

Component-Based Modeling in Mediator

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 the new language Mediator that intends to provide a two-step modeling approach in component-based software engineering. Mediator 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

Component-based software engineering, as one of the *software reuse* approaches, has been prospering for a long time. Through proper encapsulation and clearly declared interface, a *component* can be invoked by different applications without knowledge on its implementation details. Currently, there are various tool supports on component-based modeling:

1. *Industrial tools*, including commercial tools like NI LabVIEW[7], MathWorks Simulink, and academic tools like Ptolomy[10]. These tools provide powerful formalism and a large number of built-in component to support commonly-used platforms. However, due to the complexity of models, such tools mainly focus on synthesis and simulation, instead of formal verification.
2. *Formal tools*, e.g. Esterel SCADE[1] and rCOS[13]. SCADE, based on a synchronous data flow language LUSTRE, is equipped with a powerful tool-chain and widely used development of embedded systems. rCOS, on the other hand, is a refinement calculus on object-oriented design.

Existing work[15] has shown that, formal verification based on existing industrial tools is hard to realize due to its complexity and non-open architecture. However, according to the feedbacks from programmers, unfamiliarity of formal specifications is still the main obstacle stopping the from using formal tools. For example, even in the most famous formal modeling tools with graphical user interfaces (e.g. PRISM[11], Uppaal[3]), it requires at least knowledge on automata theory to properly encode the models.

Reo[4], the coordination language, provides a solution where advantages of both can be integrated in a natural way. Reo is a channel-based language where

its semantics are clearly specified in the very beginning. And thanks to its graphical notations, as shown in Figure. 1, organization of components can be illustrated and exhibited in a natural way.

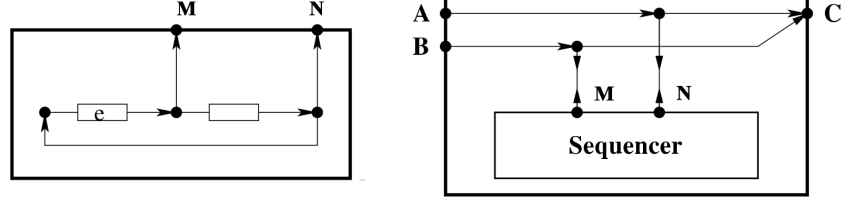


Fig. 1. How A Sequencer Connector is Defined and Reused in Reo [14]

Inspired by Reo, we present a new modeling language, Mediator. Mediator is a hierarchical modeling language that provides formalism for both high-level *system* layouts and low-level *automata*-based behavioral units. With help of a rich-featured type system, we can describe complex data structure and powerful automata in a rather formal way. And automata (or other systems) can be then declared as either components or connectors in a system. Both automata and systems are encapsulated with a interface containing *a) a set of input or output ports* and *b) a set of template parameters* so that they can be easily reused in multiple projects.

The paper is structured as follows. In Section 2, we briefly present the syntax of Mediator and formalizations of the language entities. Then in Section 3, we introduce the formal semantics of Mediator. Section 4 presents a case study where a commonly used coordination algorithm *leader election* is modeled in Mediator. The conclusion and future work can be found in Section 5.

2 Syntax of Mediator

In this section, we introduce the syntax of Mediator. A Mediator program, as shown in the following block, essentially contains several parts,

$$\langle \text{program} \rangle ::= (\langle \text{typedef} \rangle \mid \langle \text{function} \rangle \mid \langle \text{automaton} \rangle \mid \langle \text{system} \rangle)^*$$

1. *Typedefs* that give aliases to specified types.
2. *Function* definitions that defines customized functions.
3. *Automaton* blocks that describe an automaton with given parameters.
4. *System* blocks to compose automata as components or connections.

2.1 Type System

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

Table 1. Primitive Data Types

| Name | Declaration | Term Example |
|-----------------|---|----------------------------|
| Bounded Integer | <code>int lowerBound .. upperBound</code> | <code>-1,0,1</code> |
| Integer | <code>int</code> | <code>-1,0,1</code> |
| Real | <code>real</code> | <code>0.1, 1E-3</code> |
| Boolean | <code>bool</code> | <code>true, false</code> |
| Character | <code>char</code> | <code>'a', 'b'</code> |
| Enumeration | <code>enum item₁, ..., item_n</code> | <code>enumname.item</code> |

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

Composite Type. Composite type offers an approach to construct 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 | <code>T₁, ..., T_n</code> |
| Union | <code>T₁ ... T_n</code> |
| Array | <code>T [length]</code> |
| Slice | <code>T []</code> |
| Map | <code>map [T_{key}] T_{value}</code> |
| Struct | <code>struct { field₁:T₁, ..., field_n:T_n }</code> |
| Initialized | <code>T_{base} init term</code> |

- *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* $\{field_1 : T_1, \dots, field_n : T_n\}$ 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 declare 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.

```

1  typedef enum {null} init null as NULL;
2  automaton <T:type,size:int> Queue(A:in T, B:out T) {
3      variables {
4          buf : ((T | NULL) init null) [size];
5          phead : int 0 .. (size - 1) init 0;
6          ptail : int 0 .. (size - 1) init 0;
7      }
8      ...
9  }
```

2.2 Functions

The abstract syntax tree of functions is shown as follows.

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

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 Mediator 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.

```

1 function <size:int> next(pcurr:int 0..(size-1)) : int 0..(size-1) {
2   statements { return (pcurr + 1) % size; }
3 }

```

2.3 Automata : The Basic Behavioral Unit

$$\begin{aligned}
\langle \text{automaton} \rangle &::= \text{automaton } \langle \text{template} \rangle^? \langle \text{identifier} \rangle (\langle \text{port} \rangle^*) \{ \\
&\quad (\text{variables } \{ \langle \text{varDecl} \rangle^* \})^? \\
&\quad \text{transitions } \{ \langle \text{transition} \rangle^* \} \} \\
\langle \text{port} \rangle &::= \langle \text{identifier} \rangle : (\text{in} \mid \text{out}) \langle \text{type} \rangle \\
\langle \text{transition} \rangle &::= \langle \text{guardedStmt} \rangle \mid \text{group } \{ \langle \text{guardedStmt} \rangle^* \} \\
\langle \text{guardedStmt} \rangle &::= \langle \text{term} \rangle \rightarrow (\langle \text{stmt} \rangle \mid \{ \langle \text{stmt} \rangle^* \}) \\
\langle \text{stmt} \rangle &::= \langle \text{term} \rangle := \langle \text{term} \rangle \mid \text{perform } \langle \text{identifier} \rangle^+
\end{aligned}$$

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[?] language, behavior of a channel in Mediator 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* (**var₁, ..., var_n := term₁, ..., 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* (**perform** $\text{port}_1, \dots, \text{port}_n$). 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 $S = \{s_1, \dots, s_n\}$ 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.

```

1  // Internal Transitions
2  B.reqWrite && (buf[ptail] == null) -> B.reqWrite := false;
3  !B.reqWrite && (buf[ptail] != null) -> B.reqWrite := true;
4  A.reqRead && (buf[phead] != null) -> B.reqRead := false;
5  !A.reqRead && (buf[phead] == null) -> B.reqRead := true;
6
7  // enqueue operation (as an external transition)
8  (A.reqRead && A.reqWrite) -> {
9      perform A; buf[phead] := A.value; phead := next(phead);
10 }
11 // dequeue operation (as an external transition)
12 (B.reqRead && B.reqWrite) -> {
13     B.value := buf[ptail]; ptail := next(ptail); perform B;
14 }
```

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 Mediator 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[9] 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*

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

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 $Trans_G = \{t_{G_1}, \dots, t_{G_n}\}$ is a set of transition groups.*

2.4 System : The Composition Approach

Theoretically, an automaton in Mediator 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, 5], Simulink and LabVIEW) and formal languages (Reo[4], 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. 2 gives a simple diagram of a message-oriented

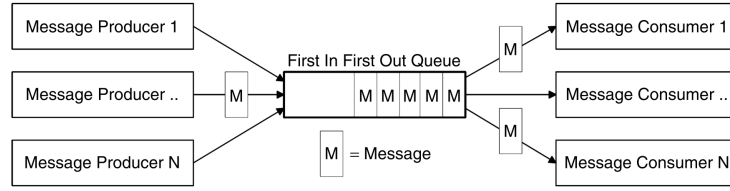


Fig. 2. A Senerio where Queue is used in Message-Oriented Middleware[6]

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 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.

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

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. 2, 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. 2 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 Mediator.

Example 4 (Mediator Model of the System in Figure. 2). 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 .

```

1  automaton <T:type> Producer (OUT: out T) { ... }
2  automaton <T:type> Consumer (IN: in T) { ... }
3
4  system <T:type> middleware_in_use () {
5      components {
6          producer_1, producer_2, producer_3 : Producer<T>;
7          consumer_1, consumer_2, consumer_3 : Consumer<T>;
8      }
9      internals M1, M2 ;
10     connections {
11         Merger<T>(producer_1.OUT, producer_2.OUT, producer_3.OUT, M1);
12         Queue<T>(M1, M2);
13         Replicator<T>(M2, consumer_1.IN, consumer_2.IN, consumer_3.IN);
14     }
15 }
```

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 Mediator by four steps.

1. Use *configurations* to formally describe the state of a automaton.
2. Convert an automaton to its canonical form.
3. Provide an algorithm that flats a complex system to an automaton.
4. Specify a transition-system-based semantics to Mediator 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 $\forall x \in V, v(x) \in \text{Dom}(\text{type}(x))$. We denote the set of all possible valuations of Vars by $\text{Val}(\text{Vars})$.

Definition 2 (Configuration). A configuration of an automaton $A = \langle \text{Ports}, \text{Vars}, \text{Trans}_G \rangle$ is defined as a tuple $(v_{\text{loc}}, v_{\text{adj}})$ where $v_{\text{loc}} \in \text{Val}(\text{Vars})$ is a valuation on local variables, and $v_{\text{adj}} \in \text{Val}(\text{Adj}(P))$ is a valuation on adjoint variables. We use $\text{Conf}(A)$ to denote all the configurations of A .

3.2 Canonical Form of Transitions and Automata

Definition 3 (Canonical Transitions). A transition $t = g \rightarrow \{s_1, \dots, s_n\}$ 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 := exp₁; perform A; b := exp₂; ... c := exp₃.**

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

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

Definition 4 (Canonical Automata). An automaton $A = \langle \text{Ports}, \text{Vars}, \text{Trans}_G \rangle$ 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,

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

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 $\text{enab}(t_G)$ to denote the condition where at least one transition in t_G is enabled, formalized as

$$\text{enab}(t_G = \{g_1 \rightarrow S_1, \dots, g_n \rightarrow S_n\}) = g_1 \vee \dots \vee g_n$$

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

$$\begin{aligned}
&g_{11} \rightarrow S_{11}, \dots, g_{1l_1} \rightarrow S_{1l_1}, \\
&g_{21} \wedge \neg enab(t_{G_1}) \rightarrow S_{21}, \dots, g_{2l_2} \wedge \neg enab(t_{G_1}) \rightarrow S_{2l_2}, \dots \\
&g_{n1} \wedge \neg enab(t_{G_1}, \dots, t_{G_{n-1}}) \rightarrow S_{n1}, \dots, g_{nl_n} \wedge \neg enab(t_{G_1}, \dots, t_{G_{n-1}}) \rightarrow S_{nl_n}
\end{aligned}$$

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

3.3 From System to Automaton

As mentioned previously, system in Mediator 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 Mediator.

Algorithm 1. shows how to construct the skeleton of target automaton. When flattening 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 Flattening Systems

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

Ensure: An automaton A

```

1:  $A \leftarrow$  empty automaton
2:  $A.Ports \leftarrow S.Ports$ 
3: rename local variables in  $Automata = \{A_1, \dots, A_n\}$  to avoid duplicated names
4:  $ext\_trans \leftarrow \{\}$ 
5: for  $i \leftarrow 1, 2, \dots, n$  do
6:    $A.Vars \leftarrow A.Vars + A_i.Vars$ 
7:    $A.Trans_G \leftarrow A.Trans_G + Internal(A_i.Trans_G)$ 
8:    $ext\_trans \leftarrow ext\_trans + External(A_i.Trans_G)$ 
9: end for
10: for  $set\_trans \in 2^{ext\_trans}$  do
11:    $new\_edge \leftarrow Schedule(S, set\_trans)$ 
12:   if  $new\_edge \neq null$  then
13:      $A.Trans_G = A.Trans_G + \{new\_edge\}$ 
14:   end if
15: 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 an 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, \dots, 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:    $t \leftarrow null$ 
3:   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  $i \leftarrow 1, \dots, n$  do
10:   add  $\perp_i, \top_i$  to  $G.V$ 
11:    $lasts \leftarrow \{\perp_i\}$ 
12:   for  $j \leftarrow 1, 3, \dots, len(t_i.S) - 1$  do
13:      $ports \leftarrow$  all the performed ports in  $t_i.S_{j+1}$ 
14:     for  $l \in lasts, p \in ports$  do
15:       if  $p \notin G.V$  then
16:         add  $p$  to  $G.V$ 
17:       end if
18:       add edge  $l \xrightarrow{t_i.S_j} p$  to  $G.E$ 
19:     end for
20:   end for
21:   for  $l \in lasts$  do
22:     add edge  $l \xrightarrow{t_i.S_{len(t_i.S)}} \top_i$  to  $G.E$ 
23:   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:    $t \leftarrow null$ 
28: else
29:    $t.S \leftarrow \{ \text{select all the statements in } G.E \text{ using topological sort} \}$ 
30:   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 Mediator *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 (i.e. execution of S_1 and S_2 under the same configuration will lead to the same result).*

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.

$$\begin{array}{c}
\frac{p \in P_{in}}{(v_{loc}, v_{adj}) \rightarrow_A (v_{loc}, v_{adj}[p.reqWrite \mapsto \neg p.reqWrite])} \text{ R-INPUTSTATUS} \\
\\
\frac{p \in P_{in}, val \in type(p.value)}{(v_{loc}, v_{adj}) \rightarrow_A (v_{loc}, v_{adj}[p.value \mapsto val])} \text{ R-INPUTVALUE} \\
\\
\frac{p \in P_{out}}{(v_{loc}, v_{adj}) \rightarrow_A (v_{loc}, v_{adj}[p.reqRead \mapsto \neg p.reqRead])} \text{ R-OUTPUTSTATUS} \\
\\
\frac{\{g \rightarrow \{s\}\} \in Trans_G \text{ is internal}}{(v_{loc}, v_{adj}) \rightarrow_A s(v_{loc}, v_{adj})} \text{ R-INTERNAL} \\
\\
\frac{\{g \rightarrow S\} \in Trans_G \text{ is external, } \{s_1, \dots, s_n\} \text{ are the assignments in } S}{(v_{loc}, v_{adj}) \rightarrow_A s_n \circ \dots \circ s_1(v_{loc}, v_{adj})} \text{ R-EXTERNAL}
\end{array}$$

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 contain 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 needs to interact with its context. Fortunately, since all the context changes 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 transition is that the latter may contain 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 Mediator.

[8] proposed a classical algorithm for a typical leader election scenario, as shown in Figure. 3. Distributed processes are organized as a *asynchronous unidirectional* ring where communication takes place only between adjacent processes and following certain direction (from left to right in this case).

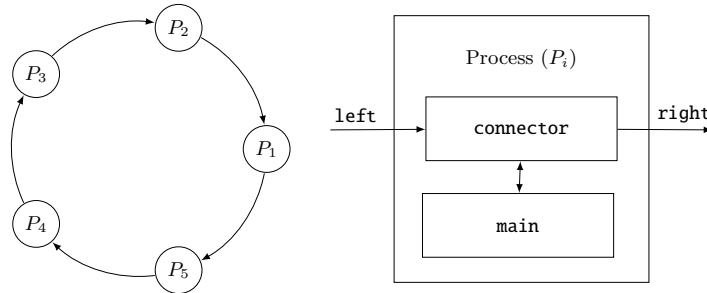


Fig. 3. (a) Topology of an Asynchronous Ring and (b) Structure of a Process

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. 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). On the other hand, **msgLocal** is used when a worker want to communicate with its corresponding connector.

```

1  typedef struct { vtype: enum {vote, ack}, id: int } as msgVote;
2  typedef struct {
3    status : enum { pending, acknowledged },
4    idLocal : int,
5    idLeader : int | NULL
6  } as msgLocal;

1  automaton <id:int> election_module (
2    left : in msgVote, right : out msgVote,
3    query : out msgLocal
4  ) { ... }
```

The following code fragment encodes a parallel program containing 3 workers and their corresponding election modules. It is a simplified version of the one in Figure. 3. In this example, *worker*, the main calculating, is passed as a parameter since we expect that this system should be capable handling different working process.

As we are modeling the leader election algorithm on a synchronous ring, only synchronous communications channel *Syncs* are involved in this example. *Sync* is a Reo channel in the first, but also modeled as an *automaton* in our framework. It's implementation details can be found in [12].

```

1  system <worker: interface (query:in msgLocal)> parallel_instance() {
2    components {
3      E1 : election_module<1>; E2 : election_module<2>;
4      E3 : election_module<3>;
5      C1, C2, C3 : worker;
6    }
7    connections {
8      Sync<msgVote>(E1.right, E2.left);
9      Sync<msgVote>(E2.right, E3.left );
10     Sync<msgVote>(E3.right, E1.left );
11
12     Sync<msgLocal>(C1,query, E1.query );
13     Sync<msgLocal>(C2,query, E2.query );
14     Sync<msgLocal>(C3,query, E3.query );
15   }
16 }
```

5 Conclusion and Future Work

References

1. Abdulla, P., Deneux, J., Stålmarch, 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. Amnell, T., Behrmann, G., Bengtsson, J., D’argenio, P.R., David, A., Fehnker, A., Hune, T., Jeannet, B., Larsen, K.G., Möller, M.O., Others: UPPAAL - Now, Next, and Future. In: Cassez, F., Jard, C., Brigitte, R., Ryan, M.D. (eds.) *Proceedings of MOVEP 2000*. LNCS, vol. 2067, pp. 99–124. Springer (2001)
4. Arbab, F.: Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science* 14(3), 329–366 (2004)
5. Berry, G., Gonthier, G.: The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming* 19(2), 87–152 (1992)
6. Curry, E.: Message-Oriented Middleware. *Middleware For Communications* pp. 1–28 (2004)
7. Elliott, C., Vijayakumar, V., Zink, W., Hansen, R.: National Instruments LabVIEW: A Programming Environment for Laboratory Automation and Measurement. *Journal of the Association for Laboratory Automation* 12(1), 17–24 (2007)
8. Hagit, A., Jennifer, W.: *Distributed computing: fundamentals, simulations, and advanced topics*. John Wiley & Sons (2004)
9. Hoare, C.A.R.: *Communicating Sequential Processes*. Prentice-Hall (1985)
10. Kim, H., Lee, E.A., Broman, D.: A Toolkit for Construction of Authorization Service Infrastructure for the Internet of Things. In: *Proceedings of IoTDI 2017*. pp. 147–158. ACM (2017)
11. 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)
12. Li, Y.: A list of Mediator models that are referenced in this paper, <https://fmgroup.liyi.today/git/MediatE/PaperProposal/tree/master/models>
13. Liu, Z., Morisset, C., Stolz, V.: rCOS: Theory and tool for component-based model driven development. In: Arbab, F., Sirjani, M. (eds.) *Proceedings of FSEN 2009*. LNCS, vol. 5961, pp. 62–80. Springer (2010)
14. Meng, S., Arbab, F., Baier, C.: Synthesis of Reo circuits from scenario-based interaction specifications. *Science of Computer Programming* 76(8), 651–680 (2011)
15. Zou, L., Zhany, N., Wang, S., Fränzle, M., Qin, S.: Verifying Simulink diagrams via a Hybrid Hoare Logic Prover. In: *Proceedings of EMSOFT 2013* (2013)

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 $S_1 = \{s_1, \dots, s_n\}, S_2 = \{s'_1, \dots, s'_n\}$, and the automaton that a transition belongs to by $Automaton(s)$. We try to use an inductive approach to prove the hypothesis that *for each assignment $s \in S_1$ and its corresponding assignment $s' \in S_2$, the shared variables it changes have the same evaluation in their post-configurations*.

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 assigned 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 referenced 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, \dots, s_i have been proved to satisfy the hypothesis, now we are going to prove that s , the first transition where shared variables are referenced in s_{i+1}, \dots, s_n and its corresponding s' also satisfy the hypothesis.
 - (a) In the pre-configuration of s and s' , all the shared variables that are referenced in s and s' have the *SAME* evaluation. (Thanks to the assumption, all assignments to shared variables in s_1, \dots, s_i share the same evaluation (on referenced variables only) with their corresponding assignments in s' . And on the other hand, for any assignments to the referenced shared variables in S_2 , its index in S_1 must be less than s , and in turn satisfy the hypothesis due to the assumption.)
 - (b) In the pre-configuration of s and s' , all the local variables of $Automaton(s)$ have the *SAME* evaluation. (Derived by the same reason as in 1.(c))
 It's apparent that in the post-configuration of s and s' , all the *assigned* shared variables have the *SAME* evaluation.