A New Coordination Language MediatE

Yi Li and Meng Sun

LMAM and Department of Informatics, School of Mathematical Sciences, Peking University, Beijing, China
liyi_math@pku.edu.cn, sunmeng@math.pku.edu.cn

Abstract. tbd

1 Introduction

2 Overview

The goal of this language MediatE mainly focus on:

1. Compositional Verification.

3 Syntax of MediatE

$$\boxed{\begin{aligned}\langle \mathit{program} \rangle \ ::= (\ &\langle \mathit{import} \rangle \mid \langle \mathit{typedef} \rangle \mid \langle \mathit{function} \rangle \\ &\mid \langle \mathit{automaton} \rangle \mid \langle \mathit{system} \rangle\ )^*\end{aligned}}$$

In the following subsections, we are going to take a simple *queue* as an example, to illustrate how certain language elements are used to compose a model.

3.1 Type System

MediatE provides a rich-featured type system that supports various commonly-used data types in both formal modeling languages and programming languages.

Primitive Type. Table. 1 shows the primitive types supported in MediatE.

Table 1. Primitive Data Types

Name	Declaration	Term Example
Bounded Integer	`int lowerBound .. upperBound`	`-1,0,1`
Integer	`int`	`-1,0,1`
Real	`real`	`0.1, 1E-3`
Boolean	`bool`	`true, false`
Character	`char`	`'a', 'b'`
Enumeration	enum $\mathtt{item}_1$, ..., $\mathtt{item}_n$	`enumname.item`

Composite Type. Composite type offers an approach to contruct complex data types with simpler ones. Several composite patterns are introduced as follows,

Table 2. Composite Data Types (`T` denotes an arbitrary data type)

Name	Declaration
Tuple	$\mathtt{T}_1\mathtt{,...,T}_n$
Union	$\mathtt{T}_1\mathtt{\vert...\vert T}_n$
Array	`T [length]`
Slice	`T []`
Map	$\mathtt{map\ [T}_{key}\mathtt{]\ T}_{value}$
Struct	$\mathtt{struct\ \{\ field}_1\mathtt{:T}_1\mathtt{,...,\ field}_n\mathtt{:T}_n\mathtt{\ \}}$
Initialized	$\mathtt{T}_{base}\mathtt{\ init\ term}$

- *Tuple*. The *tuple* operator ',' can be used to construct a finite tuple type with several base types.
- *Union*. The *union* operator '|' is designed to combine *disjoint* types as a more complicated one. This is similar to the union type in C language but much easier to use.
- *Array* and *Slice*. An *array* $T[n]$ is a finite ordered collection containing exactly n elements of type T. Moreover, a *slice* is an array of which the capacity is not specified, i.e. slice is a dynamic array.
- *Map*. A *map* $[T_{key}]$ T_{val} is a dictionary that maps a key of type T_{key} to a value of type T_{val}.
- *Struct*. A *struct* $\{field_1 : T_1, \cdots, field_n : T_n\}$ contain n fields, each has a particular type T_i and a unique identifier id_i.
- *Initialized*. A initialized type make it able to specify default values to types.

For simplicity in formalizing data types, we introduce the concept *domain* of a type.

Formalization 1 (Domain) *We use $Dom(T)$ to denote the value domain of type T, i.e. the set of all possible value of T.*

Example 1 (Types Used in A Queue). Now let us introduce some type declarations and local variables used in an automaton `Queue`. As shown in the following code fragment, we declares a singleton enumeration `NULL`, which contains only one element `null`. The buffer of a queue is in turn formalized as an array of `T` or `NULL`, indicating that a queue element can be either an assigned item or empty. The head and tail pointer are defined as two bounded integers.

```
typedef enum {null} init null as NULL;
automaton <T:type,size:int> Queue(A:in T, B:out T) {
   variables {
      buf : ((T | NULL) init null) [size];
      phead : int 0 .. (size - 1) init 0;
      ptail : int 0 .. (size - 1) init 0;
   }
   ...
}
```

3.2 Functions

The abstract syntax tree of functions is shown as follows.

$$
\begin{array}{rcl}
\langle \mathit{funcDecl} \rangle & ::= & \texttt{function}\ \langle \mathit{template} \rangle^{?}\ \langle \mathit{identifier} \rangle\ \texttt{(}\ \langle \mathit{arguments} \rangle\ \texttt{)}\ \{ \\
& & (\ \texttt{variables}\ \{\ \langle \mathit{varDecl} \rangle^{*}\ \}\ )^{?} \\
& & \texttt{statements}\ \{\ \langle \mathit{assignStmt} \rangle^{*}\ \langle \mathit{returnStmt} \rangle\ \}\\
\langle \mathit{assignStmt} \rangle & ::= & \langle \mathit{term} \rangle\ \texttt{:=}\ \langle \mathit{term} \rangle \\
\langle \mathit{returnStmt} \rangle & ::= & \texttt{return}\ \langle \mathit{term} \rangle \\
\langle \mathit{varDecl} \rangle & ::= & \langle \mathit{identifier} \rangle\ \texttt{:}\ \langle \mathit{type} \rangle\ (\ \texttt{init}\ \langle \mathit{term} \rangle\ )^{?}
\end{array}
$$

Basically, definition of a function includes:

- An optional template including a set of parameters. A parameter can be either a type parameter (decorated by `type`) or a value parameter (decorated by its type). All possible parameter values of a function should be located statically. Parameters in the template can be used in all the following language elements, e.g. type of input variables and return value, local variables and function statements.
- An identifier that indicates the name of this function.
- A set of read-only input variables.
- A optional set of local variables.
- A list of ordered statements that describes how the return value is calculated. Such a list must be ended by a `return` statement.

Functions in MediatE are side-effect free. In other words, only local variables are writable in its assignment statements.

Example 2 (Incline Operation on a Queue Pointer). The simple function describes how pointers are inclined. When a pointer is going to exceed its upper bound (determined by the parameter *size*), we will reset it to zero.

```
function <size:int> next(pcurr:int 0..(size-1)) : int 0..(size-1) {
   statements { return (pcurr + 1) % size; }
}
```

3.3 Automata : The Basic Behavioral Unit

$$
\begin{array}{rcl}
\langle \mathit{automaton} \rangle & ::= & \texttt{automaton}\ \langle \mathit{template} \rangle^{?}\ \langle \mathit{identifier} \rangle\ \texttt{(}\ \langle \mathit{port} \rangle^{*}\ \texttt{)}\ \{ \\
& & (\ \texttt{variables}\ \{\ \langle \mathit{varDecl} \rangle^{*}\ \}\ )^{?} \\
& & \texttt{transitions}\ \{\ \langle \mathit{transition} \rangle^{*}\ \}\ \} \\
\langle \mathit{port} \rangle & ::= & \langle \mathit{identifier} \rangle\ \texttt{:}\ (\ \texttt{in} \mid \texttt{out}\ )\ \langle \mathit{type} \rangle \\
\langle \mathit{transition} \rangle & ::= & \langle \mathit{guardedStmt} \rangle \mid \texttt{group}\ \{\ \langle \mathit{guardedStmt} \rangle^{*}\ \} \\
\langle \mathit{guardedStmt} \rangle & ::= & \langle \mathit{term} \rangle\ \texttt{->}\ (\ \langle \mathit{stmt} \rangle \mid \{\ \langle \mathit{stmt} \rangle^{*}\ \}\ ) \\
\langle \mathit{stmt} \rangle & ::= & \langle \mathit{term} \rangle\ \texttt{:=}\ \langle \mathit{term} \rangle \mid \texttt{perform}\ \langle \mathit{identifier} \rangle^{+}
\end{array}
$$

Template. Very similar to functions, a automaton can also be decorated with a set of template parameters, either value parameters or type parameters.

Ports. Each automaton contains a set of ports, either `in`-coming or `out`-going, to communicate with the environment. To ensure the well-defineness of automata, ports are required to have an *initialized* type, e.g. `int 0..1 init 0` instead of ~~`int 0..1`~~.

Variables. Two types of variables are used in a automaton definition, they are:

1. *Local variables* that are declared in the *variables* section. A local variable can only be referenced in its scope, i.e. the automaton definition. And similar to the ports, only initialized types are permitted when declaring local variables.
2. *Adjoint variables* that are used to describe the status of ports. For a port A, we assume that it has two boolean fields `A.reqRead` and `A.reqWrite` indicating if there is a pending *read* or *write* request on this port, and a data field `A.value` indicating the current value of this port (if a write operation is performed, `A.value` will be reassigned).

A reasonable rule comes up that, both the `reqWrite` field of a input port and the `reqRead` field of a output port are *read-only*. Similarly, we cannot rewrite the `value` field of a input port.

Transitions. Similar to the PRISM[7] language, behavior of a channel in MediatE is described by a series of guarded transitions (groups). As shown in Example 3, a *transition* comprises two parts: a boolean term *guard* that shows on what condition the transition could be fired, and a (set of) statement(s) that describe what will happen if the transition is fired. Two types of statements are supported in automata,

– *Assignment Statements* ($\texttt{var}_1\texttt{,...,var}_n$ `:=` $\texttt{term}_1\texttt{,...,term}_n$). Local variables and writable adjoint variables are permitted to be assigned here. We can also assign several variables at the same time (similar to the tuple assignment in Python).
– *Perform Statements* ($\texttt{perform port}_1\texttt{,...,port}_n$). Informally speaking, perform statements tell the environment to fire data operations on the output ports, or wait until being noticed that data operation on the input ports are fired by the environment (other automata, actually). Consequently, it's reasonable to require that the value of an input port should never be referred until the port is performed. Similarly, the value of an output port should never be assigned after the port is performed. Perform statements are mainly used when combining multiple automata, where they determine how transitions are synchronized. (See in Section 4.3)

When guard of a transition is satisfied by the context, we say the transition is *activated*. However, being activated is only necessary condition of being fired. When choosing a transition to fire, we have to consider other criteria, which will be introduced later.

Though not mentioned explicitly, transitions can be divided into two classes: *external* and *internal*. A transition is *external* iff. perform assignments are involved.

Formalization 2 (Transitions) *Formally, we use $g \rightarrow S$ to denote a transition, where g is the guard formula and $S = \{s_1, \cdots, s_n\}$ is a set of statements.*

Here we present an example to show how transitions are used to model the behavior of a queue.

Example 3 (Transitions in Queue). In a `Queue`, we use internal transitions to formalize the changes of its state. For example, becoming writable when buffer is not full, and readable when buffer is not empty. External transitions, on the other hand, mainly show how the read and write operations are performed.

```
// Internal Transitions
true -> B.reqWrite := (buf[ptail] != null) ;
true -> A.reqRead := (buf[phead] == null) ;

// External Transitions
(A.reqRead && A.reqWrite) -> {
   perform A; buf[phead] := A.value; phead := next(phead);
}
(B.reqRead && B.reqWrite) -> {
   B.value := buf[ptail]; ptail := next(ptail); perform B;
}
```

All the transitions are supposed to have the following features. They are declared on the syntax level, i.e. we will resolve this feature when discussing the formal aspect of MediatE and use a simple and standard automata model to capture all these features (see in Section. 4).

- *Alterative*. A transition won't be fired if it changes nothing in its context. For example, the first internal transition in a `Queue` will not be activated if `B.reqWrite` is already equal to `buf[ptail] != null`. This assumption is mainly used to avoid useless executions.
- *Urgent*. In some formal models, e.g. CSP[6] and Timed Automata[2], transitions may not be triggered even the guard is satisfied. On contrast, such behavior is strictly prohibited in our model. Once a transition is activated (i.e. its guard is satisfied), it have to be fired unless another guard with higher priority is also activated.
- *Ordered*. An automaton may includes a set of transitions. They are ordered by their appearance. In other words, if several transitions are activated at the same time, the literally former one will be fired first.

Priority of transitions make the automaton fully deterministic. However, in some cases non-determinism is still rather necessary. *Transition groups* are, consequently, imported to represent such behavior. Transitions in the same group do not follow the ordering rule. Instead, the group itself is literally ordered w.r.t. other groups and ungrouped transitions.

Formalization 3 (Transition Groups) *A transition group t_G can be formalized as a finite list of guarded transitions*

$$t_G = \{t_1, \cdots, t_n\}, t_i = g_i \rightarrow S_i$$

where t_i is a single transition with guard g_i and a set of statements S_i.

Since a single transition $g \rightarrow S$ can be equivalently written as a singleton group $\{g \rightarrow S\}$, it's acceptable if we assume that each automaton comprises a set of transition groups but no standalone transitions.

Formalization 4 (Automata) *We use a tuple $A = \langle Ports, Vars, Trans_G \rangle$ to represent an automaton, where $Ports$ is a set of ports, $Vars$ is a set of local variables and $Trans_G = \{t_{G_1}, \cdots, t_{G_n}\}$ is a set of transition groups.*

3.4 System : The Composition Approach

Theoretically, an automaton in MediatE is powerful to represent any classical software system (without consideration of time and probability, of course). However, modeling complex systems in transitions and tons of local variables may become a real disaster. That's why we are going to introduce a new block, called *system*, to help reuse existing automata (systems as well), and construct clear and comprehensible high-level models.

To solve this problem, hierarchical diagrams are widely used in various modeling tools (SCADE[1, 4], Simulink and LabVIEW) and formal languages (Reo[3], AADL). In such diagrams, blocks can be declared as *components* and organized by a set of connections to capture more powerful behavior, where these connections are called *channels*. Figure. 1 gives a simple diagram of a message-oriented middleware, where a queue work as a connector to coordinate between the components (message producers and consumers).

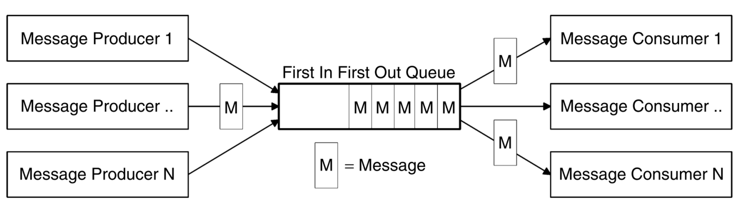

Fig. 1. A Senerio where Queue is used in Message-Oriented Middleware[5]

Both *components* and *connectors* (or *channels*) are well-known concepts in component-based software engineering. Though having different names, their semantics all turn out to the same nature, *automata*. Following with the idea, we introduce a compositional block named *system*, where automata can be declared as either components or connections. The abstract syntax tree of systems is shown as follows.

$$
\begin{array}{rcl}
\langle \mathit{system} \rangle & ::= & \texttt{system}\ \langle \mathit{template} \rangle^{?}\ \langle \mathit{identifier} \rangle\ \texttt{(}\ \langle \mathit{port} \rangle^{*}\ \texttt{)}\ \{ \\
 & & (\ \texttt{internals}\ \langle \mathit{identifier} \rangle^{+})^{?} \\
 & & (\ \texttt{components}\ \{\ \langle \mathit{componentDecl} \rangle^{*}\ \}\ )^{?} \\
 & & \texttt{connections}\ \{\ \langle \mathit{connectionDecl} \rangle^{*}\ \}\ \} \\
\langle \mathit{componentDecl} \rangle & ::= & \langle \mathit{identifier} \rangle^{+}\ \texttt{:}\ \langle \mathit{systemType} \rangle \\
\langle \mathit{connectionDecl} \rangle & ::= & \langle \mathit{systemType} \rangle\ \langle \mathit{params} \rangle\ \texttt{(}\ \langle \mathit{portName} \rangle^{+}\ \texttt{)}
\end{array}
$$

The interface of a system (i.e. its template, name, and ports) shares exactly the same form and meaning with interfaces of automata, which also implies that system is NOT a special semantics unit, but simply an compositional approach to pile up automata. A system is composed of internal nodes (optional), components(optional) and a set of connections.

Components. Automata can be declared as components in a system. Ports of a component can be referred simply by `component.portname` where *portname* is the identifier used in its declaration.

Connections. Connections, e.g. the arrows in Figure. 1, are used to connect the ports of components. Both components and connections are supposed to execute concurrently as automata.

Internals. Directly connecting one port to another is far from enough when modeling complicated systems. For example, in Figure. 1 queue work as a connection between consumers and producers. However, since connections to the middleware are dynamically established or disconnected, we still need an extra merger (from producers to the queue) and an extra replicator (from the queue to the consumers) to achieve our goal. When defining internal nodes, we don't have to specify their types. They should be automatically solved by MediatE.

Example 4 (MediatE Model of the System in Figure. 1). In the previous figure, a simple scenario is presented where a queue is used as a message-oriented middleware. To model this scenario, we need two automata *Producer* and *Consumer* (definitions are omitted due to space limit) that produce or consume message of type T.

```
automaton <T:type> Producer (OUT: out T) { ... }
automaton <T:type> Consumer (IN: in T) { ... }

system <T:type> middleware_in_use () {
   components {
      producer_1, producer_2, producer_3 : Producer<T>;
      consumer_1, consumer_2, consumer_3 : Consumer<T>;
   }
   internals M1, M2 ;
```

```
    connections {
        Merger<T>(producer_1.OUT, producer_2.OUT, producer_3.OUT, M1);
        Queue<T>(M1, M2);
        Replicator<T>(M2, consumer_1.IN, consumer_2.IN, consumer_3.IN);
    }
}
```

Since the example is rather trivial, all the connections and components are automata here. But since automata and systems share the same form of interface, it's also valid to use systems as connections or components.

4 Semantics

In the section, we introduce the formal semantics of MediatE by four steps.

1. Use *configurations* to formally describe the state of a automaton.
2. Convert an automaton to its canonical form, wiping out the dependency on language features *alternative, urgent and ordered*.
3. Provide an algorithm that flats a complex system to an automaton.
4. Specify a transition-system-based semantics to MediatE automata.

4.1 Configurations of Automata

Configurations are used to represent the state of an automaton. Since we don't have locations here, it only depends on the values of its locally accessible variables, which includes both *adjoint variables* and *local variables*.

Definition 1 (Valuation). *A valuation of a set of variables V is defined as a function v that satisfies $\forall x \in V, v(x) \in Dom(type(x))$. We denote the set of all possible valuations of $Vars$ by $Val(Vars)$.*

Definition 2 (Configuration). *A configuration of an automaton $A = \langle Ports, Vars, Trans_G \rangle$ is defined as a tuple (v_{loc}, v_{adj}) where $v_{loc} \in Val(Vars)$ is a valuation on local variables, and $v_{adj} \in Val(Adj(P))$ is a valuation on adjoint variables.*

4.2 Canonical Form of Transitions and Groups

Definition 3 (Canonical Transitions). *A transition $t = g \to \{s_1, \cdots, s_n\}$ is canonical iff. its statements $\{s_i\}$ is an interleaving sequence of assignments and performs which starts from an assignment, e.g.* `a:= exp`$_1$`; perform A; b:= exp`$_2$`;` $\cdots$.

Definition 4 (Canonical Transition Groups). *A transition-group list is canonical if and only if,*

1. *It contains exactly one transition group, and*

2. all the transition in this group are also canonical.

In order to canonicalize a transition, we need to:

1. Merge the contiguous assignments. As mentioned before, an assignment statement is represented as a function $f : EV \to EV$. Thus a list of multiple assignments $f_1, \cdots, f_n$ can be simplified by $f = f_1 \circ \cdots \circ f_n$.
2. Put identical assignments id_{EV} into any two adjacent performs.

Observable. A transition is always *observable*, i.e. it will makes some difference to the context. For example, without this assumption, a transition `true -> x := x` will block the whole model by endless meaningless executions.

Reducing Order. First we consider a simpler situation, where only *single* transitions are involved. Given a set of ordered transitions

$$\{g_1 \to S_1, g_2 \to S_2, \cdots, g_n \to S_n\}$$

As required by the *priority* assumption, a transition can be fired only if all the previous ones are not enabled (i.e. their guards are not satisfied) yet. In MediatE, this feature is resolved simply by adding $\neg g_i$ to all $g_j (j > i)$. E.g.

$$\{g_1 \to S_1, g_2 \wedge (\neg g_1) \to S_2, \cdots, g_n \wedge (\neg g_1 \wedge \neg g_2 \wedge \cdots \wedge \neg g_{n-1}) \to S_n\}$$

Now let's consider a set of ordered groups t_{G_i}, where t_{G_i} contains l_i transitions,

$$T_G = \{t_{G_1} = \{g_{11} \to S_{11}, \cdots, g_{1l_1} \to S_{1l_1}\}, \cdots, t_{G_n} = \{g_{n1} \to S_{n1}, \cdots, g_{nl_n} \to S_{nl_n}\}\}$$

Informally speaking, once a transition in t_{G_1} is enabled, all the other transitions in $tG_i (i > 1)$ should be strictly prohibited from being fired. We use $enab(t_G)$ to denote the condition where at least one transition in t_G is enabled, formalized as

$$enab(t_G = \{g_1 \to S_1, \cdots, g_n \to S_n\}) = g_1 \vee \cdots \vee g_n$$

Then we can generate the new set of transitions with no dependency on priority:

$$\begin{gathered} g_{11} \to S_{11}, \cdots, g_{1l_1} \to S_{1l_1}, \\ g_{21} \wedge \neg enab(t_{G_1}) \to S_{21}, \cdots, g_{2l_2} \wedge \neg enab(t_{G_1}) \to S_{2l_2}, \cdots \\ g_{n1} \wedge \neg enab(t_{G_1}, \cdots, t_{G_{n-1}}) \to S_{n1}, \cdots, g_{nl_n} \wedge \neg enab(t_{G_1}, \cdots, t_{G_{n-1}}) \to S_{nl_n} \end{gathered}$$

where $enab(t_{G_1}, \cdots, t_{G_{n-1}})$ is an abbreviation of $enab(t_{G_1}) \vee \cdots \vee enab(t_{G_{n-1}})$. It indicates that at least one group in t_{G_i} is enabled.

4.3 From System to Automaton

Systems, as shown previously, are simply introduced to construct automata in a more natural way. Now we show how such a system can be flatten as a standard automaton.

Algorithm 1 Scheduling in a Synchronous Set of External Transitions

Require: $t_1, t_2, \cdots, t_n$ are transitions (canonical form)
Ensure: $t = \texttt{Schedule}(t_1, \cdots, t_n)$
if $\{t_i\}$ don't belong to different automata or $\exists t_i$ is internal **then**
    $t \leftarrow null$
    return
end if
$t.g,\ t.S \leftarrow \bigwedge_i t_i.g,\ \{\}$
for $i \leftarrow 1, \cdots, n$ **do**
    if $t_i.s_1$ is an *assignment* **then**
        add $t_i.s_1$ to the head of $t.S$
    end if
    $p \leftarrow$ the first *perform* statement
    while $p \neq null$ **do**
        $a \leftarrow$ the *assignment* statement after p
        $p' \leftarrow$ the next *perform* statement after p
        if $p \in t.S$ **then**
            insert a to $t.S$ exactly after p
            remove p from $t.S$
        end if
    end while
end for
$t \leftarrow \texttt{Canonicalize}(t)$

Algorithm 2 Compose Several Automatons

Require: $A_1, A_2, \cdots, A_n$ are automata
Ensure: $A = \texttt{Compose}(A_1, \cdots, A_n)$
rename local variables in $A_1, \cdots, A_n$ to avoid duplicated names
$A \leftarrow$ empty automaton
$ext\_trans \leftarrow \{\}$
for $i \leftarrow 1, 2, \cdots, n$ **do**
    add all local variables of A_i to A
    add all internal transitions of A_i to A
    add all external transitions of A_i to ext_trans
end for
for $set\_trans \leftarrow$ subset of ext_trans **do**
    $newedge \leftarrow \texttt{Schedule}(set\_trans)$
    if $newedge \neq null$ **then**
        add $newedge$ to A
    end if
end for

4.4 Automaton as Labelled Transition System

Definition 5 (Transition System, TS). *A transition system is a tuple* $(S, \rightarrow)$ *where* S *is a set of states and* $\rightarrow \subseteq S \times \Sigma \times S$ *is a set of transitions. For simplicity reasons, we use* $s \rightarrow s'$ *to denote* (s, s') *in* $\rightarrow$*.*

Suppose $A = \langle Ports, Vars, Trans_G \rangle$ is an automaton, its semantics can be captured by a labelled transition system $\langle S_A, \rightarrow_A \rangle$ where

- S_A is the set of all configurations of A.
- $\rightarrow_A \subseteq S_A \times \Sigma_A \times S_A$ is a set of transitions constructed by the following rules.

$$\frac{p \in P_{in}}{(v_{loc}, v_{adj}) \rightarrow_A (v_{loc}, v_{adj}[p.reqWrite \mapsto \neg p.reqWrite])} \text{ R-INPUTSTATUS}$$

$$\frac{p \in P_{in}, val \in type(p.value)}{(v_{loc}, v_{adj}) \rightarrow_A (v_{loc}, v_{adj}[p.value \mapsto val])} \text{ R-INPUTVALUE}$$

$$\frac{p \in P_{out}}{(v_{loc}, v_{adj}) \rightarrow_A (v_{loc}, v_{adj}[p.reqRead \mapsto \neg p.reqRead])} \text{ R-OUTPUTSTATUS}$$

$$\frac{\{g \rightarrow \{s\}\} \in Trans_G \text{ is internal}}{(v_{loc}, v_{adj}) \rightarrow_A s(v_{loc}, v_{adj})} \text{ R-INTERNAL}$$

$$\frac{\{g \rightarrow S\} \in Trans_G \text{ is external, } \{s_1, \cdots, s_n\} \text{ are the assignments in } S}{(v_{loc}, v_{adj}) \rightarrow_A s_n \circ \cdots \circ s_1(v_{loc}, v_{adj})} \text{ R-EXTERNAL}$$

The first three rules describe the potential change of context, i.e. the adjoint variables. R-InputStatus and R-OutputStatus shows that the reading status of an output port and status of an input port may changed randomly. And R-InputValue shows that the value of an input port may be updated by the context.

The rule R-Internal models the internal transitions in $Trans_G$. As illustrated previously, an internal transition doesn't contains any perform statement. So its canonical form comprises only one assignment s. Firing such a transition will simply apply s to the current configuration.

Meanwhile, R-External models the external transitions, where the automaton need to interact with its context. Fortunately, since all the context change are captured by the first three rules, we can simply regard the context as a set of local variables. Consequently, the only difference between an internal transition and an external transitions is that the later may contains multiple assignments.

5 Discussion

6 Case Study

7 Conclusion and Future Work

References

1. Abdulla, P., Deneux, J., Stålmarck, G., Ågren, H., Åkerlund, O.: Designing safe, reliable systems using SCADE. In: Tiziana, M., Bernhard, S. (eds.) Proceedings of ISoLA 2004. LNCS, vol. 4313, pp. 115–129. Springer (2006)
2. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)
3. Arbab, F.: Reo: a channel-based coordination model for component composition. Mathematical Structures in Computer Science 14(3), 329–366 (2004)
4. Berry, G., Gonthier, G.: The Esterel synchronous programming language: design, semantics, implementation. Science of Computer Programming 19(2), 87–152 (1992)
5. Curry, E.: Message-Oriented Middleware. Middleware For Communications pp. 1–28 (2004)
6. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
7. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of Probabilistic Real-Time Systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) Proceedings of CAV 2011. LNCS, vol. 6806, pp. 1–6. Springer (2011)