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. In this paper, we presents MediatE that intends to provide a two-step modeling approach in component-based software engineering. MediatE is a formal modeling language where we take *automata* as basic behavioral units. Equipped with a rich-featured type system and template mechanism, automata are capable to model generic components and connectors. Such components and connectors are organized through *systems*. The language makes it easy to construct complex models with simply reuse of existing elements, where their formal nature are still maintained.

1 Introduction

2 Syntax of MediatE

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

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.

2.1 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.

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

– *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.

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 item$_1$, ..., item$_n$	enumname.item

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

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

- *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;
   }
```

```
    ...
}
```

2.2 Functions

The abstract syntax tree of functions is shown as follows.

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

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; }
}
```

2.3 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[8] 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* ($\texttt{var}_1\texttt{,...,}\texttt{var}_n\ \texttt{:=}\ \texttt{term}_1\texttt{,...,}\texttt{term}_n$). 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* ($\texttt{perform}\ \texttt{port}_1\texttt{,...,}\texttt{port}_n$). 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 3.3)

A transition is called *external* iff. it synchronizes with the environment through some ports, or *internal* otherwise. Literally, all transitions, where perform statements are involved, are *external* transitions. And in such transitions, the following rules are strictly required to avoid read/write conflicts.

1. Any reference to an input port (A, for example) should be placed after its corresponding perform statement `perform A`.
2. Any reference to an output port (B, for example) should be placed before its corresponding perform statement `perform B`.

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
B.reqWrite && (buf[ptail] == null) -> B.reqWrite := false;
!B.reqWrite && (buf[ptail] != null) -> B.reqWrite := true;
A.reqRead && (buf[phead] != null) -> B.reqRead := false;
!A.reqRead && (buf[phead] == null) -> B.reqRead := true;

// enqueue operation (as an external transition)
(A.reqRead && A.reqWrite) -> {
    perform A; buf[phead] := A.value; phead := next(phead);
}
// dequeue operation (as an external transition)
(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. 3).

– *Urgent*. In some formal models, e.g. CSP[7] 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.*

2.4 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).

Both *components* and *connectors* (or *channels*) are well-known concepts in component-based software engineering. Though having different names, their

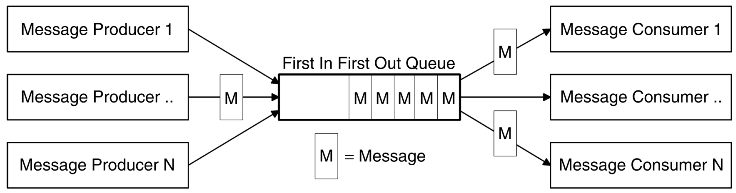

Fig. 1. A Senerio where Queue is used in Message-Oriented Middleware[5]

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.

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

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);
   }
}
```

Formalization 5 (System) *A system is denoted by a 4-tuple*

$$S = \langle Ports, Automata, Internals, Links\rangle$$

where $Ports$ is a set of ports, $Automata$ is a set of automata (both components and connections), $Internals$ is a set of internal nodes and a set of links (pairs of ports or internal nodes) that show how they are joint together.

A well-defined system satisfies the following assumptions:

1. $\forall (p_1, p_2) \in Links$, data transfer from p_1 to p_2. For example, if $p_1 \in Ports$ is an input port, p_2 could be *a) an output port of the system ($p_2 \in Ports$)*, *b) an input port of some automaton $A_i \in Automata$ ($p_2 \in A_i.Ports$)* or *c) an internal node ($p_2 \in Internals$)*.
2. $\forall n \in Internals, \exists! p_1, p_2$ i.e. $(p_1, n), (n, p_2) \in Links$.
3. The *type* function can be extended to $Internals$ and satisfies $\forall (p_1, p_2) \in Links, type(p_1) = type(p_2)$.

3 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.

3.1 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. We use $Conf(A)$ to denote all the configurations of A.*

3.2 Canonical Form of Transitions and Automata

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 and ends by assignments, e.g.* `a := exp`$_1$`; perform A; b := exp`$_2$`;` $\cdots$ `c := exp`$_3$.

Assume $g \to \{s_1, \cdots, s_n\}$ is a transition of automaton A, the following algorithm shows how it is canonicalized,

- **S1** Any continuous subsequences $s_i, \cdots, s_j (j > i)$ exists where all elements are assignment statements should be merged. As mentioned before, an assignment statement is represented as a function $f : Conf(A) \to Conf(A)$. Thus a list of multiple assignments $s_i, \cdots, s_j$ can be replaced using $s' = s_i \circ \cdots \circ s_j$.
- **S2** Keep going with *S1* until there is no further subsequence to merge.
- **S3** Put identical assignment $id_{Conf(A)}$ into any two adjacent *performs*. Similarly, if the statements start from or end with a *perform* statement, we should also use $id_{Conf(A)}$ to decorate its head and tail.

Definition 4 (Canonical Automata). *An automaton $A = \langle Ports, Vars, Trans_G \rangle$ is canonical iff. a) $Trans_G$ includes only one transition group and b) all transitions in this group are also canonical.*

Now we show how $Trans_G$ is reformed to make the automaton canonical. Assume that $Trans_G$ can be represented by,

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

Informally speaking, once a transition in t_{G_1} is enabled, all the other transitions in $t_{G_i} (i > 1)$ 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.

3.3 From System to Automaton

As mentioned previously, system in MediatE provides an approach to combine automata through *components* and *connectors*. However, a system is not a semantics element in our framework, in other words, behavior of a system relies on how the automata are organized and scheduled. In this section, we present the algorithms that formally describe the composition approach in MediatE.

Algorithm 1. shows how to construct the skeleton of target automaton. When flatting a *system*, first we rename all the variables in this sub-automata (including both components and connectors) to avoid name conflicts. Next we simply copy all the internal transitions to the target automaton.

Algorithm 1 Flatting Systems

Require: A system $S = \langle Ports, Automata, Internals, Links\rangle$
Ensure: An automaton A

$A \leftarrow$ empty automaton
$A.Ports \leftarrow S.Ports$
rename *local variables* in $Automata = \{A_1, \cdots, A_n\}$ to avoid duplicated names
$ext\_trans \leftarrow \{\}$
for $i \leftarrow 1, 2, \cdots, n$ **do**
  $A.Vars \leftarrow A.Vars + A_i.Vars$
  $A.Trans_G \leftarrow A.Trans_G + Internal(A_i.Trans_G)$
  $ext\_trans \leftarrow ext\_trans + External(A_i.Trans_G)$
end for
for $set\_trans \in 2^{ext\_trans}$ **do**
  $new\_edge \leftarrow \mathtt{Schedule}(S, set\_trans)$
  if $new\_edge \neq null$ **then**
    $A.Trans_G = A.Trans_G + \{new\_edge\}$
  end if
end for

External transitions, on the other hand, have to synchronize with its corresponding external transitions in other automata. For example, when an automaton want to read some thing from a input port P_1, there must be another one

Algorithm 2 Scheduling in a Synchronous Set of External Transitions

Require: A System S, a set of transitions $t_1, t_2, \cdots, t_n$ (in canonical form)
Ensure: A synchronized transition t

1: **if** $\{t_i\}$ don't belong to different automata or $\exists t_i$ is internal **then**
2: $\quad t \leftarrow null$
3: $\quad$ **return**
4: **end if**
5: $t.g,\ t.S \leftarrow \bigwedge_i t_i.g,\ \{\}$
6: *shared_ports*$\leftarrow$ all the
7:
8: $G \leftarrow$ a Graph $\langle V, E\rangle$ {create a dependency graph}
9: **for** $i \leftarrow 1, \cdots, n$ **do**
10: $\quad$ add $\bot_i, \top_i$ to $G.V$
11: $\quad$ $lasts \leftarrow \{\bot_i\}$
12: $\quad$ **for** $j \leftarrow 1, 3, \cdots, len(t_i.S) - 1$ **do**
13: $\quad\quad$ $ports \leftarrow$ all the performed ports in $t_i.S_{j+1}$
14: $\quad\quad$ **for** $l \in lasts, p \in ports$ **do**
15: $\quad\quad\quad$ **if** $p \notin G.V$ **then**
16: $\quad\quad\quad\quad$ add p to $G.V$
17: $\quad\quad\quad$ **end if**
18: $\quad\quad\quad$ add edge $l \xrightarrow{t_i.S_j} p$ to $G.E$
19: $\quad\quad$ **end for**
20: $\quad$ **end for**
21: $\quad$ **for** $l \in lasts$ **do**
22: $\quad\quad$ add edge $l \xrightarrow{t_i.S_{len(t_i.S)}} \top_i$ to $G.E$
23: $\quad$ **end for**
24: **end for**
25:
26: **if** (G comprises a ring) *or* ($\exists v \in G.v$ is a port whose *degree* $\neq 4$) **then**
27: $\quad t \leftarrow null$
28: **else**
29: $\quad$ $t.S \leftarrow \{$ select all the statements in $G.E$ using topological sort $\}$
30: $\quad$ replace `perform P` in $t.S$ with `P.reqRead, P.reqWrite := false, false`
31: **end if**

that is writing something to its output port P_2 where P_1 and P_2 are overlapped in the system.

In MediatE *systems*, only adjoint variables (`reqRead, reqWrite and value`) are shared between automata. During synchronization, the most important principle is to make sure assignments to shared variables are executed before dereferencing them. The detailed algorithm is described in Algorithm 2.

Line 25 shows several situations where the synchronization process fails,

1. The generated graph includes a *ring*, which is a sign of *circular dependencies*. For example, in one transition `perform A` shows up earlier than `perform B`, but in another transition the order is reversed.

2. The generated graph includes a non-trivial vertex (labelled by a perform statement) whose degree is not equal to 4. In other words, a port is not properly synchronized or synchronized with more than two transitions.

Topological sorting, as we all knows, may generate different schedules for the same graph. The following theorem shows that all these schedules are equivalent as transition statements.

Theorem 1 (Equivalence between Schedules). *If two set of assignment statements S_1, S_2 are generated from the same set of external transitions, they have exactly the same behavior.*

3.4 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.

4 Case Study

In modern distributed computing frameworks (e.g. MPI and ZooKeeper), *leader election* plays an important role to organize multiple servers efficiently and consistently. This section shows how a classical leader election algorithm is modeled and easily used to coordinate other components in MediatE.

[6] proposed an classical algorithm for a typical leader election scenario, as shown in Figure. 2. Distributed processes are organized as a *asynchronous unidirectional* ring where communication take place only between adjacent processes and following certain direction (from left to right in this case).

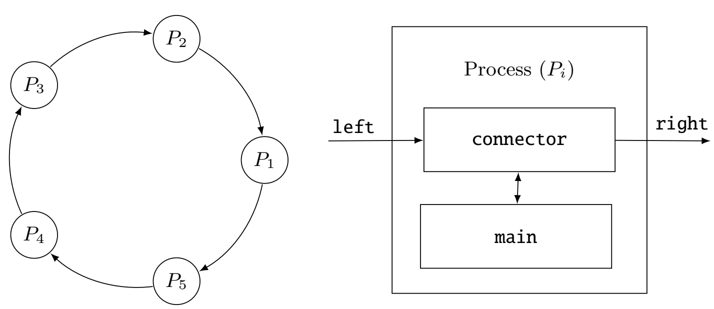

Fig. 2. (a) Topology of a Asynchronous Ring and (b) Structure of a Process

The algorithm mainly includes the following steps:

1. To begin with, each process sends a voting message including its own *id* to its successor.
2. A process, when receives a voting message, will
   – forward the message to its successor if it contains a larger *id* than itself,
   – ignore the message if it contains a smaller *id* than itself, and
   – take itself as a leader if it contains the same *id* with itself.

Here we formalize this algorithm through a more general approach. Leader election is encapsulated as `connector` since it is also responsible to handle the communication between processes. A computing module `main` is attached to the connector, and used to model computing tasks.

Two types of messages, `msgVote` and `msgLocal`, are supported when formalizing this architecture.

Example 5 (Definition of Message Types). Voting messages `msgVote` are transferred between the connectors. A voting message carries two fields, *vtype* that declares the stage of leader election (either it is still voting or some process has already been acknowledged) and a *id* an identifier of the current leader (if have).

```
typedef struct { vtype: enum {vote, ack}, id: int } as msgVote;
typedef struct {
  status : enum { pending, acknowledged },
  idLocal : int,
  idLeader : int | NULL
} as msgLocal;
```

```
automaton <id:int> election_module (
  left : in msgVote, right : out msgVote,
  query : in msgLocal, notice : out msgLocal
) { ... }

system cluster_instance() {
  components {
    E1 : election_module<1>; E2 : election_module<2>;
    E3 : election_module<3>;
    C1, C2, C2 : worker;
  }
  connections {
    Sync<msgVote>(E1.left, E2.right);
    Sync<msgVote>(E2.right, E3.left );
    Sync<msgVote>(E3.right, E1.left );

    Sync<msgLocal>(C1,query, E1.query );
    Sync<msgLocal>(C1,notice, E1.notice);
    Sync<msgLocal>(C2,query, E2.query );
    Sync<msgLocal>(C2,notice, E2.notice);
    Sync<msgLocal>(C3,query, E3.query );
    Sync<msgLocal>(C3,notice, E3.notice);
  }
}
```

5 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. Hagit, A., Jennifer, W.: Distributed computing: fundamentals, simulations, and advanced topics. John Wiley & Sons (2004)
7. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
8. 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)

Appendix

Theorem 1 (Equivalence between Schedules). *If two set of assignment statements S_1, S_2 are generated from the same set of external transitions, they have exactly the same behavior (i.e. execution of S_1 and S_2 under the same configuration will lead to the same result.*

Proof. Apparently, when executing statements, all the changes on configurations come from *assignments*. Once we successfully prove that for each assignment, its pre-configuration and post-configuration in S_1 and S_2 are exactly the same, we are able to finish this proof.

In the following proof, we denote S_1 and S_2 by $S_1 = \{s_1, \cdots, s_n\}, S_2 = \{s'_1, \cdots, s'_n\}$, and the automaton that a transition belongs to by $Automaton(s)$.

1. Let's come to the *FIRST* assignment state s in S_1 where shared variables is assigned. We assume that its corresponding statement in S_2 is s'. Comparing s and s', we have:
   (a) s' is also the first assignment in S_2 which modifies *this set* of variables. *(A shared variable can be assigned in one of its owner, thus all assignments that modifies this variable belong to the same transition, and their order is strictly maintained.)*
   (b) s and s' include no reference to other shared variables. *(A shared variable can be dereferenced only when it has been assigned before, however s is the first assignment which modifies a shared variable.)*
   (c) In the pre-configuration of s and s', all the local variables of $Automaton(s)$ have the same evaluation. *(Derived from the same reason in (a)).*

   Consequently, in the post-configuration of s and s', all the shared variables have the *SAME* evaluation.
2. Assume that all assignments (to shared variables) in $s_1, \cdots, s_i$ have been proved that satisfy the proposition, now we are going to prove that s, the first transition where shared variables are dereferenced in $s_{i+1}, \cdots, s_n$ and its corresponding s' also satisfy it.