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, are the basic behavior units. *Systems* declare components or connectors through automata, 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.

Keywords: Component-based modeling · Coordination · Formal method

1 Introduction

Component-based software engineering has been prospering for decades. Through proper encapsulations and clearly declared interfaces, *component*s can be reused by different applications without knowledge of their implementation details.

Currently, there are various tool supporting component-based modeling. NI LabVIEW [14], MATLAB Simulink [8] and Ptolomy [10] provide powerful modeling platforms 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. There is also a set of formal tools that prefer simple but verifiable model, e.g. Esterel SCADE [2] and rCOS [12]. 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 [15] 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 [11] and UPPAAL [3]), sufficient knowledge about automata theory is necessary to properly encode the models.

The channel-based coordination language Reo [4] 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

J. Proença and M. Lumpe (Eds.): FACS 2017, LNCS 10487, pp. 1–19, 2017.
DOI: 10.1007/978-3-319-68034-7_1

care about the implementation details of components. Instead, it takes *connector*s as the first-class citizens. Connectors are organized and encapsulated through a compositional approach to capture complex interaction and communication behavior among 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 behavior units. A rich-featured type system describes complex data structures and powerful automata in a formal way. Both components and connectors can be declared through automata to compose a system. Moreover, automata and systems are encapsulated with *a set of input or output ports* (which we call an *interface*) and *a set of template parameters* so that they can be easily reused in multiple applications.

The paper is structured as follows. In Sect. 2, we briefly present the syntax of Mediator and formalizations of the language entities. Then in Sect. 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, represented by a variant of Extended Backus-Naur Form (known as EBNF) where:

- Terminal symbols are written in `monospaced fonts`.
- Non-terminal productions are encapsulated in ⟨*angle brackets*⟩.
- We use "?" to denote "zero or one occurence", "*" to denote "zero or more occurence" and "+" to denote "one or more occurence".

A Mediator *program* is defined as follows:

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

*Typedef*s specify alias for given types. *Function*s define customized functions. *System*s declare hierarchical structures of components and connections between them. Both components and connections are described by *automata* based on local variables and transitions.

2.1 Type System

Mediator provides a rich-featured type system to support 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.*

Table 1. Primitive data types

Name	Declaration	Term example
Integer	`int`	`−1,0,1`
Bounded integer	`int lowerBound .. upperBound`	`−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]`
List	`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`

Composite Types. Composite types can be used to construct complex data types from simpler ones. Several composite patterns are introduced as follows (Table 2):

- *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 different types as a more complicated one.
- *Array* and *List.* An *array* $T[n]$ is a finite ordered collection containing exactly n elements of type T. Moreover, a *list* is an array of which the capacity is not specified, i.e. a list 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 a finite number of fields, each has a unique identifier $field_i$ and a particular type T_i.
- *Initialized.* An initialized type is used to specify default value of a type T_{base} with `term`.

Parameter Types. A generalizable automaton or system that includes a template function or template component needs to be defined on many occasions. For example, a binary operator that supports various operations (+,×, etc.), or an encrypted communication system that supports different encryption algorithms.

Parameter types make it possible to take functions, automata or systems as template parameters. Mediator supports two parameter types:

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* with 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 has the argument types $\texttt{T}_1, \cdots, \texttt{T}_n$ and result types `T`. Functions are permitted to appear in templates of *other functions*, *automata* and *system*s.

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). A queue is a well-known data structure being used in various message-oriented middlewares. In this example, we introduce some type declarations and local variables used in an automaton `Queue` defining the queue structure. As shown in the following code fragment, we declare a singleton enumeration `NULL`, which contains only one element `null`. The buffer of a queue is in turn formalized as an array of `T` or `NULL`, indicating that the elements in the 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, ptail : int 0 .. (size - 1) init 0;
6      }
7      ...
8  }
```

2.2 Functions

Functions are used to encapsulate and reuse complex computation processes. In Mediator, the notion of *functions* is a bit different from most existing programming languages. Mediator functions include no control statements at all but assignments, and have access only to its local variables and arguments. This design makes functions' behavior more predictable. In fact, the behavior of functions in Mediator can be simplified into mathematical functions.

The abstract syntax tree of functions is as follows.

⟨*funcDecl*⟩ ::= `function` ⟨*template*⟩ $^?$ ⟨*identifier*⟩ ⟨*funcInterface*⟩ `{`
`(` `variables` `{` ⟨*varDecl*⟩ * `}` `)`$^?$
`statements` `{` ⟨*assignStmt*⟩ * ⟨*returnStmt*⟩ `}`

⟨*funcInterface*⟩ ::= `(` `(` ⟨*identifier*⟩ `:` ⟨*type*⟩ `)`* `)` `:` ⟨*type*⟩

⟨*assignStmt*⟩ ::= ⟨*term*⟩ `(` `,` ⟨*term*⟩ `)`* `:=` ⟨*term*⟩ `(` `,` ⟨*term*⟩ `)`*

⟨*returnStmt*⟩ ::= `return` ⟨*term*⟩

⟨*varDecl*⟩ ::= ⟨*identifier*⟩ `:` ⟨*type*⟩ `(` `init` ⟨*term*⟩ `)`$^?$

Basically, a function definition includes the following parts.

Template. A function may contain an optional template with 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 referred in all the following language elements, e.g. parameter declarations, arguments, return types and statements.

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

Type. Type of a function is determined by the *number and types of arguments*, together with *the type of its return value.*

Body. Body of a function includes an optional set of local variables and a list of ordered (assignment or return) statements. In an assignment statement, local variables, parameters and arguments can be referenced, but only local variables are writable. The list of statements always ends up with a `return` statement.

Example 2 (Incline Operation on Queue Pointers). Incline operation of pointers are widely used in a *round-robin* queue, where storage are reused circularly. The `next` function shows 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)
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 by modeling tools like NI LabVIEW and Mathworks Simulink/Stateflow.

Here we introduce the notion of *automaton* as the basic behavior 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 *automaton* is as follows.

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

Template. Compared with templates in functions, templates in automata provide support for parameters of *function type.*

Name. The identifier of an automaton.

Type. Type of an automaton 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*. For example, a port P that takes integer values as input is denoted by `P:in int`. To ensure the well-definedness of automata, ports are required to have *initialized* data types, e.g. `int 0..1 init 0` instead of `int 0..1`.

Variables. Two classes of variables are used in an automaton definition. *Local variables* are declared in the *variables* segment, which can be referenced only in its owner automaton. *Port variables*, on the other hand, are shared variables that describe the status and values of ports.

Port variables are denoted as fields of ports. An arbitrary port P has two corresponding Boolean port variables `P.reqRead` and `P.reqWrite` indicating whether there is any pending *read* or *write* requests on P, and a data field `P.value` indicating the current value of P. When automata are combined, port variables are shared between automata to perform communications. To avoid data-conflict, we require that only `reqRead` and `value` fields of input ports, and `reqWrite` fields of output ports are writable. Informally, an automaton only requires data from its input port and writes data to its output port.

Transitions. In Mediator, behavior of an automaton is described by a list of guarded transitions (groups). A *transition* (denoted by *guard* `->` *statements*) comprises two parts, a Boolean term *guard* that declares the activating condition of this transition, and a (sequence of) statement(s) describing how variables are updated when the transition is fired.

We have two types of statements supported in automata:

- *Assignment Statement* ($\texttt{var}_1$,...,$\texttt{var}_n$:= $\texttt{term}_1$,...,$\texttt{term}_n$). Assignment statements update variables with new values where only local variables and writable port variables are assignable.
- *Synchronizing Statement* (`sync` $\texttt{port}_1$,...,$\texttt{port}_n$). Synchronizing statements are used as synchronizing *flag*s when joining multiple automata. In a synchronizing statement, the order of ports being synchronized is arbitrary. For further details, please refer to Sect. 3.3.

A transition is called *external* iff. It synchronizes with its environment through certain ports or *internal* nodes with synchronizing statements. In such transitions, we require that *any assignment statements including reference to an input(output) port should be placed after(before) its corresponding synchronizing statement.*

We use $g \rightarrow S$ to denote a transition, where g is the guard formula and $S = [s_1, \cdots, s_n]$ is a sequence of statements.

Transitions in Mediator automata are literally ordered. Given a list of transitions $g_1 \rightarrow S_1, \cdots, g_n \rightarrow S_n$ where $\{g_{i_j}\}_{j=1,\cdots,m}$ is satisfied, only the transition $g_{min\{i_j\}} \rightarrow S_{min\{i_j\}}$ will be fired. In other words, $g_i \rightarrow S_i$ is fired iff. g_i is satisfied and for all $0 < j < i$, g_j is unsatisfied.

Example 3 (Transitions in Queue). For a queue, we use internal transitions to capture the modifications corresponding to the changes of its environment. For example, the automaton `Queue` tries to:

1. Read data from its input port A by setting `A.reqRead` to *true* when the buffer isn't full.
2. Write the earliest existing buffered data to its output port B when the buffer is not empty.

External transitions, on the other hand, mainly show the implementation details for the enqueue and dequeue operations.

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

// enqueue operation (as an external transition)
(A.reqRead && A.reqWrite) -> {
    sync A; // read data from input port 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; // write data to output port B
}
```

If all transitions are organized with priority, the automata would be fully deterministic. However, in some cases non-determinism is still more than necessary. Consequently, we introduce the notion of *transition group* to capture non-deterministic behavior. A transition group t_G is formalized as a finite set of guarded transitions $t_G = \{t_1, \cdots, t_n\}$ where $t_i = g_i \rightarrow S_i$ is a single transition with guard g_i and a sequence of statements S_i.

Transitions encapsulated in a `group` 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).

Example 4 (Another Queue Implementation). In Example 3, 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 the help of transition groups, here we show another non-deterministic implementation which solves this problem.

```
group {
    (A.reqRead && A.reqWrite) -> {
        sync A; buf[phead] := A.value; phead := next(phead);
    }
    (B.reqRead && B.reqWrite) -> {
        B.value := buf[ptail]; ptail := next(ptail); sync B;
    }
}
```

In the above code fragment, the two external transitions are encapsulated together 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 (the set of port variables are denoted by $Adj(A)$, which can be obtained from $Ports$ directly) and $Trans_G = [t_{G_1}, \cdots, t_{G_n}]$ is a sequence of transition groups, where all single transitions are encapsulated as singleton transition groups.

2.4 System: The Composition Approach

Theoretically, automata and their product is capable to model various classical applications. 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, which is exactly the reason why we introduce the notion of *system* as an *encapsulation mechanism*. Basically, a *system* is the textual representation of a hierarchical diagram where automata and smaller systems are organized as *component*s or *connection*s.

Example 5 (A Message-Oriented Middleware). A simple diagram of a message-oriented middleware [5] is provided in Fig. 1, where a queue works as a connector to coordinate the message producers and consumers.

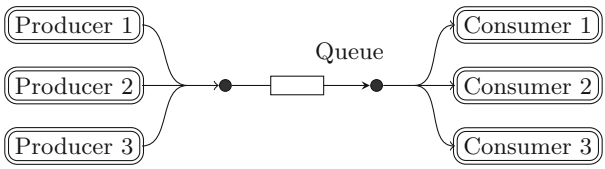

Fig. 1. A scenario where queue is used as message-oriented middleware

The abstract syntax tree of *system*s is as follows:

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

Template. In templates of systems, all the parameter types being supported include: *(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 as *name* and *type* of an automaton.

Components. In `components` segments, we can declare any entity of an *interface type* as components, e.g. an automaton, a system, or a parameter of interface type. Ports of a component can be referenced by `identifier.portName` once declared.

Connections. Connections, e.g. the queue in Fig. 1, are used to connect *(a) the ports of the system itself, (b) the ports of its components, and (c) the internal nodes.* We declare the connections in `connections` segments. Both components and connections are supposed to run as automata in parallel.

Internals. Sometimes we need to combine multiple connections to perform more complex coordination behavior. Internal nodes, declared in `internals` segments, are untyped identifiers which are capable to weld two ports with consistent data-flow direction. For example, in Fig. 1 the two internal nodes (denoted by •) are used to combine a *replicator*, a queue and a *merger* together to work as a multi-in-multi-out queue.

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 of such a pair is either 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 transfers from p_1 to p_2. For example, if $p_1 \in Ports$ is an input port, p_2 could be
   - an output port of the system ($p_2 \in Ports$),
   - an input port of some automaton $A_i \in Automata$ ($p_2 \in A_i.Ports$), or
   - an internal node ($p_2 \in Internals$).
2. $\forall n \in Internals, \exists! p_1, p_2$, s.t. $\langle p_1, n \rangle, \langle n, p_2 \rangle \in Links$ and p_1, p_2 have the same data type.

Example 6 (Model of the System in Fig. 1). In Fig. 1, 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* (details are omitted due to space limit, and can be found at [1]) that produce or consume messages 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 }
```

3 Semantics

In this section, we introduce the formal semantics of Mediator through the following steps. First we use the concept *configuration* to describe the state of an automaton. Next we show what the canonical forms of the transitions and automata are, and how to make them canonical. Finally, we define the formal semantics of automata as *labelled transition systems (LTS)*.

Instead of formalizing systems as LTS directly, we propose an algorithm that flattens the hierarchical structure of a system and generates a corresponding automaton.

3.1 Configurations

States of a Mediator automaton depend on the values of its *local variables* and *port variables*. First we introduce the definition of *evaluation* on a set of variables.

Definition 1 (Evaluation). *An evaluation 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 evaluations of $Vars$ by $EV(Vars)$.*

Basically, an evaluation 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 supported 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 $v_{loc} \in EV(Vars)$ is an evaluation on local variables, and $v_{adj} \in EV(Adj(A))$ is an evaluation on port variables. We use $Conf(A)$ to denote the set of all 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 map configurations to their updated ones $s_a : Conf(A) \to Conf(A)$.

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`. Such irregular forms may lead to an extremely complicated and non-intuitive process 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. $[s_1, \cdots, s_n]$ is a non-empty interleaving sequence of assignments and synchronizing statements which starts and ends with assignments.*

Suppose $g \rightarrow [s_1, \cdots, s_n]$ is a transition of automaton A, it can be made canonical 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 $Conf(A) \rightarrow Conf(A)$, the subsequence $s_i, \cdots, s_j$ can be replaced by $s' = s_j \circ \cdots \circ s_i$[1].

S2. Keep on 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 starts or ends with a synchronizing statement, we should also use $id_{Conf(A)}$ to decorate its head or 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). *$A = \langle Ports, Vars, Trans_G \rangle$ is a canonical automaton iff. (a) $Trans_G$ includes only one transition group and (b) all transitions in this group are canonical.*

Now we show for an arbitrary automaton $A = \langle Ports, Vars, Trans_G \rangle$, how $Trans_G$ is reformed to make A canonical. Suppose $Trans_G$ is a sequence of transition groups t_{G_i}, where the length of t_{G_i} is denoted by l_i,

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

Informally speaking, once a transition in t_{G_i} is activated, all the other transitions in $t_{G_j} (j > i)$ are 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

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

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:

$$activated(t_{G_1}) \vee \cdots \vee activated(t_{G_{n-1}})$$

Then we can generate the new group of transitions with no dependency on priority as followings.

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

[1] The symbol $\circ$ denotes the composition operator on functions.

3.3 From System to Automaton

Mediator provides an approach to construct hierarchical system models from automata. In this section, we present an algorithm that flattens such a hierarchical system into a typical automaton.

For a system $S = \langle Ports, Entities, Internals, Links \rangle$, Algorithm 1 flattens it into an automaton $A_S = \langle Ports, Vars', Trans'_G \rangle$, where we assume that all the entities are canonical automata (they will be flattened recursively first if they are systems). The whole process is mainly divided into 2 steps:

1. Rebuild the structure of the flattened automaton, i.e. to integrate local variables and resolve the internal nodes.
2. Put the transitions together, including both internal transitions and external transitions according to the connections.

First of all, we refactor all the variables in all entities (in $Entities$) to avoid name conflicts, and add them to $Vars'$. Besides, all internal nodes are resolved in the target automaton, and be represented as

$$\{i\_field | i \in Internals, field \in \{\texttt{reqRead, reqWrite, value}\}\} \subseteq Vars'$$

Once all local variables needed are well prepared, we can merge the transitions for both *internal* 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 reads from an input port P_1, there must be another automaton which is writing to its output port P_2, where P_1 and P_2 are welded in the system. An example is presented as follows.

Example 7 (Synchronizing External Transitions). Consider two queues that cooperate on a shared internal node: `Queue(A,B)` and `Queue(B,C)`. Obviously the dequeue operation of `Queue(A,B)` and enqueue operation of `Queue(B,C)` should be synchronized and scheduled. During the synchronization, the basic principle is to make sure that synchronizing statements on the same ports should be aligned strictly.

```
Dequeue Operation:                Enqueue Operation:                After Scheduling:

(B.reqRead && B.reqWrite)-> {     (B.reqRead && B.reqWrite)-> {     (B_reqRead && B_reqWrite)-> {
    B.value := buf[ptail];                                            B_value:=buf1[ptail1];
    ptail := next(ptail);                                             ptail1:=next(ptail1);
    sync B; <---- sync with --    --> sync B; <--- and goes to      --> B_reqRead,B_reqWrite:=
                                        -                                 false,false;
                                                                      buf2[phead2]:=B_value;
                                      buf[phead] := B.value;          phead2:=next(phead2);
}                                     phead := next(phead);       }
                                  }
```

Algorithm 1. `Flat` a System into 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: add $A_i.Vars$ to $A.Vars$
15: **for** $internal \in A_i.Ports$ **do**
16: add $\{internal\texttt{.reqRead}, internal\texttt{.reqWrite}, internal\texttt{.value}\}$ to $A.Vars$
17: **end for**
18: add *all internal transitions* in $A_i.Trans_G$ to $A.Trans_G$
19: add *all external transitions* in $A_i.Trans_G$ to ext_trans
20: **end for**
21: **for** $set\_trans \in \mathcal{P}(ext\_trans)$ **do**
22: add `Schedule`(S, set_trans) to $A.Trans_G$ **if** it is not *null*
23: **end for**

During the synchronization, we refactor the local variables `ptail, phead` and `buf`, and transfer internal node B to a set of local variables. Synchronizing statement `sync B` is aligned between two transitions and in turn leads to the final result, where scheduled synchronizing statements are replaced by its local behavior – to reset its corresponding port variables.

We now formally present the flatting algorithms for systems. In the following we use $\mathcal{P}(A)$ to denote the powerset of A.

In Mediator *systems*, only port variables are shared between automata. During synchronization, the most important principle is to make sure assignments to port variables are performed before the port variables are referenced. Basically, this is a topological sorting problem on dependency graphs. A detailed algorithm is described in Algorithm 2. In this algorithm, we use

- $\bot$ and $\top$ to denote starting and ending of a transition's execution,
- $synchronizable(t_1, \cdots, t_n)$ to denote that the transitions are synchronizable, i.e. they come from different automaton and for each port being synchronized, there are exactly 2 transitions in $t_1, \cdots, t_n$ that synchronize it, and
- $reset\_stmt(p)$ to denote the corresponding statement that resets a port's status `p.reqRead, p.reqWrite := false, false`.

Algorithm 2. `Schedule` a 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 not *synchronizable*$(t_1, \cdots, t_n)$ **then return** $t \leftarrow null$
$t.g$, $t.S$, $G \leftarrow \bigwedge_i t_i.g$, [ ], an empty graph $\langle V, E \rangle$
for $i \leftarrow 1, \cdots, n$ **do**
  add $\bot_i, \top_i$ to $G.V$
  syncs, *ext_syncs* $\leftarrow \{\bot_i\}, \{\}$
  for $j \leftarrow 1, 3, \cdots, len(t_i.S)$ **do**
    add $t_i.S_j$ to $G.V$
    if *ext_syncs* $\neq \{\}$ **then** add '`sync` *ext_syncs*' $\rightarrow t_i.S_j$ to G.E
    for $p \in$ *syncs* **do**
      add edge *reset_stmt*$(p) \rightarrow t_i.S_j$ to $G.E$
    end for
    syncs $\leftarrow$ { all the synchronized ports in $t_i.S_{j+1}$ } $\setminus S.Ports$
    ext_syncs $\leftarrow$ { all the synchronized ports in $t_i.S_{j+1}$ } $\cap S.Ports$
    if $j < len(t_i.S)$ **then**
      for $p \in$ *syncs* **do**
        add *reset_stmt*(p) to $G.V$ if is is not included yet
        add edge $t_i.S_j \rightarrow$ *reset_stmt*(p) to $G.E$
      end for
      if *ext_syncs* $\neq \{\}$ **then**
        add '`sync` *ext_syncs*' to $G.V$
        add edge $t_i.S_j \rightarrow$ '`sync` *ext_syncs*' to $G.E$
      end if
    else
      add edge $t_i.S_j \rightarrow \top_i$ to $G.E$
    end if
  end for
end for
if G comprises a ring **then** $t \leftarrow null$
else $t.S \leftarrow$ [select all the statements in $G.E$ using topological sort]

Algorithm 2 may not always produce a valid synchronized transition. When the dependency graph has a *ring*, the algorithm fails due to *circular dependencies*. For example, transition g_1->{`sync A;sync B;`} and transition g_2->{`sync B;sync A;`} cannot be synchronized where both A, B need to be triggered first.

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

Theorem 1 (Equivalence between Schedules). *If two sequences 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 they are 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). *A labelled 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, 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 LTS $\langle S_A, \Sigma_A, \rightarrow_A, s_0 \rangle$ where

- $S_A = Conf(A)$ is the set of all configurations of A.
- $s_0 \in S_A$ is the initial configuration where all variables (except for `reqRead`s and `reqWrite`s) are initialized with their default value, and all `reqRead`s and `reqWrite`s are initialized as `false`.
- $\Sigma_A = \{i\} \cup \mathcal{P}(Ports)$ is the set of all actions, where i denotes the internal action (i.e. no synchronization is performed).
- $\rightarrow_A \subseteq S_A \times \Sigma_A \times S_A$ is a set of transitions obtained by the following rules.

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

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

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

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

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

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

The rule R-Internal specifies the internal transitions in $Trans_G$. As illustrated previously, an internal transition contains no synchronizing statement. So its canonical form comprises only one assignment s. Firing such a transition will simply apply s to the current configuration.

Meanwhile, the rule R-External specifies the external transitions, where the automaton interact with its environment. Fortunately, since all the environment changes are captured by the first three rules, we can simply regard the environment as another set of local variables. Consequently, the only difference between an internal transition and an external transition is that the later one may contain multiple assignments.

4 Case Study

In modern distributed computing frameworks (e.g. MPI [6] and ZooKeeper [9]), *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 reused to coordinate other components in Mediator.

In [7] the authors proposed a classical algorithm for a typical leader election scenario, as shown in Fig. 2. Distributed processes are organized as an *asynchronous unidirectional* ring where communication takes place only between adjacent processes and following certain direction (indicated by the arrows on edges in Fig. 2(a)).

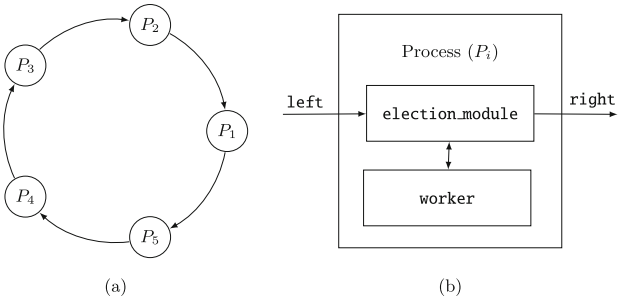

Fig. 2. (a) Topology of an asynchronous ring and (b) Structure of a process

The algorithm has the following steps. At first, each process sends a voting message containing its own *id* to its successor. When receives a voting message, the process will *(a)* forward the message to its successor if it contains a larger *id* than the process itself, or *(b)* ignore the message if it contains a smaller *id* than the process itself, or *(c)* take the process itself as a leader if it contains the same *id* with itself, and send an acknowledgement message to this successor, which will be spread over around the ring.

Here we formalize this algorithm through a more general approach. Leader election is encapsulated as the `election_module`. A computing module `worker`, attached to the `election_module`, is an implementation of the working process.

Two types of messages, `msgVote` and `msgLocal`, are supported when formalizing this architecture. Voting messages `msgVote` are transferred between the processes. 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 *id* is an identifier of the current leader (if it exists). On the other hand, `msgLocal` is used when a process communicates with its corresponding worker.

Example 8 (The Election Module). The following automaton shows how the election algorithm is implemented in Mediator. Due to the space limit, we omit some transitions here. A full version can be found at [1].

```
automaton <id:int> election_module ( left : in msgVote, right : out msgVote,
    query : out msgLocal
) {
    variables {
        leaderStatus : enum { pending, acknowledged } init pending;
        buffer : (voteMsg | NULL) init {vtype: vote, id:id};
        leaderId : (int | NULL) init null;
    }
    transitions {
        (buffer != null)&&(buffer.vtype == vote)&&(buffer.id < id) -> {buffer := null;}
        (buffer != null)&&(buffer.vtype == vote)&&(buffer.id == id) -> {buffer.vtype :=
             ack;}
        (buffer != null)&&(buffer.vtype == ack)&&(buffer.id < id) -> {
            // restart voting if the acknowledged leader has a smaller id
            buffer := { vtype: vote, id: id };
        }
        (buffer != null)&&(buffer.vtype == ack)&&(buffer.id >= id) -> {
            leaderStatus := acknowledged;
            leaderId := buffer.id;
            buffer := buffer.id == id ? null : buffer;
        }
    }
}
```

The following code fragment encodes a parallel program containing 3 *worker*s and 3 *election_module*s to organize the *worker*s. In this example, we do not focus on the implementation details on *worker*s, but hope that any component with a proper interface could be embedded into this system instead.

```
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);
    }
}
```

As we are modeling the leader election algorithm on a synchronous ring, only synchronous communication channels *Sync*s are involved in this example. The implementation details of `Sync` can be found in [1].

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 behavior unit *automata* that captures the formal nature of components and connections, 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 checker for Mediator, and extend it through symbolic approach. An automatic code-generator is also being built to generate platform-specific codes like *Arduino* [13].

Acknowledgements. The work was partially supported by the National Natural Science Foundation of China under grant no. 61532019, 61202069 and 61272160.

References

1. A list of Mediator models. https://github.com/liyi-david/Mediator-Proposal
2. Abdulla, P.A., Deneux, J., Stålmarck, G., Ågren, H., Åkerlund, O.: Designing safe, reliable systems using scade. In: Margaria, T., Steffen, B. (eds.) ISoLA 2004. LNCS, vol. 4313, pp. 115–129. Springer, Heidelberg (2006). doi:10.1007/11925040_8
3. Amnell, T., Behrmann, G., Bengtsson, J., D'Argenio, P.R., David, A., Fehnker, A., Hune, T., Jeannet, B., Larsen, K.G., Möller, M.O., Pettersson, P., Weise, C., Yi, W.: UPPAAL - now, next, and future. In: Cassez, F., Jard, C., Rozoy, B., Ryan, M.D. (eds.) MOVEP 2000. LNCS, vol. 2067, pp. 99–124. Springer, Heidelberg (2001). doi:10.1007/3-540-45510-8_4
4. Arbab, F.: Reo: a channel-based coordination model for component composition. Math. Struct. Comput. Sci. **14**(3), 329–366 (2004)
5. Curry, E.: Message-oriented middleware. In: Mahmoud, Q. (ed.) Middleware for Communications, pp. 1–28. Wiley (2004)
6. Gropp, W., Lusk, E., Thakur, R.: Using MPI-2: Advanced Features of the Message-Passing Interface. MIT Press, Cambridge (1999)
7. Hagit, A., Jennifer, W.: Distributed Computing: Fundamentals, Simulations, and Advanced Topics. Wiley, Hoboken (2004)
8. Hahn, B., Valentine, D.T.: SIMULINK toolbox. In: Essential MATLAB for Engineers and Scientists, pp. 341–356. Academic Press (2016)
9. Junqueira, F.P., Reed, B.C., Serafini, M.: Zab: high-performance broadcast for primary-backup systems. In: Proceedings of DSN 2011, pp. 245–256. IEEE Compute Society (2011)
10. Kim, H., Lee, E.A., Broman, D.: A toolkit for construction of authorization service infrastructure for the internet of things. In: Proceedings of IoTDI 2017, pp. 147–158. ACM (2017)

11. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011). doi:10.1007/978-3-642-22110-1_47
12. Liu, Z., Morisset, C., Stolz, V.: rCOS: theory and tool for component-based model driven development. In: Arbab, F., Sirjani, M. (eds.) FSEN 2009. LNCS, vol. 5961, pp. 62–80. Springer, Heidelberg (2010). doi:10.1007/978-3-642-11623-0_3
13. Margolis, M.: Arduino Cookbook. O'Reilly Media Inc., Sebastopol (2011)
14. National Instruments: Labview. http://www.ni.com/zh-cn/shop/labview.html
15. Zou, L., Zhan, N., Wang, S., Fränzle, M., Qin, S.: Verifying simulink diagrams via a hybrid hoare logic prover. In: Proceedings of EMSOFT 2013, pp. 9:1–9:10. IEEE (2013)