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 propose a new language Mediator to describe component-based models. Mediator provides a two-step modeling approach. *Automata*, encapsulated with a interface of ports, is the basic behavior unit. *Systems* make it able to declare automata as components or connectors, and then glue them together as a more complex model. With help of Mediator, components and systems can be modeled separately, while the formal nature is still precisely guaranteed. Through some simple examples, we show that this new language can be used in various practical scenarios.

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[6], MathWorks Simulink , and academic tools like Ptolomy[8]. 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[11]. SCADE, based on a synchronous data flow language LUSTRE, is equipped with a powerful toolchain and widely used development of embedded systems. rCOS, on the other hand, is a refinement calculus on object-oriented design.

Existing work[13] 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[9], Uppaal[2]), it requires at least knowledge on automata theory to properly encode the models.

Reo[3], 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.

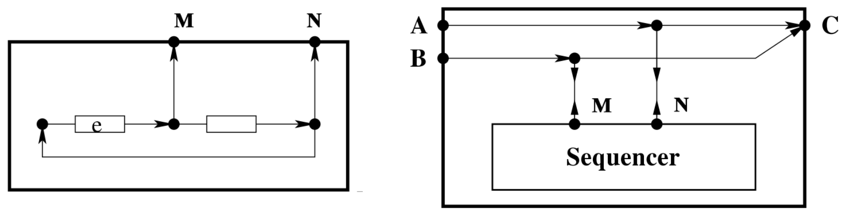

Fig. 1. How A Sequencer Connector is Defined and Reused in Reo [12]

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,

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

1. *Typedef*s 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	`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`

Primitive Types. Table. 1 shows the primitive types supported in Mediator.
Composite Types. Composite types offer 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	$\mathtt{T}_1,\ldots,\mathtt{T}_n$
Union	$\mathtt{T}_1 \vert \ldots \vert \mathtt{T}_n$
Array	`T [length]`
Slice	`T []`
Map	map [$\mathtt{T}_{key}$] $\mathtt{T}_{value}$
Struct	struct { $\mathtt{field}_1$:$\mathtt{T}_1$,..., $\mathtt{field}_n$:$\mathtt{T}_n$ }
Initialized	$\mathtt{T}_{base}$ 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;
   }
   ...
}
```

Parameter Types. On many occasions, you may want to define a generalizable structure that includes a template function or template component. For example, a binary operator that support various operation (+,×, etc.), or an encrypted communication system that can make use of different encryption components. Parameter types make it able to take functions and components (or systems, of course) as a template parameter. But such types will be resolved in instantiation, and hence can only be used in templates of instantiable structures (automata and systems).

1. *An Interface*, denoted by $\mathtt{interface}\ (\mathtt{port}_1\mathtt{:T}_1, \cdots, \mathtt{port}_n\mathtt{:T}_n)$, defines a parameter that could be any *automaton* or *system* that have exactly the same interface (i.e. both number and directions of the ports are matching). Interfaces are only used in templates of *system*s.
2. *A Function*, denoted by $\mathtt{func}\ (\mathtt{arg}_1\mathtt{:T}_1, \cdots, \mathtt{arg}_n\mathtt{:T}_n)\ \mathtt{:}\ \mathtt{T}$ defines a function that have the same argument types and return types. Functions are permitted to show up in templates of *system*s and *automata*.

In Example. 7 we have a system with a *interface* parameter.

2.2 Functions

Functions make it able to encapsulate complex computations and reuse them. In Mediator, the functions are a bit different from common programming languages – they include no control statements but assignments. This design makes functions' behavior more predictable (i.e. it can be simplified into a single mathematical representation). For the same reason, functions have access only to its local variables and arguments.

The abstract syntax tree of functions is shown as follows.

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

Basically, definition of a function includes the following parts.

Template. A function may contains 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). Values of the parameters should be determined in compiling-time. Once a parameter is declared, it can be referenced in all the following language elements, e.g. *a) the following parameter declarations*, *b) arguments and return types* and *c) function statements.*

Name. An identifier that indicates the name of this function.

Type. Type of a function (`func` type in Section. 2.1) is determined by its *a) number of arguments, b) type of arguments and c) type of return value.* Note that here the arguments are read-only. In other words, any assignment to an argument is strictly prohibited.

Body. Body of a function includes an optional set of local variables and a list of ordered statements that describes how the return value is calculated. It must be ended by a `return` statement.

Example 2 (Incline Operation on Bounded Integers). Incline operation of pointers are commonly used in a *round-robin* queue, where storage are reused circularly. The `next` function shows that how pointers in such queues (denoted by a bounded integer) incline.

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

2.3 Automaton : The Basic Behavioral Unit

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

Template. Compared with templates in functions, template in an automaton supports parameters of *function type*.

Name. The identifier of automaton.

Type. Type of an automaton (an `interface` type in Section 2.1) is determined by the *number* and *type*s of its ports. Type of a port in an automaton contains a prefix, either `in` or `out`, indicating the direction of its data-flow, and a normal data type as a suffix. 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.
2. *Adjoint variables* used to describe the status of ports. Syntactically, they are denoted as built-in fields of ports. For example, considering 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.

We require that for an output port the `reqRead` field is read-only and the `reqWrite` field is writable. Similarly, for an input port the `reqRead` field is writable but its `reqWrite` field is read-only. The `value` field can be overwritten only in an output port.

Transitions. In Mediator, behavior of a automaton is described by a set of guarded transitions (groups), with no explicit concept of locations. As shown in Example 3, a *transition* (denoted by *guard* -> *statements*) comprises two parts, a boolean term *guard* that declares the activating condition of this transition, and a (set of) statement(s) that describe how the variables are updated when the transition is fired.

Currently, we have two types of statements supported in automata, they are:

– *Assignment Statement* ($\texttt{var}_1,\dots,\texttt{var}_n$:= $\texttt{term}_1,\dots,\texttt{term}_n$). An assignment statement supports multiple assignments at the same time, where local variables and writable adjoint variables are permitted to be assigned.
– *Synchronizing Statement* (sync $\texttt{port}_1,\dots,\texttt{port}_n$). Synchronizing statements are synchronizing *flag*s used when joining multiple automata. More details about synchronizing statements are introduced in Section 3.3.

With the introduction of shared variables, synchronizing transitions in automata joining is not as easy as in traditional automata where all variables are local. Informally speaking, the synchronizing statements are used to create a proper schedule of assignment statements so that assignments of shared variables are performed before referring them.

Synchronizing statements are also important flags to distinguish external transitions and internal transitions. A transition is called *external* iff. it synchronizes with its environment through some ports, or *internal* otherwise. Literally, all transitions, where synchronizing statements are involved, are *external* transitions. In such transitions, the following rules are strictly required to avoid read/write conflicts.

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

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

Different from a typical automaton, transitions in Mediator automata are organized with *priority*. A transition has higher priority iff. it is placed in front of the other one. And when multiple transitions are activated by the environment, the one with highest priority will be fired first. For example, suppose $g_1 \to S_1, \cdots, g_n \to S_n$ is a list of transitions, we could use an equivalent form to rewrite them as the followings, where priority is not required any more.

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

Example 3 (Transitions in Queue). In a queue, we use internal transitions to capture the changes of environment and perform corresponding updates consistently. For example, the input port A (already defined in Example. 1) becomes ready to read (i.e. `reqRead` set to *true*) when the buffer is not full, and the output port B becomes ready to write when the buffer is not empty, and vice versa. External transitions, on the other hand, mainly show the details of the enqueue and dequeue operations.

```
// 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) -> {
    sync 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); sync B;
}
```

Priority of transitions make the automaton fully deterministic. However, in some cases non-determinism is still more than necessary. *Transition groups* are, consequently, imported to handle such cases. When encapsulated by a group, transitions are unordered and don't ruled by priority. Instead, the group itself is literally ordered w.r.t. other groups and single transitions (basically we can take all single transitions as a trivial transition group).

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

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

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

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

Example 4 (Yet Another Queue Implementation). Let's consider the external transitions in Example. 3 which captures the core behavior of a queue. When both enqueue and dequeue operations are activated, in that example, reading will always be fired first. Such a queue may get stuff up immediately when requests start accumulating. But here we presents another non-deterministic implementation based on *transition groups* to solve this problem.

```
group {
    // enqueue operation (as an external transition)
    (A.reqRead && A.reqWrite) -> {
        sync 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); sync B;
    }
}
```

In this code fragment above, the two transitions are encapsulated in a group. Consequently, firing of the dequeue operation doesn't rely on deactivation of the enqueue operation.

With all the language elements of an automaton properly formalized, now we introduced the formalization of a complete automaton.

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 that are defines in Formalization. 3.*

2.4 System : The Composition Approach

Theoretically, an automaton in Mediator is powerful enough to represent any classical software system (where time and probability are not involved, of course). However, modeling complex systems through a mess of transitions and tons of local variables may become a real disaster.

As we have mentioned previously, Mediator is designed to help the programmers, even nonprofessionals, to enjoy the convenience of formal tools. To achieve this goal, we introduce a new language element called *system*. Basically, a *system* is a textural format of a hierarchical diagram (see in Figure. 2) where automata and smaller systems are naturally organized as different roles (*components* or *connections*). 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*.

Hierarchical diagrams have already been used in various modeling tools (for example, SCADE[1, 4], Simulink and LabVIEW) and formal languages (Reo[3], AADL). However, in most tools, connections are simply synchronous link that seal two ports together. Inspired by Reo, we make it able to declare an automaton as a connection, which lead to more powerful and intuitive diagrams.

Example 5 (Hierarchical Diagram of a Middleware). Figure. 2 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).

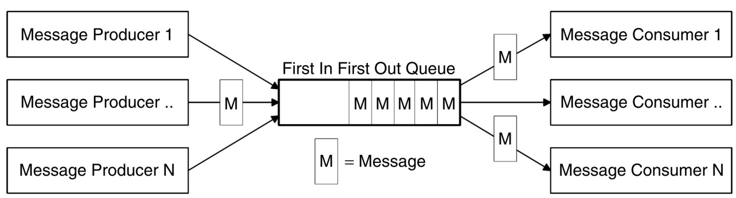

Fig. 2. A Senerio where Queue is used in Message-Oriented Middleware[5]

The abstract syntax tree of *systems* is shown as follows.

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

The *type* of a system (i.e. its template, name, and ports) shares exactly the same form and meaning with *type* of an automaton. This also suggests that system is NOT a special semantics unit, but simply an compositional approach to pile up automata. We declare a system with its template, name and type, then it is implemented by an optional set of *internal nodes*, an optional set of *components* and a set of *connections*.

Template. In templates of systems, all parameters types are supported including *a)* parameters of abstract type `type`, *b)* parameters of primitive types and composite types, and *c)* interfaces and functions.

Name and Type. Exactly the same with *name* and *type* of an automaton.

Components. In the `components` segment, we can declare components of an *interface type*, e.g. name of an automaton (in Example. 6), name of a system, or a parameter of interface type (in Example. 7). Concrete values should be provided in declaration if required. After being declared, ports of a component can be referred simply by `component.portname`.

Connections. Connections, e.g. the arrows in Figure. 2, are used to link between *a) the ports of itself, b) the ports of components, and c) the internal nodes.* We declare the connections is a `connections` segment as shown in the following example. Both components and connections are supposed to execute concurrently as automata.

Internals. In certain cases, we need to combine multiple connections to perform more complicated coordination. Internal nodes, as declared in `internals` segment, are untyped identifiers which are capable to be linked to two other internal nodes or ports. Essentially, data flow in an internal node should always follow the same direction, i.e. an internal node doesn't collect or generate any data, it only receives from one end and forward it simultaneously. The direction, together with the type of an internal node, should be determined when being compiled.

Example 6 (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.

```
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);
    }
}
```

Now we introduce the formalization of systems. Since both components and connections are automata, we will not distinguish them in the formal structure.

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 defined in Formalization. 4(both components and connections), Internals is a set of internal nodes and Links is a set of pairs, where each element is a port or an internal node. A link (p_1, p_2) *suggests that* p_1 *and* p_2 *are linked 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* $\forall x \in V, v(x) \in Dom(type(x))$. *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 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 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**
    $new\_edge \leftarrow$ `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 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*

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

if $\{t_i\}$ don't belong to different automata or $\exists t_i$ is internal **then**
  $t \leftarrow null$
  return
end if
$t.g,\ t.S \leftarrow \bigwedge_i t_i.g,\ \{\}$
$shared\_ports \leftarrow$ all the

$G \leftarrow$ a Graph $\langle V, E\rangle$ {create a dependency graph}
for $i \leftarrow 1, \cdots, n$ **do**
  add $\bot_i, \top_i$ to $G.V$
  $lasts \leftarrow \{\bot_i\}$
  for $j \leftarrow 1, 3, \cdots, len(t_i.S) - 1$ **do**
    $ports \leftarrow$ all the performed ports in $t_i.S_{j+1}$
    for $l \in lasts, p \in ports$ **do**
      if $p \notin G.V$ **then**
        add p to $G.V$
      end if
      add edge $l \xrightarrow{t_i.S_j} p$ to $G.E$
    end for
  end for
  for $l \in lasts$ **do**
    add edge $l \xrightarrow{t_i.S_{len(t_i.S)}} \top_i$ to $G.E$
  end for
end for

if (G comprises a ring) *or* ($\exists v \in G.v$ is a port whose *degree* $\neq 4$) **then**
  $t \leftarrow null$
else
  $t.S \leftarrow \{$ select all the statements in $G.E$ using topological sort $\}$
  replace `perform P` in $t.S$ with `P.reqRead, P.reqWrite := false, false`
end if

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.

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

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

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

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

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

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 Mediator.

[7] proposed an classical algorithm for a typical leader election scenario, as shown in Figure. 3. 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).

The algorithm mainly includes the following steps:

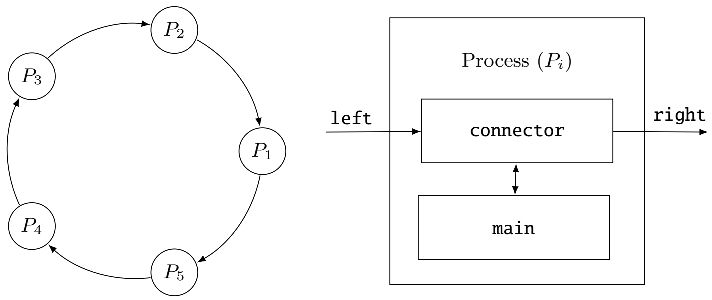

Fig. 3. (a) Topology of a Asynchronous Ring and (b) Structure of a Process

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.

```
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 : out msgLocal
) { ... }
```

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 *Sync*s 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 [10].

Example 7 (A Complete Cluster System with 3 Instances).

```
system <worker: interface (query:in msgLocal)> parallel_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>(C2,query, E2.query );
  Sync<msgLocal>(C3,query, E3.query );
 }
}
```

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. 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)
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. 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)
7. Hagit, A., Jennifer, W.: Distributed computing: fundamentals, simulations, and advanced topics. John Wiley & Sons (2004)

8. 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)
9. 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)
10. Li, Y.: A list of Mediator models that are referenced in this paper, https://fmgroup.liyi.today/git/MediatE/PaperProposal/tree/master/models
11. 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)
12. Meng, S., Arbab, F., Baier, C.: Synthesis of Reo circuits from scenario-based interaction specifications. Science of Computer Programming 76(8), 651–680 (2011)
13. 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 $S_1 = \{s_1, \cdots, s_n\}, S_2 = \{s'_1, \cdots, s'_n\}$, 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 $s_1, \cdots, s_i$ have been proved to satisfy the hypothesis, now we are going to prove that s, the first transition where shared variables are referenced in $s_{i+1}, \cdots, s_n$ 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 $s_1, \cdots, s_i$ 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.