an initial configuration *conf* and an imported event *e*, if no static or runtime error happens because of the transitions triggered by *e* based on *conf*, there always exists a stuck configuration which can be reached from *conf*. Note that, although the proof is omitted, whether a configuration is a stuck configuration is decidable.

```
Theorem Termination: forall conf e
(p:stuckConfig conf + ~ stuckConfig conf), initialConfig conf
-> In e (raisedevents conf)
-> ~(generalErrConfig e conf)
->  (exists conf', chainMerge conf conf' e  /\ (stuckConfig conf')).
```

The idea of proving termination is relatively direct. Because each state machine can only transition once during each macro-step and at least one state machine executes the transition in each micro-step, the number of micro-steps to be taken is bounded by the number of state machines. In Coq, we define the upper bound of micro-steps that a configuration can take illustrated below. The definition takes two inputs, one is the configuration and another is the proof of whether the configuration is a stuck configuration. If the configuration is stuck, the upper bound is zero, indicating that no more micro-steps are allowed to be taken. Otherwise, the upper bound is the number of state machines that have not executed the transition so far.

```
Definition distanceUpperBound (conf:configuration)
(p: (stuckConfig conf) + ~(stuckConfig conf) ) : nat :=
match p with
| inl p' => 0
| inr p' => length (dom (controlstate conf)) - length (finishedscenarios conf)
end.
```

Given the definition, we can prove that the upper bound of the configuration decreases whenever a micro-step is taken. The corresponding lemma is shown below, which can be proved by showing that at least one state machine will be added to the set of executed state machines after the micro-step.

```
Lemma decrease: forall conf conf' e
(p1: (stuckConfig conf) + ~(stuckConfig conf) )
```

```
(p2:(stuckConfig conf') + ~(stuckConfig conf')),
correct_configuration conf -> synchrony conf conf' e
-> UpperBound conf'  p2 <   UpperBound conf  p1.
```

Instead of proving the termination theorem directly, we define relation *Connected* below and prove the termination on it. Relation *Connected* has two cases, denoted as *ConnectedRefl* and *ConnectedTrans*. *ConnectedRefl* indicates that *Connected* is a reflexive relation and *ConnectedTrans* relates two configurations a1 and a3 when a3 can be obtained by multiple micro-steps from a1.

```
Inductive Connected : configuration -> configuration -> raisedEvent -> Prop :=
| ConnectedRefl: forall a e, Connected a a e
| ConnectedTrans: forall a1 a2 a3 e e', synchrony  a1 a2 e
-> Connected a2 a3  e' -> Connected a1 a3 e.
```

The termination lemma on the relation *Connected* is shown below. By having the lemma *decrease*, we could prove this termination lemma by doing the strong induction on the upper bound of micro-steps a configuration can take to the final configuration.

```
Lemma TerminatesOnConnected: forall a e,
correct_configuration a
-> In e (raisedevents a)
-> ~(generalErrConfig e a)
-> exists a',  Connected a a' e /\  (stuckConfig a').
```

In the chain merge rule, two micro-steps can be chained when the prior micro-step is the triggered by the imported event so that the internal event is hidden along with the source configuration of the second micro-step. The *Connected* takes the reverse order, as shown in the rule *ConnectedTrans*. To bridge these two relations, we define the relation *ConnectedBack*. It can then be proved that if a1 and a2 can be related by *Connected*, they can also be related by *ConntectedBack*.

```
Inductive ConnectedBack : configuration -> configuration -> raisedEvent -> Prop :=
| ConnectedBackRefl: forall a e, ConnectedBack a a e
| ConnectedBackTrans: forall a1 a2 a3 e1 e2,
ConnectedBack a1 a2 e1 -> synchrony a2 a3 e2-> ConnectedBack a1 a3 e1.

Lemma Connected_ConnectedBack :
  forall a1 a2 e, Connected a1 a2 e -> ConnectedBack a1 a2 e.
```

Finally, the relationship between *ConntectedBack* and *chainMerge* is described as the lemma shown below.

```
 Lemma ConnectedBack_SmedlConnected :
  forall a1 a2 e, initialConfig a1
-> In e (raisedevents a1)
->  ~(generalErrConfig e a1)
-> ConnectedBack a1 a2 e -> (a1 = a2 \/ chainMerge a1 a2 e).
```

Given the above definitions and lemmas, the termination theorem can be proved.

### 5.2 Determinism

The semantics of the monitor does not impose any restriction on the execution order of transitions, which may lead to the non-determinism if the monitor is not well-formed. For instance, the following monitor specification is ill-formed. After imported event e triggers the execution of transition in m1, events e1 and e2 are raised and put into the set of pending events. Then, the monitor can either choose e1 or e2 to trigger the next micro-step. If e1 is used first, the value of x is assigned with the current value of y before y is updated by triggering e2. If the order is reversed, y is increased by 1 before x is updated, leading to the different result from the previous case.

**Listing 1.2.** an ill-formed monitor

```
object Mon
state
  int x, y;
events
 imported e();
 internal e1(),e2();
scenarios
  m1:
    s11 -e-> s12 {raise e1;raise e2;}
  m2:
    s21 -e1->s22 {x = y;}
  m3:
    s31-e2->s32{y = y+1;}
}
```

To describe the determinism, the equivalence between configurations is first defined below. Two configurations are equivalent to each other if 1) they have valuation of state variables; 2) the corresponding state machine is in the same state; 3) they have the same set of pending events and 4) they have the same set of executed state machines. It can be easily seen that this relation is an equivalence relation.

```
Definition equivConf (conf conf':configuration) : Prop :=
(datastate conf) = (datastate conf') /\ (controlstate conf) = (controlstate conf')
/\ Permutation (raisedevents conf) (raisedevents conf')
/\ Permutation (finishedscenarios conf) (finishedscenarios conf').
```

Because the valuation of state variables influences on the execution of the monitor, it is not possible to check the determinism of a monitor statically. Instead, an approximate sufficient condition is defined to decide whether the monitor can take a deterministic macro-step given the current configuration and imported event. The key point is to check if there are potential dependencies relation between transitions, such that the execution of one transition will change the subsequent transitions. First we give some definitions.

**Definition TriggeringSet:** the union of all internal or exported events which may trigger the transitions in the monitor.

**Definition UsedVarSet_e:** the set of state variables used in all possible transitions triggered by the event e.

**Definition UpdatedVarSet:** the set of state variables to be updated in all possible transitions triggered by the event e.

*TriggeringSet* is the union of all internal or exported events which can trigger the transitions of the monitor. *UsedVarSet* of an event e is the set of all state variables to be used in the guard conditions or updating variables of transitions which may be triggered by e. *UpdatedVarSet* of an event e is the set of all state variables to be updated in the actions of transitions which may be triggered by e.

Then, the causal relation between two events is defined below. The *leadBasicCase* illustrates that if e2 can be raised in actions of transitions triggered by e1, then e1 *causes* e2. The *leadTransit* indicates that *eventLeadTo* is a transitive relation.

```
Inductive eventLeadTo : configuration-> event_definition
-> event_definition -> Prop :=
| leadBasicCase:  forall ev conf ev',  In ev'
(getRaised (getEventInstances (ev) (conf)) (controlstate conf))
-> eventLeadTo conf ev ev'
| leadTransit: forall ev ev' ev'' conf, eventLeadTo conf ev ev'
-> eventLeadTo conf ev' ev'' -> eventLeadTo conf ev ev''
.
```

Another important relation defined here is the interference-free between two events. Two events e1 and e2 are interference-free if the following conditions are satisfied: 1) the UpdatedVarSet and UsedVarSet of two events are disjointed; 2) The UpdateVarSet and UsedVarSet of two events are disjointed; 3) two events can not *lead to* the same event and 4) two events are not in the alphabet of the same state machine.

```
Inductive interference_free: configuration ->
event_definition -> event_definition -> Prop :=
| dep : forall conf e1 e2,
(listIntersectionEmptyProp (UpdatedVarSet conf e1)  (UpdatedVarSet conf e2) )
-> (listIntersectionEmptyProp (UpdatedVarSet conf e1) (UsedVarSet conf e2))
-> (listIntersectionEmptyProp  (UpdatedVarSet conf e2) (UsedVarSet conf e1) )
->  listIntersectionEmptyProp (scenario_alphabet (controlstate conf) e1)
                               (scenario_alphabet (controlstate conf) e2)
-> ~(exists e3, eventLeadTo conf e1 e3 /\ eventLeadTo conf e2 e3)
->  interference_free conf e1 e2
.
```

Given a configuration, the transitions an event can trigger is decided by the current valuation of state variables and the local variables. According to the definition, if two events e1 and e2 are interference-free of each other, the execution of two events will not influence the other, as shown in Fig 2. This property is defined as the following lemma *diamond*.

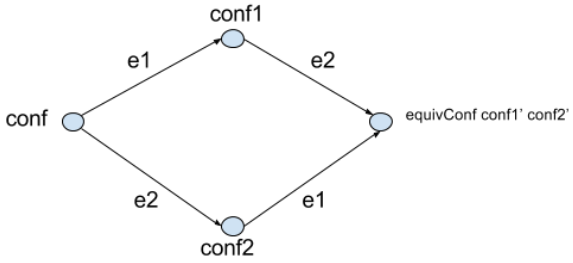


**Fig. 2.** diamond property of interference-free events

```
Lemma diamond: forall conf conf1 conf2  e1 e2,
interference_free conf (eventDefinition e1) (eventDefinition e2) ->
correct_configuration conf ->
synchrony conf conf1 e1 ->
synchrony conf conf2 e2 ->
(exists conf'1 conf'2, synchrony conf1 conf'1 e2
/\ synchrony conf2 conf'2 e1
/\ equivConf conf'1 conf'2).
```

The sufficient condition of a deterministic macro-step is given below. First, no events in the monitor have cyclic causal relation. Then, for all internal and exported events which may trigger the transitions in the monitor, if any pair of events e1 and e2 has no causal relations, then they are interference-free of each other.

```
Definition deter_property (conf:configuration) : Prop :=
noCyclicCausalRelation conf /\
(forall e1 e2,
In e1 (TriggeringSet conf) /\ In e2 (TriggeringSet conf)
/\ ~(eventLeadTo conf e1 e2)
/\ ~(eventLeadTo conf e2 e1)
 ->
(interference_free conf e1 e2))
.
```

The determinism theorem is defined below. If an initial configuration *conf* can reach two stuck configurations *conf1* and *conf2*, then *conf1* and *conf2* are equivalent. The definition of relation *chainMergeWithEventList* is also given here, showing that if configuration can reach to another configuration by multiple micro-steps, then they are related, along with the number of steps taken.

```
Theorem deterministicMacroStep: forall conf conf1 conf2 n1 n2,
initialConfig conf  -> deter_property conf ->
chainMergeWithEventList conf conf1 n1 ->
```

```
chainMergeWithEventList conf conf2 n2 ->
stuckConfig conf1 ->
stuckConfig conf2 ->
equivConf conf1 conf2.

Inductive chainMergeWithEventList: configuration -> configuration
 -> nat ->  Prop :=
| basic_case : forall conf conf' e, synchrony conf conf' e
-> chainMergeWithEventList conf conf' (S O)
| inductive_case forall conf conf' conf'' n  e,
   chainMergeWithEventList conf conf' n  ->
   synchrony conf' conf'' e ->
   chainMergeWithEventList conf conf'' (S n)
```

To prove the determinism theorem, we first need to prove the lemma *weakconfluence* given below. This lemma shows that for a correct configuration *conf*, if 1) the property *deter_property* is satisfied; 2) *conf* can respectively step to *conf1* and *conf2* in one micro-step and multiple micro-steps, then *conf1* and *conf2* will confluent back to two equivalent configurations *conf1'* and *conf2'*. The lemma can be proved by performing induction on n and by using the lemma *diamond*.

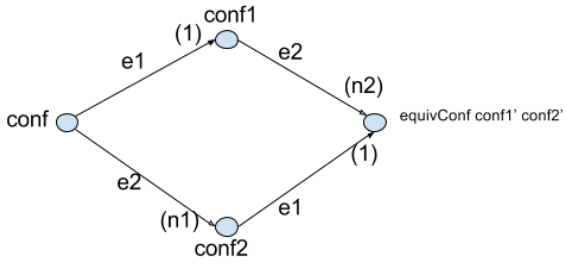


**Fig. 3.** weak confluence property of interference-free events

```
Lemma weakconfluence: forall  n conf conf1 conf2 ,
correct_configuration conf  ->
 deter_property conf ->
chainMergeWithEventList conf conf1 1 ->
chainMergeWithEventList conf conf2 n1 ->
(exists conf1' conf2' n2  , chainMergeWithEventList conf1 conf1' n2
/\ chainMergeWithEventList''' conf2 conf2' 1  /\ equivConf conf1' conf2').
```

Then, we need to further prove the lemma *confluence* given below. The structure of *confluence* is similar with *weakconfluence* except that in the premise *conf*

can also take multiple micro-steps to *conf1*. This lemma can be proved by performing induction on n1 with using *weakconfluence*.

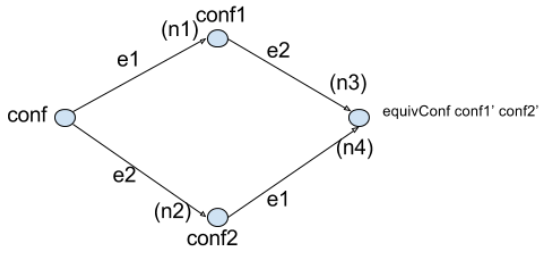


**Fig. 4.** confluence property of interference-free events

```
Lemma confluence: forall  n1 n2 conf conf1 conf2 ,
correct_configuration conf  ->
 deter_property conf ->
chainMergeWithEventList conf conf1 n1 ->
chainMergeWithEventList conf conf2 n2 ->
(exists conf1' conf2' n3 n4  , chainMergeWithEventList conf1 conf1' n3
/\ chainMergeWithEventList conf2 conf2' n4  /\ equivConf conf1' conf2').
```

Finally, the determinism theorem can be proved by contradiction using *confluence*. Note that until now the structure of proving determinism is finished but the proof of diamond and other auxiliary lemmas is still under work.

## 6 Discussion and Conclusions

We presented an operational semantics of SMEDL, a DSL for specifying runtime monitors in a state-machine-style. The execution of the monitor can be seen as a series of macro-steps triggered by the incoming imported event. The macro-step is constructed through the synchronous composition of micro-steps. Each micro-step is a composition of state machine transitions with the same triggering event. The semantic rules of building the macro-step were presented formally and have been encoded into Coq. Moreover, two key properties, termination and determinism, were defined and proved using Coq.

After finishing the proof proposed in the report, we will further improve the efficiency of the semantic rules so that the macro-steps do not need to be constructed explicitly step by step. We will also explore a loose property with a more precise and efficient checking algorithm to ensure the determinism. Apart from the specification language for the single monitor, SMEDL contains the architecture language for describing the monitor network. The formalization of the architecture language is also one of our future projects.

## Acknowledgments

This project is originally a joint project with BAE Systems. I am grateful to John Wiegley, Theo Giannakopoulos from BAE and Clment Pit-Claudel from MIT for providing me much help in design of SMEDL syntax and use of Coq.

## References

1. Zhang, T., Gebhard, P., Sokolsky, O.: Smedl: Combining synchronous and asynchronous monitoring. In: International Conference on Runtime Verification, Springer (2016) 482–490
2. Leucker, M., Schallhart, C.: A brief account of runtime verification. The Journal of Logic and Algebraic Programming **78**(5) (2009) 293–303
3. Francalanza, A.: A theory of monitors. In: Foundations of Software Science and Computation Structures, Springer (2016) 145–161
4. Francalanza, A., Gauci, A., Pace, G.J.: Distributed system contract monitoring. J. Log. Algebr. Program. **82**(5-7) (2013) 186–215
5. Bertot, Y., Castéran, P.: Interactive theorem proving and program development: Coq?Art: the calculus of inductive constructions. Springer Science & Business Media (2013)