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 formalize component-based system models. Mediator supports a two-step modeling approach. *Automata*, encapsulated with an interface of ports, is the basic behavior unit. *Systems* declare automata as components or connectors, and glue them together. With the help of Mediator, components and systems can be modeled separately and precisely. Through various examples, we show that this language can be used in practical scenarios.

1 Introduction

Component-based software engineering has been prospering for decades. Through proper encapsulation and clearly declared interface, a *component* can be reused by different applications without knowledge of its implementation details.

Currently, there are various tool supports on component-based modeling. NI LabVIEW [11], MATLAB Simulink [6], Ptolomy [7] provide powerful formalism and a large number of built-in component libraries to support commonly-used platforms. However, due to the complexity of models, such tools mainly focus on synthesis and simulation, instead of formal verification. We also have a set of formal tools that prefer simple but verifiable model, e.g. Esterel SCADE [1] and rCOS [10]. SCADE, based on a synchronous data flow language LUSTRE, is equipped with a powerful tool-chain and widely used in development of embedded systems. rCOS, on the other hand, is a refinement calculus on object-oriented designs.

Existing work [12] has shown that, formal verification based on existing industrial tools is hard to realize due to the complexity and non-open architecture of these tools. Unfortunately, unfamiliarity of formal specifications is still the main obstacle hampering programmers from using formal tools. For example, even in the most famous formal modeling tools with perfect graphical user interfaces (like PRISM [8] and Uppaal [2]), sufficient knowledge about automata theory is necessary to properly encode the models.

The channel-based coordination language Reo [3] provides a solution where advantages of both formal languages and graphical representations can be integrated in a natural way. As an exogenous coordination language, Reo doesn't consider the behavior of components. Instead, it takes *connector*s as the first-class citizens. Connectors are organized through a compositional approach to capture complex interaction and communication between components.

In this paper we introduce a new modeling language *Mediator*. Mediator is a hierarchical modeling language that provides proper formalism for both high-level *system* layouts and low-level *automata*-based behavioral units. A rich-featured type system describes complex data structures and powerful automata in a formal way. And automata can be declared as either components or connectors in a system. Both automata and systems are encapsulated with an 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 applications.

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 provides a case study where a commonly used coordination algorithm *leader election* is modeled in Mediator. Section 5 concludes the paper and comes up with some future work we are going to work on.

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,

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

1. A *typedef* specifies an alias name for given types.
2. A *function* defines a customized function that can be reused in other functions, automata and systems.
3. An *automaton* defines an automaton through local variables and transitions.
4. A *system* blocks declare a set of components and connections between them. Both components and connections are described by automata.

2.1 Type System

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

Primitive Types. Table. 1 shows the primitive types supported by Mediator including: *integers and bounded integers, real numbers with arbitrary precision, boolean values, single characters (ASCII only) and finite enumerations.*
Composite Types. Composite types can be used to contruct complex data types from 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.

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` $\texttt{item}_1$`, ...,` $\texttt{item}_n$	`enumname.item`

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

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

- *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* $\{field_1 : T_1, \cdots, field_n : T_n\}$ contains n fields, each has a particular type T_i and a unique identifier $field_i$.
- *Initialized*. An initialized type is used to specify default value of a term with type T_{base}.

Parameter Types. In many situations, we need to define a generalizable automaton or system that includes a template function or template component. For example, a binary operator that support various operation ($+$,$\times$, etc.), or an encrypted communication system that supports different encryption algorithms. Parameter types make it able to take functions and components (or systems, of course) as a template parameter.

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

An example of parameter types can be found in Example. 7.

For simplicity, 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). Queue is a well-known data structure, and it is also used in various message-oriented middlewares. In this example, we 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 the elements in a queue can be either an assigned item or empty. The head and tail pointers 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

Functions encapsulate complex computing steps and reuse them. In Mediator, the functions are a bit different from common programming languages – they include no control statements at all but assignments. This design makes functions' behavior more predictable since it can be resolved into a single formula. For the same reason, functions have access only to its local variables and arguments.

The abstract syntax tree of functions is shown as follows.

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

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 clearly specified during compilation. 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) types 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. The statements are supposed to end up with a `return` statement.

Example 2 (Incline Operation on Queue Pointers). 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) are inclined.

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

2.3 Automaton : The Basic Behavioral Unit

Automata theory is widely used in formal verification. And its variations, finite-state machines for example, are also accepted as modeling tools like NI LabVIEW and Mathworks Simulink/Stateflow.

Here we introduce the *automaton* block as the basic behavioral unit. Compared with other variations, an *automaton* in Mediator contains local variables and typed ports that support complicated behavior and powerful communication. The abstract syntax tree of *automata* is shown as follows.

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

Template. Compared with templates in functions, templates in automata provide support for 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 contains its *direction* either `in` or `out` and its *data type*. To ensure the well-defineness of automata, ports are required to have an *initialized* data type, e.g. `int 0..1 init 0` instead of ~~`int 0..1`~~.

Variables. Two classes of variables are used in an automaton definition. The first one, *local variables*, are declared in the *variables* segment, which can be referenced only in its owner automaton. Nextly, *adjoint variables* are used to describe the status and value 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 any pending *read* or *write* requests on A, and a data field `A.value` indicating the current value of A. Since a port actually connects one automaton to another, in this paper we also call them *shared variables* when focus on sharability.

We require that for an output port the `reqRead` field is read-only and the `reqWrite` field is writable, and for an input port the `reqRead` field is writable but its `reqWrite` field is read-only. Similarly, the `value` field can be assigned when it belongs to an output port.

Transitions. In Mediator, behavior of an automaton is described by a set of guarded transitions (groups), with no explicit concept of locations (actually locations can be easily encoded as local enumeration variables). 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* ($\texttt{var}_1\texttt{,...,var}_n$ `:=` $\texttt{term}_1\texttt{,...,term}_n$). Assignment statements update variables with their new values where only local variables and writable adjoint variables are assignable.
- *Synchronizing Statement* (`sync` $\texttt{port}_1\texttt{,...,port}_n$). Synchronizing statements are synchronizing *flag*s used when joining multiple automata. When followed by multiple ports, it means that the order of synchronizing these ports is arbitrary. 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 help properly schedule the assignment statements from different automata to avoid data conflict.

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. Lit-

erally, all transitions, where synchronizing statements are involved, are *external* transitions. In such transitions, the following rules shoudl be strictly followed.

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`.

As a formal notation, 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.

Not like transitions in typical automata, transitions in Mediator automata are organized with *priority*. A transition has higher priority than all the others that are placed below. When multiple transitions are activated by the environment, the one with highest priority will be fired first. Formally speaking, suppose $g_1 \to S_1, \cdots, g_n \to S_n$ is a list of transitions with priority, we could use an equivalent form to rewrite them as the followings, where priority is not necessary 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 the environment and perform corresponding modifications consistently. For example, the automaton `Queue` (see in Example. 1) tries to *a) read data from its input port A by setting* `A.reqRead` *to* true *when the buffer isn't filled and b) write the buffered data to its output port B when the buffer is not empty*. External transitions, on the other hand, mainly show the implementations of the enqueue and dequeue operations.

```
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) -> A.reqRead := false;
5  !A.reqRead && (buf[phead] == null) -> A.reqRead := true;
6
7  // enqueue operation (as an external transition)
8  (A.reqRead && A.reqWrite) -> {
9      sync 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); sync B;
14 }
```

If all transitions are organzied with priority, automata would be fully deterministic. However, in some cases non-determinism is still more than necessary. Consequently, we introduce *transition groups* to capture non-deterministic behavior. Transitions encapsulated by `group`s are not 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 singleton transition group).

A transition group t_G is formalized as a finite list of guarded transitions $t_G = \{t_1, \cdots, t_n\}, t_i = g_i \rightarrow S_i$ where t_i is a single transition with guard g_i and a set of statements S_i.

Example 4 (Another Queue Implementation). Let's consider the external transitions in Example. 3. In that example, when both *enqueue* and *dequeue* operations are activated, *enqueue* will always be fired first. Such a queue may get stuff up immediately when requests start accumulating, and in turn lead to excessive memory usage. With help of transition groups, here we presents another non-deterministic implementation to solve this problem.

```
1  group {
2      // enqueue operation (as an external transition)
3      (A.reqRead && A.reqWrite) -> {
4          sync A; buf[phead] := A.value; phead := next(phead);
5      }
6      // dequeue operation (as an external transition)
7      (B.reqRead && B.reqWrite) -> {
8          B.value := buf[ptail]; ptail := next(ptail); sync B;
9      }
10 }
```

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

We use a 3-tuple $A = \langle Ports, Vars, Trans_G \rangle$ to represent an automaton in Mediator, 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, where all single transitions are encapsulated as a singleton transition group.

2.4 System : The Composition Approach

Theoretically, automata and their product is capable to model various 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 could become a real disaster.

As mentioned before, Mediator is designed to help the programmers, even nonprofessionals, to enjoy the convenience of formal tools. And that's exactly the reason we introduce the language element *system*. Basically, a *system* is a textural format of a hierarchical diagram (see in Figure. 1) where automata and smaller systems are organized as different roles (*component* or *connection*).

Example 5 (Hierarchical Diagram of a Message-Oriented Middleware). Figure. 1 gives a simple diagram of a message-oriented middleware, where a queue work as a connector to coordinate between the components (message producers and consumers).

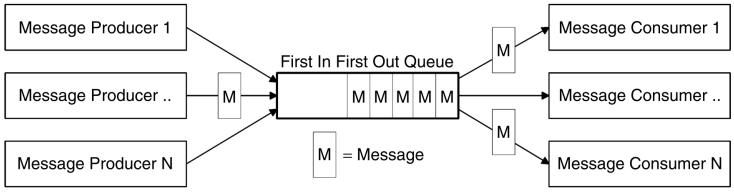

Fig. 1. A Senerio where Queue is used in Message-Oriented Middleware [4]

The abstract syntax tree of *system*s is shown as follows.

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

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 any entity of an *interface type* as components, e.g. an automaton (see in Example. 6), a system, or a parameter of interface type (see in Example. 7). After being declared, ports of a component can be referenced by `component.portname`.

Connections. Connections, e.g. the arrows in Figure. 1, are used to link between *a) the ports of itself, b) the ports of components, and c) the internal nodes*. We declare the connections in the `connections` segments as shown in Example. 6. Both components and connections are supposed to run parallelly 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 weld two connections.

Example 6 (Mediator 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*.

```
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 }
```

A system is denoted by a 4-tuple $S = \langle Ports, Entities, Internals, Links \rangle$ where $Ports$ is a set of ports, $Entities$ is a set of automata or systems (including 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 $\langle p_1, p_2 \rangle$ suggests that p_1 and p_2 are linked together. A well-defined system satisfies the following assumptions:

1. $\forall \langle p_1, p_2 \rangle \in Links$, 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. $\langle p_1, n \rangle, \langle n, p_2 \rangle \in Links$ and p_1, p_2 have the same data type.

3 Semantics

In the section, we introduce the formal semantics of Mediator through the following steps. First we propose the concept *configuration* to describe the state of an automaton. Next we show what the canonical form of the transitions and automata are, and how to canonicalize them. Finally, we define the formal semantics of automata as *labelled transition systems.*

We don't directly formalize a system as a labelled transition system. Instead, we propose an algorithm that flattens its hierarchical structure and generates a corresponding automaton.

3.1 Configurations of Automata

State of a Mediator automaton depends on the values of its *local variables* and *adjoint variables.* First we introduce the definition of *valuation* on a set of variables.

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

Basically, a valuation is a function that maps variables to one of its valid values, where we use $\mathbb{D}$ to denote the set of all values of all types. Now we can introduce *configuration* that snapshots an automaton.

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.*

Now we can mathematically describe the language elements in an automaton:

- *Guard*s of an automaton A are represented by boolean functions on its configurations $g : Conf(A) \to Bool$.
- *Assignment Statement*s of A are represented by functions that maps a configuration to the updated one $s_a : Conf(A) \to Conf(A)$.

3.2 Canonical Form of Transitions and Automata

Different statement combinations may have the same behavior. For example, `a := b; c := d` and `a, c := b, d`. These irregular forms may lead to an extremely complicated and non-intuitive algorithm when joining multiple automata.

To simplify this process, we introduce the *canonical* form of transitions and automata as follows.

Definition 3 (Canonical Transitions). *A transition $t = g \to \{s_1, \cdots, s_n\}$ is canonical iff. its statements $\{s_i\}$ is a non-empty interleaving sequence of assignments and synchronizing statements which start from and end by assignments, e.g.* `a := exp`$_1$`; sync A; b := exp`$_2$`;` $\cdots$ `c := exp`$_3$.

Suppose $g \to \{s_1, \cdots, s_n\}$ is a transition of automaton A, it can be canonicalized through the following steps.

S1 If we find a continuous subsequence $s_i, \cdots, s_j$ (where s_k is an assignment statement for all $k = i, i+1, \cdots, j$, and $j > i$), we merge them as a single one. Since the assignment statements are formalized as functions $Conf(A) \to Conf(A)$, the subsequence $s_i, \cdots, s_j$ can be replaced by $s' = s_i \circ \cdots \circ s_j$.
S2 Keep going with *S1* until there is no further subsequence to merge.
S3 Use identical assignments $id_{Conf(A)}$ to fill the gap between any adjacent synchronizing statements. Similarly, if the statements' list start from or end up with a synchronizing statement, we should also use $id_{Conf(A)}$ to decorate its head and tail.

It's clear that once we found such a continuous subsequence, the merging operation will reduce the number of statements. Otherwise it stops. It's clear that S is a finite set, and the algorithm always terminates within certain time.

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 an automaton canonical. Suppose $Trans_G$ is composed of a set of transition groups t_{G_i}, where the length of t_{G_i} is denoted by l_i,

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

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

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

Then we can generate the new group of transitions with no dependency on priority as followings. To simplify the equations, we use $activated(t_{G_1}, \cdots, t_{G_{n-1}})$ to indicate that at least one group in $t_{G_1}, \cdots, t_{G_{n-1}}$ is activated. It's equivalent form is $activated(t_{G_1}) \vee \cdots \vee activated(t_{G_{n-1}})$.

$$\begin{aligned} Trans'_G = \{\ & g_{11} \to S_{11}, \cdots, g_{1l_1} \to S_{1l_1}, \\ & g_{21} \wedge \neg activated(t_{G_1}) \to S_{21}, \cdots, g_{2l_2} \wedge \neg activated(t_{G_1}) \to S_{2l_2}, \cdots \\ & g_{n1} \wedge \neg activated(t_{G_1}, \cdots, t_{G_{n-1}}) \to S_{n1}, \cdots, \\ & g_{nl_n} \wedge \neg activated(t_{G_1}, \cdots, t_{G_{n-1}}) \to S_{nl_n}\} \end{aligned}$$

3.3 From System to Automaton

System in Mediator provides an approach to construct hierarchical models from automata (declared as *components* and *connectors*). In this section, we present the algorithms that flatten the hierarchical system into a typical automaton.

For a system $S = \langle Ports, Entities, Internals, Links \rangle$, Algorithm. 1 shows it is flattened into an automaton, where we assume that all the entities are canonical automata (we will recursively flatten the entities first if they are also systems).

During the initialization step we refactor all the variables in its entities to avoid name conflicts, and establish the *link*s by replacing occurrence of one port (or internal node) with its corresponding one with reference to *Link*. After preparation of entities, we put all the transitions together, both *internal* ones and *external* ones.

Internal transitions are easy to handle. Since they do not synchronize with other transitions, we directly put all the internal transitions in all entities into the flattened automaton, also as internal transitions.

External transitions, on the other hand, have to synchronize with its corresponding external transitions in other entities. For example, when an automaton

want to read some thing from a input port P_1, there must be another one is writing something to its output port P_2 where P_1 and P_2 are welded in the system.

Algorithm 1 Flattening a System to an Automaton

Require: A system $S = \langle Ports, Entities, Internals, Links \rangle$
Ensure: An automaton A
1: $A \leftarrow$ an empty automaton
2: $A.Ports \leftarrow S.Ports$
3: $Automata \leftarrow$ all the flattened automata of $S.Entities$
4: rename *local variables* in $Automata = \{A_1, \cdots, A_n\}$ to avoid duplicated names
5: **for** $l = \langle p_1, p_2 \rangle \in S.Links$ **do**
6: **if** $p_1 \in S.Ports$ **then**
7: replace all occurrance of p_2 with p_1
8: **else**
9: replace all occurrance of p_1 with p_2
10: **end if**
11: **end for**
12: $ext\_trans \leftarrow \{\}$
13: **for** $i \leftarrow 1, 2, \cdots, n$ **do**
14: $A.Vars \leftarrow A.Vars + A_i.Vars$
15: $A.Trans_G \leftarrow A.Trans_G + Internal(A_i.Trans_G)$
16: $ext\_trans \leftarrow ext\_trans + External(A_i.Trans_G)$
17: **end for**
18: **for** $set\_trans \in P(ext\_trans)$ **do**
19: $new\_edge \leftarrow$ `Schedule`(S, set_trans)
20: **if** $new\_edge \neq null$ **then**
21: $A.Trans_G = A.Trans_G + \{new\_edge\}$
22: **end if**
23: **end for**

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 performed before they are referenced. Basically, this is a topological sorting problem. A detailed algorithm is described in Algorithm 2.

Algorithm. 2 does not always produce a synchronized transitions. Line 25 shows several situations where the synchronization process fails:

1. The dependency graph includes a *ring*, which is a sign of *circular dependencies*. For example, transition `g1 -> {sync A;sync B;}` and transition `g2 -> {sync B;sync A;}`, where both ports require to be triggered first.
2. The dependency graph includes a non-trivial vertex (neither $\perp$ nor $\top$) whose degree is not equal to 4. In other words, a port is not properly synchronized or synchronized with more than two transitions (as mentioned in Section. 2.4, any communication happens only between two automata).

Algorithm 2 Scheduling in a Synchronous Set of External Transitions

Require: A System S, a set of external canonical transitions $t_1, t_2, \cdots, t_n$
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
$t.g,\ t.S \leftarrow \bigwedge_i t_i.g,\ \{\}$

$G \leftarrow$ 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 synchronized 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* ($\exists v \in G.v \backslash S.Ports$ 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 }
  $\forall P \in G.v \backslash S.Ports$ replace `sync P` in $t.S$ with
    `P.reqRead, P.reqWrite := false, false`
end if

Topological sorting, as we all knows, may generate different schedules for the same dependency 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. S_1 and S_2 will lead to the same result when executed under the same configuration).*

3.4 Automaton as Labelled Transition System

With all the language elements properly formalized, now we introduce the formal semantics of *automata* based on *labelled transition system*.

Definition 5 (Labelled Transition System, LTS). *A transition system is a tuple $(S, \Sigma, \rightarrow, s_0)$ where S is a set of states with initial state $s_0 \in S$, Σ is a set of actions, and $\rightarrow \subseteq S \times \Sigma \times S$ is a set of transitions. For simplicity reasons, we use $s \xrightarrow{a} s'$ to denote $(s, a, s') \in \rightarrow$.*

Suppose $A = \langle Ports, Vars, Trans_G \rangle$ is an automaton, its semantics can be captured by a labelled transition system $\langle S_A, \Sigma_A, \rightarrow_A, s_0 \rangle$ where

- S_A is the set of all configurations of A.
- s_0 is the initial configuration where all variables (except `reqRead` and `reqWrite`) are initialized with their default value, and `reqRead` and `reqWrite` are initialized as `false`.
- $\Sigma_A = \{i\} \cup P(Ports)$ is the set of all actions.
- $\rightarrow_A \subseteq S_A \times \Sigma_A \times S_A$ is a set of transitions constructed by the following rules.

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

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

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

$$\frac{\{g \rightarrow \{s\}\} \in Trans_G \text{ is internal}}{(v_{loc}, v_{adj}) \xrightarrow{i}_A s(v_{loc}, v_{adj})} \text{ R-Internal}$$

$$\frac{\begin{array}{c}\{g \rightarrow S\} \in Trans_G \text{ is external, } \{s_1, \cdots, s_n\} \text{ are the assignments in } S \\ \{p_1, \cdots, p_m\} \text{ are the synchronized ports}\end{array}}{(v_{loc}, v_{adj}) \xrightarrow{\{p_1, \cdots, p_m\}}_A s_n \circ \cdots \circ s_1(v_{loc}, v_{adj})} \text{ R-External}$$

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 by the context randomly. And R-InputValue shows that the value of an input port may also be updated.

The rule R-Internal models the internal transitions in $Trans_G$. As illustrated previously, an internal transition doesn't contains any synchronizing 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 another 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.

[5] 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).

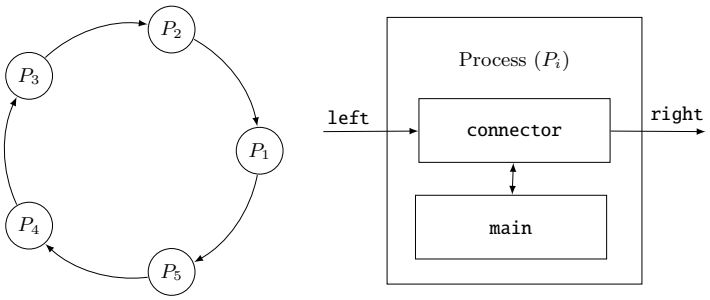

Fig. 2. (a) Topology of a Asynchronous Ring and (b) Structure of a Process

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. 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. 2. 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 [9].

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

A new modeling language Mediator is proposed in this paper to help with component-based software engineering through a formal way. With the basic semantics unit *automata* that capture the formal nature of a component, and

systems for hierarchical composition, the language is easy-to-use for both formal method researchers and system designers.

This paper is a preface of a set of under-development tools. We plan to build a model checking algorithm for a finite subset of Mediator, and then extend it through symbolic approach. A automatic code-generator is also being built to generate platform-specific codes like *Arduino*.

Acknowledgements

The work was partially supported by the National Natural Science Foundation of China under grant no. TBD.

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. Curry, E.: Message-Oriented Middleware. Middleware For Communications pp. 1–28 (2004)
5. Hagit, A., Jennifer, W.: Distributed computing: fundamentals, simulations, and advanced topics. John Wiley & Sons (2004)
6. Hahn, B., Valentine, D.T.: Essential MATLAB for engineers and scientists. pp. 337–351. Academic Press (2016)
7. 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)
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)
9. Li, Y.: A list of Mediator models that are referenced in this paper, https://fmgroup.liyi.today/git/MediatE/PaperProposal/tree/master/models
10. 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)
11. National Instruments, title = LabVIEW, u..h.:
12. 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.