Science of Computer Programming 192 (2020) 102438

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Mediator: A component-based modeling language for concurrent and distributed systems

Yi Li, Weidi Sun, Meng Sun [*]

School of Mathematical Sciences, Peking University, Beijing, China

ARTICLE INFO

Article history:
Received 17 November 2019
Received in revised form 26 February 2020
Accepted 2 March 2020
Available online 3 March 2020

Keywords:
Component-based development
Modeling language
Mediator
Coordination
Distributed systems

ABSTRACT

In this paper we propose a new language *Mediator* to formalize component-based concurrent and distributed system models. *Mediator* supports a two-step hierarchical modeling approach: *Automata*, which provide 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. The distributed *Mediator* and its semantics can be used to capture the inherent real-time and asynchronous behavior in distributed systems. Properties of *Mediator* models can be specified through CTL* formulae that support various families of properties such as safety and liveness, which can be verified using the nuXmv model checker. A leader election example is presented to show that this language is capable for modeling practical scenarios.

1. Introduction

Component-based software engineering has been prospering for decades. Through proper encapsulations and clearly declared interfaces, *components* can be reused by different applications without knowledge of their implementation details. Currently, there are various tools supporting component-based modeling. For example, NI LabVIEW [28], MATLAB Simulink [14] and Ptolomy [17] 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 [3] and rCOS [23].

It has been shown in [32] 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 UPPAAL [4] and PRISM [18]), sufficient knowledge about automata theory is necessary to properly encode the models.

The channel-based coordination language Reo [5] 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 does not care about the implementation details of components. Instead, it takes *connectors* as the first-class citizens. Connectors are or-

[*] Corresponding author.
E-mail addresses: liyi_math@pku.edu.cn (Y. Li), weidisun@pku.edu.cn (W. Sun), sunm@pku.edu.cn (M. Sun).

https://doi.org/10.1016/j.scico.2020.102438
0167-6423/© 2020 Elsevier B.V. All rights reserved.

ganized and encapsulated through a compositional approach to capture complex interaction and communication behavior among components.

In [20] we introduce a hierarchical modeling language *Mediator*, which provides proper formalism for both high-level *system* layouts and low-level *automata*-based behavior units. Both components and connectors can be specified through automata and put together 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. To satisfy the strong requirement for formal modeling and verification techniques of distributed systems, we extend *Mediator* with clocks and corresponding statements in [22] to capture real-time behavior, and use asynchronous message passing in semantics level to describe delayed communication in distributed scenarios. Moreover, to keep the *chain-synchronous* behavior in *Mediator*, we integrate a builtin 2PC transaction protocol in the semantics of distributed *Mediator*.

Component-based modeling has been well investigated in the past decades. For example, rCOS [24] provides a refinement-based modeling framework for component-based and object-oriented systems, in which components can be formally specified and verified, but interactions among components can only be captured by the behavior of components implicitly. In process algebras such as CSP [15] and CCS [27], the representation of interactions between processes is also specified implicitly by using the synchronization between actions of different processes. On the other hand, exogenous coordination languages like Reo [5] focus on coordination of interactions among components, which offer powerful glue mechanisms for implementation of component connectors but ignore the internal computation of components completely. *Mediator* takes both perspectives from internal behavior of components and external interactions between them into account, provides modeling elements explicitly for both of them, and thus becomes a user-friendly language for the modeling of component-based systems. Another language similar to *Mediator* is BIP [9]. Comparing with BIP, *Mediator* focuses on not only the modeling and verification of component-based systems, but also the requirements on application development. For example, we have developed tools for automatic code generation from *Mediator* models to other platforms like Arduino [21,2].

This paper is an extension of [20] and [22] which introduce the first language proposal of *Mediator* and its distributed extension respectively. Comparing with [20] and [22], this paper extends the previous work mainly by:

- a complete rich-featured type system for *Mediator*, including typing and sub-typing rules, more detailed term and type definitions, more convenient connection types, that describes complex data structures and powerful automata in a formal way, and
- the property specifications for *Mediator* models in CTL* and a case study for verifying such properties in the nuXmv model checker [10].

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 shows the distributed extension of *Mediator*. Section 5 specifies the semantics of distributed *Mediator*. Section 6 introduces the property specifications of *Mediator* models in CTL*. Section 7 presents a case study to show the capability of *Mediator* for modeling practical scenarios and verifying its properties with the help of the nuXmv model checker. Section 8 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 $\langle$*angle brackets*$\rangle$.
- “Zero or one occurence” is denoted by “$^?$”, “zero or more occurence” is denoted by “*” and “one or more occurence” is denoted by “$^+$”.

2.1. Program

A *Mediator program* is defined as follows:

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

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

2.2. Types and terms

Mediator provides a rich-featured type system to support various data types that are widely used in most formal modeling languages and programming languages. The formal definition of *Mediator* types are as follows:

$$\begin{aligned}\langle \textit{type} \rangle &::= \langle \textit{primitive type} \rangle \mid \langle \textit{composite type} \rangle \\ \langle \textit{parameter type} \rangle &::= \langle \textit{type} \rangle \mid \langle \textit{abstract type} \rangle \mid \langle \textit{interface type} \rangle \mid \langle \textit{function type} \rangle\end{aligned}$$

According to the grammar, two classes of types $\langle \textit{type} \rangle$ and $\langle \textit{parameter type} \rangle$ are introduced to describe types for different language elements, where $\langle \textit{type} \rangle$ is the definition for types of *terms* which can be further divided into *primitive* and *composite* types, and $\langle \textit{parameter type} \rangle$ is the definition for types of *template parameters* (more details are provided in Section 2.3).

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*. All possible values of a primitive type T is denoted by $Dom(T)$, as shown in the last column of Table 1.

Composite types. Composite types can be used to construct complex data types from simpler ones. Several composite patterns are introduced in Table 2 and interpreted as follows:

- *Tuple*. The *tuple* operator ',' can be used to construct a finite tuple type with several base types.
- *Union*. The *union* operator '|' can be used 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 of which 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 ($+, \times$, 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 the following parameter types:

1. *Abstract type* is denoted by a single keyword `type`. A parameter typed `type` can be any concrete type. For example, to instantiate an automaton `A` (with template `<T:type>`) as a component `c`, we can write `c:A<int>` (as shown in Section 2.5). Abstract types can be used in all templates.
2. *Interface type* is denoted by `interface` $(io_1\ T_1, \cdots, io_n\ T_n)$ where $io_i \in \{$`in, out`$\}$ indicates the directions of the ports. Value of this type could be any *automaton* or *system* with exactly the same interface (i.e. number, types and directions of the ports are a perfect match). Interface types are only used in templates of *systems*.

Table 1
Primitive data types.

Name	Declaration	Value example	*Dom*
Integer	`int`	`-1,0,1`	$\mathbb{Z}$
Bounded integer	`int` $l..r$	`-1,0,1`	$\mathbb{Z} \cap [l, r]$
Real	`real`	`0.1, 1E-3`	$\mathbb{R}$
Boolean	`bool`	`true, false`	$\{true, false\}$
Character	`char`	`'a', 'b'`	All Characters
Enumeration	`enum` $item_1, \cdots, item_n$	$item_1$	$\{item_i \mid i \in 1, \cdots, n\}$

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

Name	Declaration
Tuple	$T_1, \ldots, T_n$
Union	$T_1 \mid \ldots \mid T_n$
Array	$T\ [length]$
List	T `[]`
Map	`map` $[T_{key}]\ T_{value}$
Struct	`struct {` $field_1{:}T_1, \ldots, field_n{:}T_n$ `}`
Initialized	T_{base} `init` $term$

3. *Function type*, denoted by $\texttt{func}\ (T_1, \cdots, T_n)\texttt{:}T$, defines a function that has the argument types $T_1, \cdots, T_n$ and result types T. Function types are permitted to appear in templates of *other functions*, *automata* and *systems*.

Definition 2.1 *(Subtyping).* Suppose the set of all term types and parameter types are denoted by $\mathbb{T}$, subtype relation $\succeq$ on $\mathbb{T}$ is a partial ordering relation where $T \succeq T'$ indicates that T' is a subtype of T. The relation is constructed by the following rules.

$$\frac{}{T \succeq T}\ \textsc{S-Id} \qquad \frac{T_1 \succeq T_2, T_2 \succeq T_3}{T_1 \succeq T_3}\ \textsc{S-Trans} \qquad \frac{l, r \in \mathbb{Z}}{\texttt{int} \succeq \texttt{int}\ l..r}\ \textsc{S-Integer} \qquad \frac{T \succeq T', l \leq l'}{T\ \ [l] \succeq T'\ \ [l']}\ \textsc{S-Array}$$

$$\frac{l, r, l', r' \in \mathbb{Z}, [l', r'] \subseteq [l, r]}{\texttt{int}\ \ l..r \succeq \texttt{int}\ \ l'..r'}\ \textsc{S-BoundedInteger} \qquad \frac{T_1 \succeq T_1', \cdots, T_n \succeq T_n'}{T_1, \cdots, T_n \succeq T_1', \cdots, T_n'}\ \textsc{S-Tuple}$$

$$\frac{}{\texttt{int 0..1} \succeq \texttt{bool}}\ \textsc{S-Bool} \qquad \frac{}{\texttt{real} \succeq \texttt{int}}\ \textsc{S-Real} \qquad \frac{T \succeq T'}{T\ \ \texttt{[]} \succeq T'\ \ \texttt{[]}}\ \textsc{S-List}$$

$$\frac{1 \leq i \leq n}{T_1 \,|\, \ldots \,|\, \mathtt{T}_n \succeq T_1 \,|\, \ldots \,|\, \mathtt{T}_{i-1} \,|\, \mathtt{T}_{i+1} \,|\, \ldots \,|\, \mathtt{T}_n}\ \textsc{S-Union} \qquad \frac{T_1 \succeq T_1', \cdots, T_n \succeq T_n'}{T_1 \,|\, \ldots \,|\, \mathtt{T}_n \succeq T_1' \,|\, \ldots \,|\, \mathtt{T}_n'}\ \textsc{S-Union-2}$$

$$\frac{T_{key}' \succeq T_{key}, T_{val} \succeq T_{val}'}{\texttt{map}\ \ \texttt{[}\mathtt{T}_{key}\texttt{]}\ \ \mathtt{T}_{val} \succeq \texttt{map}\ \ \texttt{[}\mathtt{T}_{key}'\texttt{]}\ \ \mathtt{T}_{val}'}\ \textsc{S-Map} \qquad \frac{T \succeq T', t : T, t' : T'}{T\ \ \texttt{init}\ \ t \succeq T'\ \ \texttt{init}\ \ t'}\ \textsc{S-Initialized}$$

$$\frac{\begin{array}{c} T = \texttt{struct\{}id_1\texttt{:}T_1\texttt{,}\cdots\texttt{,}id_n\texttt{:}T_n\texttt{\}}, 1 \leq i \leq n \\ T' = \texttt{struct\{}id_1\texttt{:}T_1\texttt{,}\cdots\texttt{,}id_{i-1}\texttt{:}T_{i-1}\texttt{,}id_{i+1}\texttt{:}T_{i+1}\texttt{,}\cdots\texttt{,}id_n\texttt{:}T_n\texttt{\}} \end{array}}{T' \succeq T}\ \textsc{S-Struct}$$

$$\frac{T_1 \succeq T_1', \cdots, T_n \succeq T_n'}{\texttt{struct\{}id_1\texttt{:}T_1\texttt{,}\cdots\texttt{,}id_n\texttt{:}T_n\texttt{\}} \succeq \texttt{struct\{}id_1\texttt{:}T_1\texttt{'}\texttt{,}\cdots\texttt{,}id_n\texttt{:}T_n\texttt{'}\texttt{\}}}\ \textsc{S-Struct-2}$$

$$\frac{T_1 \succeq T_1', \cdots, T_n \succeq T_n'}{\texttt{interface}\ \ \texttt{(}io_1\ \ \mathtt{T}_1\texttt{,}\cdots\texttt{,}io_n\ \ \mathtt{T}_n\texttt{)} \succeq \texttt{interface}\ \ \texttt{(}io_1\ \ \mathtt{T}_1'\texttt{,}\cdots\texttt{,}io_n\ \ \mathtt{T}_n'\texttt{)}}\ \textsc{S-Interface}$$

$$\frac{f\texttt{:func}\ \ \texttt{(}T_1\texttt{,}\cdots\texttt{,}T_n\texttt{)}\texttt{:}T, f'\texttt{:func}\ \ \texttt{(}T_1'\texttt{,}\cdots\texttt{,}T_n'\texttt{)}\texttt{:}T' \qquad T_1' \succeq T_1, \cdots, T_n' \succeq T_n, T \succeq T'}{f \succeq f'}\ \textsc{S-Func}$$

Informally speaking, if $T \succeq T'$ then any term of type T' can be assigned to a variable (or parameter) typed T. For example, a term typed `int 0..1` can be assigned to a variable typed `int`.

Subtyping rules are very important in the implementation of *Mediator*. The language is designed with a static typing system where types of all terms are resolved during compilation time. For example, internal nodes (in Section 2.5) are not explicitly typed. If we know that a port typed `int 0..1` is writing to this node and another port typed `int` is reading from this node, it is obvious that the node is well-defined and we can use any type between `int 0..1` and `int` as its type. However, if the types of these two ports are switched, we cannot find any possible type which is less general than `int 0..1` and more general than `int` for this internal node, and hence the model is invalid.

Terms. Formal grammar of terms in *Mediator* are shown as follows.

```
⟨term⟩ ::= ⟨value⟩
         | ⟨identifier⟩ ( (( ⟨term⟩ , )* ⟨term⟩ )? )
         | struct { (( ⟨identifier⟩ = ⟨term⟩ , )* ⟨identifier⟩ = ⟨term⟩ )? }
         | [ (( ⟨term⟩ , )* ⟨term⟩ )? ]
         | map [ (( ⟨term⟩ => ⟨term⟩ , )* ⟨term⟩ => ⟨term⟩ )? ]
         | ⟨term⟩ . ⟨identifier⟩
         | ⟨term⟩ [ ⟨term⟩ ]
```

Table 3
Constructive and destructive terms.

Type	Constructive term	Destructive term
Structure	struct $\{id_1 : T_1$, $id_2 : T_2\}$	$t.id_1$
Map	map [t_1 => v_n, ..., t_1 => v_n]	$t[t_{key}]$
Array&List	[t_1, ..., t_n]	$t[i]$

Table 4
Operators supported in *Mediator*.

Notation	Corresponding function	Supported type
a + b	_add(a,b)	func (int, int):int func (real, real):real
a - b	_sub(a,b)	
a * b	_mul(a,b)	
a / b	_div(a,b)	
a > b	_ge(a,b)	func (int, int):bool func (real, real):bool
a >= b	_geq(a,b)	
a < b	_le(a,b)	
a <= b	_leq(a,b)	
a == b	_eq(a,b)	
a && b	_and(a,b)	func (bool, bool):bool
a || b	_or(a,b)	
!a	_not()	func (bool):bool

The grammar illustrates seven classes of *Mediator* terms. The first one, named *value*, is the set of values of all possible primitive types. Secondly, *function calls* are composed of function identifiers (built-in or custom) and a set of arguments. Moreover, we have *constructive* and *destructive* terms for structures, arrays, lists and maps. Examples of these terms are shown as in Table 3.

Operators in *Mediator* are supported through *notations*. In other words, all the operators in a term are rewritten as function calls with corresponding function identifiers. If the corresponding function with proper type exists in the current environment, we say the term is valid. Table 4 shows the operators supported by *Mediator*.

The typing rules of terms are presented as follows.

$$\frac{t : T, T' \succeq T}{t : T'}\ \textsc{T-Subtype} \qquad \frac{t \in Dom(T),\ T \text{ is } primitive}{t : T}\ \textsc{T-Primitive} \qquad \frac{t_1 : T, \cdots, t_n : T}{[t_1, \cdots, t_n] : T[n]}\ \textsc{T-Array}$$

$$\frac{t : T[n], i : \texttt{int}\ \texttt{0}\ \texttt{..}(n-1)}{t[i] : T}\ \textsc{T-Array-2} \qquad \frac{t_1 : T, \cdots, t_n : T}{[t_1, \cdots, t_n] : T\ [\,]}\ \textsc{T-List} \qquad \frac{t : T\ [\,], i : \texttt{int}}{t[i] : T}\ \textsc{T-List-2}$$

$$\frac{t_1 : T_1, \cdots, t_n : T_n}{\{id_1 = t_1, \cdots, id_n = t_n\} : \texttt{struct}\{id_1 : T_1, \cdots, id_n : T_n\}}\ \textsc{T-Struct}$$

$$\frac{t : \texttt{struct}\{id_1 : T_1, \cdots, id_n : T_n\}, 1 \le i \le n}{t.id_i : T_i}\ \textsc{T-Struct-2}$$

$$\frac{t_1 : T_1, \cdots, t_n : T_n, f : \texttt{func}\ \ (T_1, \cdots, T_n) : T}{f(t_1, ..., t_n) : T}\ \textsc{T-Func}$$

Example 2.1 *(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.

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

2.3. 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.

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

Basically, a function definition includes the following parts:

- *Template*: A function may contain an optional template with a set of *template 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*: The type of a function is determined by the *number and types of arguments*, together with *the type of its return value.*
- *Body*: The 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 2.2 *(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.

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

Here % is the modular operation.

2.4. 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.

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

- *Template*: Compared with templates in functions, templates in automata provide support for parameters of *function type*.
- *Name*: The identifier of an automaton.
- *Type*: The 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 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* (var$_1$,...,var$_n$ = term$_1$,...,term$_n$). Assignment statements update variables with new values where only local variables and writable port variables are assignable.
- *Synchronizing Statement* (sync port$_1$,...,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 Section 3.4.

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 $g \to S$ 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 $g_1 \to S_1, \cdots, g_n \to S_n$, $g_i \to S_i$ is fired iff g_i is satisfied and for all $0 < j < i$, g_j is unsatisfied.

Example 2.3 *(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*[1] to *true* when the buffer is not 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
}
```

[1] *reqWrite* and *reqRead* are properties of the ports, they decide that whether the ports are ready to read and write. We need them to synchronize the input and output of the ports with the help of "Sync".

If all transitions are organized with priority, the automata will be completely deterministic. However, in some cases non-determinism is 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 (we can take all single transitions as a singleton transition group).

Example 2.4 *(Another queue implementation).* In Example 2.3, 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 does not 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 is 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.

2.5. System: the composition approach

Theoretically, automata and their product are 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 *components* or *connections*.

Hierarchical diagrams have already been used in various modeling tools (for example, SCADE [3,8], Simulink [14] and LabVIEW [28]). However, in most tools, connections are simply synchronous link that seal two ports together. Inspired by the connectors in Reo, *Mediator* declares an automaton as a connection, which leads to more powerful and intuitive diagrams.

The abstract syntax tree of *systems* is as follows:

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

The *type* of a system (i.e. its template, name, and ports) shares exactly the same form and meaning with *type* of an automaton. This also suggests that system is NOT a special semantics unit, but simply an compositional approach to pile up automata.

- *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.

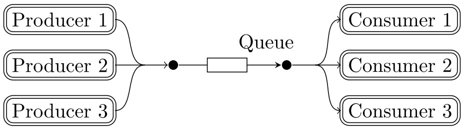


Fig. 1. A Scenario where Queue is used as Message-Oriented Middleware.

Table 5
Options for basic connections.

Options	Semantics
`sync, broadcast`	Accept values from one of the input ports (selected nondeterministically) and broadcasts the accepted values to all the output ports. The sending operation and the receiving operation are supposed to finish simultaneously.
`sync, unicast`	Accept values from one of the input ports (selected nondeterministically) and sends the accepted values to one of the output ports (selected nondeterministically if multiple output ports are prepared). The sending operation and the receiving operation are supposed to finish simultaneously.

`async, broadcast,`
`capacity=n`
	Accept values from one of the input ports (selected nondeterministically) and stores it in a queue whose capacity is specified by `capacity=n`. When all the output ports are prepared, the dequeued values will be broadcasted to all of them.

`async, unicast,`
`capacity=n`
	Accept values from one of the input ports (selected nondeterministically) and stores it in a queue whose capacity is specified by `capacity=n`. All the dequeued values are only sent to one of the output ports (selected nondeterministically if multiple output ports are prepared).

- *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.
- *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.

Example 2.5 *(A message-oriented middleware).* A simple diagram of a message-oriented middleware [11] is provided in Fig. 1, where a queue works as a connector to coordinate the message producers and consumers.

Components and connections are the core parts of a system. Both components and connections are supposed to run as automata in parallel. A typical *Mediator* connection, called *custom connection*, is a reference of an automaton or another system, where the connected ports or nodes are correspondingly joint with the ports of the referenced automaton or system. The syntax of custom connections is shown in the previous syntax tree as the first constructor of ⟨*connectionDecl*⟩. For example, to connect two nodes `M1` and `M2` by a queue with capacity 10, we write `Queue<T,10>(M1, M2)` (see in Example 2.6). In this case we say `M1` is joint with the input port and `M2` is joint with the output port of this queue.

Custom connections are very powerful. However, sometimes it is not convenient enough. For example, in the middleware example we have three producers and three consumers. To connect their input and output ports, we have to put a `merger` channel before and a `replicator` channel after the queue. Even worse, different `merger` automata have to be manually defined for different numbers of input and output ports. To address this problem, we propose another type of connections named *basic connections*. As described in the second constructor of ⟨*connectionDecl*⟩ in the syntax tree, basic connections support arbitrary numbers of input and output ports. The behavior of basic connections depends on the options specified by users. The options being supported by basic connections are listed in Table 5.

The semantics of basic connections are provided via automatically generating corresponding automata according to the options provided. For example, the basic connection `M1-(async,broadcast,capacity=10)->M2` is equivalent to the custom connection `Queue<T, 10>(M1, M2)`. The generated code for this basic connection is shown as follows.

```
automaton <T : type, bufsize: int> CONN ( _AG_PI0: in T , _AG_PO0: out T) {
  variables {
    buf: T | NULL [bufsize] init null;
    front: int 0 .. bufsize - 1 init 0;
    rear: int 0 .. bufsize - 1 init 0;
  }
  transitions {
     _AG_PI0.reqRead != buf[rear] == null -> _AG_PI0.reqRead = buf[rear] == null;
     _AG_PO0.reqWrite != buf[front] != null -> _AG_PO0.reqWrite = buf[front] != null;
     group {
       true -> {
        sync _AG_PI0;
        buf[rear] = _AG_PI0.value;
        rear = rear + 1 % bufsize;
      }
      true -> {
        _AG_PO0.value = buf[front];
        sync _AG_PO0;
        front = front + 1 % bufsize;
      }
    }
  }
}
```

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. Each element of a pair in $Links$ is either a port or an internal node. A link $\langle p_1, p_2\rangle$ suggests that p_1 and p_2 are joint together. A system is well-defined if it satisfies the following assumptions:

1. $\forall\langle p_1, p_2\rangle \in Links$, data transfers from p_1 to p_2, e.g., 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.

There is no difference between the well-defined ports in systems and automata.

Example 2.6 *(*Mediator *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* (Irrelevant details of the two automata are omitted here and can be found at [1]) that produce or consume messages of type *T*.

```
automaton <T:type> Producer (OUT: out T) { ... }
automaton <T:type> Consumer (IN: in T) { ... }

system <T:type> middleware_in_use () {
   components {
      producer_1, producer_2, producer_3 : Producer<T>;
      consumer_1, consumer_2, consumer_3 : Consumer<T>;
   }
   internals M1, M2 ;
   connections {
      Merger<T>(producer_1.OUT, producer_2.OUT, producer_3.OUT, M1);
      Queue<T, 10>(M1, M2);
      Replicator<T>(M2, consumer_1.IN, consumer_2.IN, consumer_3.IN);
   }
}
```

Moreover, we can use basic connections to simplify this model. In this case only one basic connection is used:

```
connections {
   (producer_1.OUT, producer_2.OUT, producer_3.OUT) -(async,broadcast,capacity=10)-> (consumer_1.IN,
       consumer_2.IN, consumer_3.IN);
}
```

3. Semantics

In this section, we introduce the formal semantics of *Mediator*. For automata we introduce their formal semantics in the form of LTS. For systems we propose an algorithm that flattens their hierarchical structures and reschedule them as automata.

3.1. Notations

For clarity reasons, we introduce the formal notations for the core concepts in Section 2. They are: *types*, *variables*, and *ports*:

Types.

A powerful type system which covers most commonly-used data types is provided in *Mediator* [20]. Using $\mathbb{D}$ to represent the universal set of *supported data values*, a type can be denoted by a pair $(dom, init)$ where $dom \subseteq \mathbb{D}$ is a data domain and $init \in dom$ is the initial value.

Variables.

The universal set of variables is denoted by $\mathbb{V}$. The type of a variable is denoted by the typing function:

$$type : \mathbb{V} \to \{(dom, init) \mid dom \subseteq \mathbb{D}, init \in dom\}.$$

Evaluation.

An evaluation of a set of variables V is defined as a function $e : V \to \mathbb{D}$ that satisfies $\forall v \in V, e(v) \in type(v).dom$. We denote the set of all possible evaluations of V by $EV(V)$. Basically, an evaluation is a function that maps a variable to one of its valid values. Several operators on evaluations are defined as follows:

- $\oplus$. The combining operator '$\oplus$' is used to join two evaluations. Suppose $e_1 \in E_{V_1}, e_2 \in E_{V_2}$, combination $e \in E_{V_1 \cup V_2} = e_1 \oplus e_2$ is also an evaluation where:

  $$\forall v \in V_1 \cup V_2, e(v) = \begin{cases} e_1(v) & v \in V_1 \backslash V_2 \\ e_2(v) & v \in V_2 \end{cases}$$

  The combination operator is not commutative. When value of a variable is specified in both operands, the latter one will overwrite its predecessor.
- $\mapsto$. The replacing operator $\mapsto$ overwrites an evaluation with the given variable-value pair. Let $e \in E_V, v_0 \in V, a \in dom(v)$,

  $$\forall v \in V, e[v_0 \mapsto a](v) = \begin{cases} e(v) & v \neq v_0 \\ a & v = v_0 \end{cases}$$
- $|$. An evaluation can be restricted to a sub-evaluation by the restricting operator $|$. Let $e \in E_V$ and $V' \subseteq V$, $e|_{V'} \in E_{V'}$ is also an evaluation where $\forall v \in V', e|_{V'}(v) = e(v)$.
- $=$. We say two evaluations $e_1 \in E_{V_1}$ and $e_2 \in E_{V_2}$ are equivalent, denoted by $e_1 = e_2$, iff $V_1 = V_2$ and $\forall v \in V_1, e_1(v) = e_2(v)$.

Ports.

Ports are important language elements through which entities interact with their contexts. A port p is formalized as a pair (io, t) where $io \in \{in, out\}$ denotes its direction and t is its type. The universal set of ports are denoted by $\mathbb{P}$. Adjoint variables of a port p are denoted by $var(p)$:

$$\begin{aligned} var(p) \subset \mathbb{V} &= \{p.\texttt{reqRead}, p.\texttt{reqWrite}, p.\texttt{value}\} \\ type(p.\texttt{reqRead}) &= type(p.\texttt{reqWrite}) \\ &= (\{true, false\}, false) \\ type(p.\texttt{value}) &= t \end{aligned}$$

We use the predicate *ready* to denote whether a set of ports is ready to be fired:

$$ready(P = \{p_i\}) = \bigwedge_i (p_i.\texttt{reqRead} \wedge p_i.\texttt{reqWrite})$$

Not all the adjoint variables are writable by its owner entity. For an input port p $(p.io = in)$, $p.\texttt{reqWrite}$ and $p.\texttt{value}$ are read-only. Similarly, when $p.io = out$, $p.\texttt{reqRead}$ is read-only. The set of writable variables of a port p is denoted by $var_w(p)$. This notation can be extended to sets of ports. Let $P = \{p_i\} \subseteq \mathbb{P}$, we denote its corresponding writable variable set as $var_w(P) = \bigcup_i var_w(p_i)$.

3.2. Configurations

States of a *Mediator* automaton are mainly composed of:

- The evaluation of local variables and port variables. Once their values are given, the state of an automata is well-determined.
- The synchronization status of ports. For example, whether a port is synchronized in the last transition that leads to the current state. The synchronization status is just a log for passed synchronizations and hence has no further influence on the following behavior. In this paper, it is mainly used for defining semantics of the properties.

Definition 3.1 *(Configuration).* A configuration of an automaton $A = \langle Ports, Vars, Trans_G \rangle$ is defined as a tuple (e, P) where $e \in EV(Vars \cup Adj(A))$ is an evaluation on both local variables and port variables, $P \subseteq Ports$ logs the ports which are synchronized in the latest transition, and $Conf(A)$ is the set of all configurations.

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

- *Guards* of an automaton A represented by Boolean functions on its configurations $g : Conf(A) \to Bool$.
- *Assignment Statements* of A represented by functions that map configurations to their updated ones $s_a : Conf(A) \to Conf(A)$.

3.3. 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 3.2 *(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.

A transition $g \to [s_1, \cdots, s_n]$ in automaton A can be made canonical through the following steps:

S1 If we find a maximal continuous subsequence of assignments $s_i, \cdots, s_j$ (where $j > i$),[2] 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 $s' = s_j \circ \cdots \circ s_i$.[3]

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 sequence of statements 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 $[s_1, \cdots, s_n]$ is a finite sequence, and the algorithm always terminates within certain time.

Definition 3.3 *(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.

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

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

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

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

and use *enabled*$(t_{G_1}, \cdots, t_{G_{n-1}})$ to indicate that at least one group in $t_{G_1}, \cdots, t_{G_{n-1}}$ is enabled. Its equivalent form is:

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

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

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

[2] The subsequence is continuous if s_k is an assignment statement for all $k = i, i+1, \cdots, j$, and maximal if there are no other statements before s_i and after s_j, or if there exist some s_{i-1} and s_{j+1}, which are the statements just before and after this subsequence, both s_{i-1} and s_{j+1} are not assignment statement.

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

3.4. 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 local variables in all sub-entities (in *Entities*) to avoid name conflicts, and add them to $Vars'$. Besides, all internal nodes are replaced by their adjoint variables in the target automaton, and represented as

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

Once all local variables being 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 3.1 *(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);            }
}                                   }
```

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

In the following we use $\mathcal{P}(A)$ to denote the powerset of A. In *Mediator systems*, only port variables are shared between automata. So the basic idea is to replace all internal ports (i.e. ports of sub-entities) by local variables in the flattened automaton.

During the synchronization, the most important principle is to make sure assignments to port variables are performed before the port variables are referenced. This is, in fact, a topological sorting problem on dependency graphs. A detailed algorithm is described in Algorithm 2. In this algorithm, we use

- $\perp$ and $\top$ to denote the starting and ending of a transition's execution,
- $synchronizable(t_1, \cdots, t_n)$ to denote that the transitions $t_1, \cdots, t_n$ 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 "`p.reqRead, p.reqWrite = false, false`" that resets the status of port p.

The generated statement dependency graph of Example 3.1 is shown in Fig. 2. 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, transitions g_1`->{sync A;sync B;}` and g_2`->{sync B;sync A;}` cannot be synchronized where both `A` and `B` need to be triggered first.

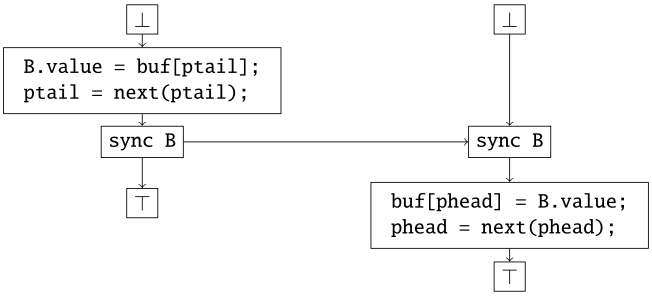


Fig. 2. The generated graph $\langle V, E\rangle$ in Algorithm 2.

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

Theorem 3.1 *(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).*

Proof. First we assume that $S_1 = \{s_1, \cdots, s_n\}$ and $S_2 = \{s'_1, \cdots, s'_n\}$ belong to an automaton A. Considering the LTS-based formal semantics provided in Section 3.5, changes on configurations only come from *assignments*.

We formally describe the execution of these assignment statements through *pre-configurations* and *post-configuration*s. Pre- and post-configurations indicate the configurations of an automaton before and after executing a certain statement. In this proof, we assume that the pre-configurations of s_1 and s'_1 are exactly the same.

We try to use an inductive approach to prove the following hypothesis: *for each assignment $s \in S_1$ and its corresponding assignment $s' \in S_2$, the port variables being changed have the same evaluation in their post-configurations.*

1. Consider the *first* assignment s in S_1 where there are assigned port variables. It is assumed 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 port variables that are also modified by s. *(A port variable can be assigned in only one automaton, thus all assignments that modifies these variables belong to the same transition, and their order is strictly maintained thanks to topological sorting.)*
   (b) s and s' include no reference to other port variables. *(According to Algorithm 2, a port variable can be referenced only when it has been assigned before. As previously assumed, s is the first assignment which modifies a port variable, so s cannot refer any other port variables since all of them have not been assigned yet.)*
   (c) In the pre-configuration of s and s', all the local variables of A have the same evaluation. *(Derived from the same reason in (a), together with the hypothesis that s and s' shares the same pre-configuration.)*

   Consequently, in the post-configuration of s and s', all the port variables have the *same* evaluation.
2. Assume that all assignments (to port 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 port 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 port variables that are referenced in s and s' have the *same* evaluation. *(Thanks to the assumption, all assignments to port 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 port 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 A 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* port variables have the *same* evaluation.

With the help of the hypothesis, we can prove the original theorem as follows:

1. For each port variable v and the last statement in $s \in S_1$ where v is assigned, it's obvious that the corresponding statement $s' \in S_2$ is also the last statement that modifies v. And according to the hypothesis, the value of v after execution of s and s' are exactly the same. Consequently, the final value of v after execution of S_1 and S_2 are also exactly the same.
2. For each local variable v and the last statement $s \in S_1$ where v is assigned, it is easy to prove that in the pre-configurations of s and its corresponding statement $s' \in S_2$, all the local variables and referenced port variables have the same value. *(When a port variable is referenced, the last assignment on this variable has been executed already. And due to the hypothesis, these variables also have the same value in their pre-configurations.)* □

3.5. Automaton as labeled transition system

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

Definition 3.4 *(Labeled transition system).* A labeled 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 $(s, a, s') \in \rightarrow$. More generally, we use $s \rightarrow s'$ to denote $\exists \sigma \in \Sigma : s \xrightarrow{\sigma} s'$.

Suppose $A = \langle Ports, Vars, Trans_G \rangle$ is an automaton, its semantics can be captured by a LTS $[\![A]\!] = \langle S_A, \Sigma_A, \rightarrow_A, s_0 \rangle$ 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 `reqReads` and `reqWrites`) are initialized with their default value, and all `reqReads` and `reqWrites` 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 where we use $Ports_{in}$ and $Ports_{out}$ to denote the sets of all input ports and output ports in $Ports$.

$$\frac{p \in Ports_{in}}{(e, P) \xrightarrow{i}_A (e[p.\texttt{reqWrite} \mapsto \neg p.\texttt{reqWrite}], \varnothing)} \ \textsc{R-InputStatus}$$

$$\frac{p \in Ports_{in},\ val \in Dom(Type(p.value))}{(e, P) \xrightarrow{i}_A (e[p.\texttt{value} \mapsto val], \varnothing)} \ \textsc{R-InputValue} \qquad \frac{\{g \rightarrow \{s\}\} \in Trans_G \text{ is internal}}{(e, P) \xrightarrow{i}_A (s(e), \varnothing)} \ \textsc{R-Internal}$$

$$\frac{p \in Ports_{out}}{(e, P) \xrightarrow{i}_A (e[p.\texttt{reqRead} \mapsto \neg p.\texttt{reqRead}], \varnothing)} \ \textsc{R-OutputStatus}$$

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

The first three rules show how an automaton interacts with its environment. According to Section 2, ports are the connections between automata and their environments. Specifically, changes of port variables directly reflect the information obtained from the environment. We assume that port variables of an automaton can be updated at any time, which is formalized by internal actions.

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. The rule R-External specifies the external transitions, where the automaton interact with its environment. 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. The distributed extension of mediator

In this section we extend *Mediator* with two features: the ability to capture timed behavior and the semantics-level support for asynchronous message passing. For clarity, the extended version of *Mediator* is called distributed *Mediator* hereinafter.

4.1. A motivating example: distributed senders and receivers

A simple message passing model is presented in Fig. 3, which can be formally specified by a *Mediator* system composed of three automata: two message senders (both have one output port *OUT*) and one receiver (has one input port *IN*). A connection called *merger* is used to coordinate their behavior by receiving data items from both senders and sending them to the receiver. In the following, we introduce the core concepts in *Mediator* and show how this message passing model is encoded in *Mediator*.

The following code fragments show how *sender* and *receiver* in Fig. 3 are encoded in *Mediator*. A sender has a single output port OUT typed as bounded integer `int 0..1` and a receiver has a single input port IN with the same type. A

Algorithm 1 `Flattening` a system into an automaton.

```
Require: A system $S = \langle Ports, Entities, Internals, Links \rangle$
Ensure: An automaton $A$
$A \leftarrow$ an empty automaton
$A.Ports \leftarrow S.Ports$
for $E \in S.Entities$ do
    if $E$ is an automaton then
        add $E$ to $Automata$
    else
        add Flat($E$) to $Automata$
    end if
end for
$Automata \leftarrow$ all the flattened automata of $S.Entities$
rename local variables in $Automata = \{A_1, \cdots, A_n\}$ to avoid duplicated names
for $l = \langle p_1, p_2 \rangle \in S.Links$ do
    if $p_1 \in S.Ports$ then
        replace all occurrence of $p_2$ with $p_1$
    else
        replace all occurrence of $p_1$ with $p_2$
    end if
end for
$ext\_trans \leftarrow \{\}$
for $i \leftarrow 1, 2, \cdots, n$ do
    add $A_i.Vars$ to $A.Vars$
    for $p \in A_i.Ports$ do
        add {$p$.reqRead, $p$.reqWrite, $p$.value} to $A.Vars$
    end for
    add all internal transitions in $A_i.Trans_G$ to $A.Trans_G$
    add all external transitions in $A_i.Trans_G$ to $ext\_trans$
end for
for $set\_trans \in \mathcal{P}(ext\_trans)$ do
    add Schedule($S, set\_trans$) to $A.Trans_G$ if it is not null
end for
```

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 $t_i.S_{j+1}$ } $\backslash S.Ports$
        $ext\_syncs \leftarrow$ { 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 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 ]
```

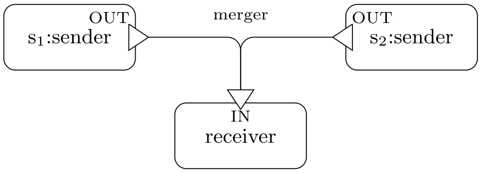


Fig. 3. A simple message passing model.

sender is always prepared to write but a receiver here only takes the *first input from its environment* (this makes it easier when illustrating the distributed extensions).

Both sender and receiver have two transitions, the first one updates the status of ports and the second one performs synchronizing operations. As required in [20], values of an input port can be used only after the port is synchronized. Similarly, values of an output port can be assigned only before the port is synchronized. Through this way, the language is free from data racing.

```
automaton <id:int 0..1> sender (OUT: out int) {
   variables {
      counter : int init 0;
   }
   transitions {
      !OUT.reqWrite -> OUT.reqWrite = true;
      OUT.reqRead && OUT.reqWrite -> {
         OUT.value = id;
         sync OUT;
         counter = counter + 1;
      }
   }
}
```

```
automaton receiver (IN: in int) {
   variables {
      counter: int init 0; last: int;
   }
   transitions {
      counter == 0 -> IN.reqRead = true;
      IN.reqRead && IN.reqWrite -> {
         sync IN;
         last = IN.value;
         counter = counter + 1;
      }
   }
}
```

4.2. Clocks

To capture real-time behavior, we introduce clock variables as a new primitive type. We use $\mathbb{C} \subset \mathbb{V}$ to denote the set of all clock variables, and assume that:

$$\mathbb{R}_{\geq 0} \subset \mathbb{D} \wedge \forall c \in \mathbb{C} : type(c) = (\mathbb{R}_{\geq 0}, 0)$$

Unlike other variables, clock variables cannot be directly modified by assignment statements. A set of new statements to capture real-time behavior with clocks are presented as follows:

reset The reset statement is used to apply reset operation on clocks. Once it is executed, the value of the clock variable being reset will be changed to 0.

sleep The sleep statement forces the system to wait for a certain amount of time.

ite Though it is not related to real-time behavior directly, the if-then-else (ite) statement is introduced for convenience. An ite statement consists of a condition formula, an arbitrary number of statements in its 'then' block (denoted by $\{s_i^t\}$) and the optional 'else' block (denoted by $\{s_i^e\}$).

All the statements supported by distributed *Mediator* are provided in Table 6.

Table 6
Statements supported by distributed mediator.

Statement	Description
`v = expr`	assign `v` by the calculated result of `expr`
`sync p`	send or receive the synchronous flag of `p`
`reset c`	reset the value of clock `c` to 0
`sleep expr`	force the automaton to wait for *expr* time units

`if` (g) `{`s_1^t`;...;`s_m^t`}`
`else {`s_1^e`;...;`s_n^e`}`
	choose one branch to execute according to the value of g

4.3. Synchronization through message passing

Similar as [25,31], we assume that a *reliable asynchronous message passing* interface is provided by the low level software where our model is executed on, i.e.:

- Messages will not get lost or disguised but could be delayed for an arbitrary time. This assumption is reasonable since such interfaces have already been implemented on plenty of platforms, e.g. TCP on basic network, MPI [29] on clusters, etc.
- Orders of events are kept only when they have the same senders and receivers.

Mediator provides support for synchronous communication through ports and `sync` statements. But asynchronous communication can also be encoded by extra automata. For example, a *queue* example is provided in [20] where the queue was implemented as an automaton whose read and write operations cannot be performed in the same transition. However, in this semantics all communication between ports is synchronous, which fails to describe distributed scenarios where reliable synchronization is missing. Consequently, distributed *Mediator* supports asynchronous behavior in the semantics. To be specific, ports are loosely connected through message passing channels. When port variables are modified, notification messages are sent to other entities being connected to this port.

Meanwhile, to maintain the consistency between distributed *Mediator* and the original *Mediator*, we keep synchronous communication in the syntax level, e.g. the `sync` statements. Informally speaking, such statements still represent *synchronous* behavior, but in semantics level such behavior is realized by asynchronous message passing and a 2PC-based transaction protocol (see in Section 4.4 for more details).

When the execution environment of entities becomes distributed, the variables shared by different entities might become inconsistent. For example, if a sender says it is prepared to write to the `OUT` port, the corresponding receiver may not be able to know that immediately. So at certain time points, the status of a port could be different between observers. In this case, *Messages* are used in distributed *Mediator* to model the asynchronous behavior.

What will happen when the execution environment of entities becomes distributed? Obviously, the variables shared by different entities might become inconsistent. For example, if a sender says it is prepared to write to the `OUT` port, the corresponding receiver may not be able to know that immediately. So at certain time points, the status of a port could be different between observers. Similarly, when the sender performs its synchronizing statement `sync OUT`, the receiver may need to wait for a duration before it can receive a synchronizing message and perform `sync IN`.

Messages are used in distributed *Mediator* to model asynchronous behavior.

Definition 4.1 *(Messages).* The set of valid messages $\mathbb{M}(P)$ on a given set of ports $P \subseteq \mathbb{P}$ is defined as:

$$\begin{aligned}\mathbb{M}(P) = {} & \{upd(v,x) | v \in var(P), x \in type(v).dom\} \cup \\ & \{syn(p) | p \in P\} \cup \{ack(p) | p \in P\} \cup \\ & \{fail(p) | p \in P\} \cup \{cmt(p) | p \in P\}\end{aligned}$$

An *updating message* $upd(v,x)$ updates the value of port variable v by x. For example, in the system `prog`, the ports `merger.IN1` and `s1.OUT` are connected. Once `merger` modifies the adjoint variables of `merger.IN1`, updating messages will be sent to `s1` and the corresponding adjoint variables of `s1.OUT` will be modified.

A *synchronizing message* $sync(p)$ is sent from the automaton where an output port is synchronized, and to the automaton where the corresponding input port is waiting for synchronization. For example, from `s1.OUT` to `merger.IN1`. Once the target automaton receives this message, it knows that all updating messages on the synchronized port have been handled, and the value of this port can be used safely.

The other three types of messages are used for handling transactions, which are introduced in Section 4.4.

4.4. Transitions as transactions

A very important feature in *Mediator* is *chain synchronization*, which is inspired by Reo. Intuitively, if transition t_1 synchronizes port A and B, transition t_2 synchronizes port B and C, then t_1 and t_2 must be fired synchronously.

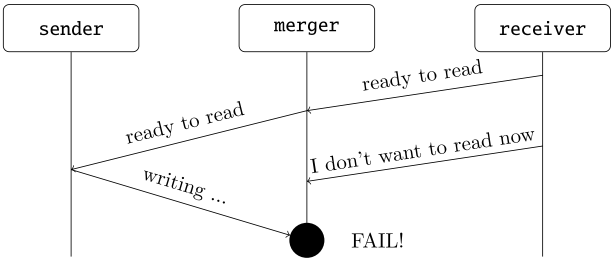


Fig. 4. A synchronization failure.

However, this problem becomes rather difficult in distributed scenarios where communications are usually asynchronous and unreliable. The simple example in Fig. 4 shows that synchronization may fail due to latency of messages. In this case we should be able to rollback a transition that has already started. Such behavior is similar to *transactions* in distributed database systems [19].

It is obvious that two transitions being synchronized through a pair of connected ports either both succeed or both fail. Therefore, *transactions* can be defined as follows:

Definition 4.2 *(Transaction).* A transaction γ on a set of ports $P \subseteq \mathbb{P}$ is denoted by a 3-tuple (P_s, P_a, P_f) in which $P_s, P_a, P_f \subseteq P$. We use P_s to denote the set of ports that have been synchronized, P_a to denote the set of ports that have been acknowledged and P_f to denote the set of ports that have been labeled as *failed*.

Two transactions (P_s, P_a, P_f) and (P'_s, P'_a, P'_f) are *joinable* with a given port p if $\exists p_1 \in (P_s \cup P_a), p_2 \in (P'_s \cup P'_a)$ such that owner of p_1 and p_2 is the same as owner of p. Moreover, the join operation on a set of transactions Γ is defined as follows where $\gamma_i = (P_{s,i}, P_{a,i}, P_{f,i})$ and $\gamma_1 + \gamma_2 = (P_{s,1} \cup P_{s,2}, P_{a,1} \cup P_{a,2}, P_{f,1} \cup P_{f,2})$, where

$$join(\Gamma, p) = \begin{cases} join(\Gamma \backslash \{\gamma_1, \gamma_2\} \cup \{\gamma_1 + \gamma_2\}), p) & (a) \\ \Gamma & otherwise \end{cases}$$

and we have $\gamma_i \in \Gamma$ are joinable with p at (a).

We adapt the two-phase commit protocol (2PC, [7]) here to solve the chain synchronization problem with import of transactions. In the adapted protocol, *systems* play important roles as transaction managers. The adapted 2PC protocol includes two phases:

Prepare.

- An automaton sends a *syn* message when an output port is synchronized through the sync statement. Another automaton who receives this message will also sends a *syn* message back for confirmation.
- Theses *syn* messages are captured by a system (where the automaton belongs), which in turn creates and joins the transactions.
- When a transition is interrupted (port failure occurs) or normally finished, it sends acknowledging messages to the system to acknowledge all successfully synchronized ports. Otherwise it sends $fail$ messages to notify the system which ports suffer from failures.

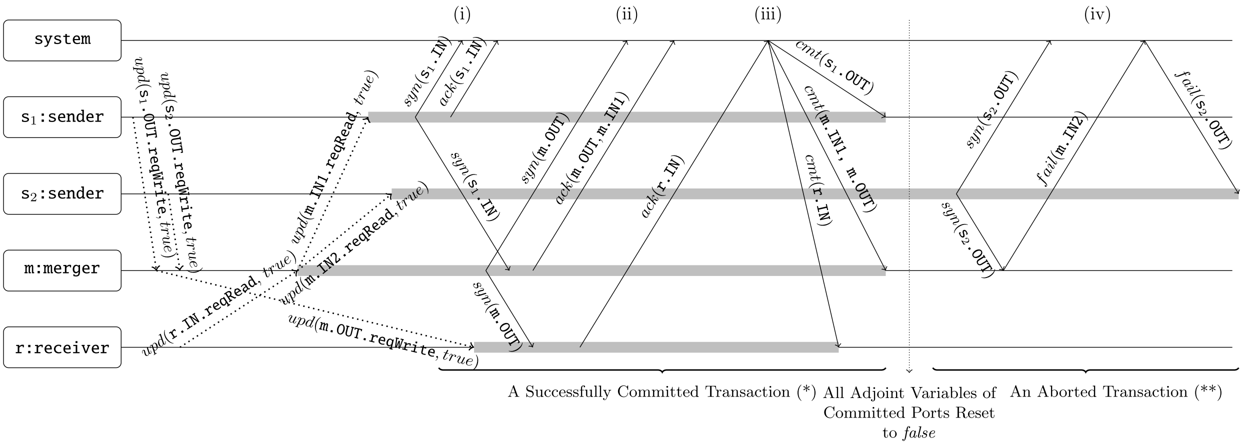
Commit/Abort.

- If all the ports in a transaction are acknowledged, the system will broadcast commit messages to them to commit the transaction.
- If all the ports in a transaction are either acknowledged or failed (at least one is failed), the system will broadcast $fail$ messages to all the acknowledged ports. So that all transitions involved will be aborted and the changes they made will be discarded.

Fig. 5 shows how the adapted 2PC protocol works through one possible execution of the simple message passing model in Fig. 3: (i) transaction $\gamma_1 = (\{s_1.\texttt{OUT}, m.\texttt{IN1}\}, \varnothing, \varnothing)$ is created, (ii) transaction $\gamma_2 = (\{r.\texttt{IN}, m.\texttt{OUT}\}, \varnothing, \varnothing)$ is created and *joined* with γ_1, (iii) the joint transaction is acknowledged and committed, and (iv) transaction $(\{s_2.\texttt{OUT}, m.\texttt{IN2}\}, \varnothing, \varnothing)$ is created but failed.

5. Semantics of distributed mediator

Here we present the operational semantics of distributed *Mediator* through LTS. The notations that will be used in the semantics are introduced in Section 3. We show how the distributed automata and systems are mapped to LTS.



* s_1 successfully passed the data through merger to receiver where 3 transitions are involved

** s_2 has fired the external sending transition but the synchronizing request is rejected by merger

Fig. 5. Sequence Graph of Successfully Committed and Aborted Transactions in Example.

5.1. *Semantics of distributed* Mediator *automata*

We now provide the formal definition of a distributed *Mediator* automaton. The definition essentially follows the one in Section 3 except that the set of clock variables is new in this definition, which captures the real-time dimension of its behavior. For clarity reasons, in this paper we simply use a cost function $cost(s) : \mathbb{S} \to \mathbb{R} \cup \{+\infty\}$ to denote the elapsed time of s.

Definition 5.1 *(Distributed* Mediator *automaton).* A distributed Mediator automaton (DMA) $\mathcal{A}$ is defined as a 4-tuple (P, C, V, T) where

- $P \subset \mathbb{P}$ is a finite set of ports,
- $C \subset \mathbb{V}$ is a finite set of clock variables where $type(c) = (\mathbb{R}, 0)$ for all $c \in C$,
- $V \subset \mathbb{V}$ is a finite set of local variables, and
- T is a finite set of transitions. A transition $t \in T$ is denoted by a pair $t = (g, S)$ where g is the guard expression and $S = [s_1, \cdots, s_m] \in \mathbb{S}^*$ is a finite sequence of statements.

Here we assume that all transitions in T are chosen non-deterministically. In other words, if guard of two transitions are satisfied simultaneously, both transitions have the opportunity to be fired.[4] Now we define the semantics of DMA.

Definition 5.2 *($[\![\mathcal{A}]\!]$).* For an arbitrary DMA $\mathcal{A} = (P, C, V, T)$, its operational semantics $[\![\mathcal{A}]\!]$ is captured by the LTS $[\![\mathcal{A}]\!] = (L, l_0, \Sigma, \to)$ where:

- L is the set of locations, each location in L is denoted by a 7-tuple $(ev, \overline{ev}, ep, ec, S, \gamma, Q)$ where $ev \in E_V$ denotes the evaluation of local variables, $\overline{ev} \in E_V$ denotes the uncommitted evaluation of local variables during the execution of a transition, $ep \in E_{var(P)}$ denotes the evaluation of port variables, $ec \in E_C$ denotes the evaluation of clock variables, $S \in \mathbb{S}^*$ denotes the sequence of statements to be executed, γ denotes the transaction currently being executed and $Q \in \mathbb{M}_P^*$ denotes the message queue,
- $l_0 = (init_V, init_V, init_{var(P)}, init_C, [\cdot], \varnothing, [\cdot])$ is the initial location,
- $\Sigma = \{\texttt{send}\ m | m \in \mathbb{M}_P\} \cup \{\texttt{recv}\ m | m \in \mathbb{M}_P\} \cup \{\tau\} \cup \mathbb{R}_{\geq 0}$ is the set of actions that denotes message passing, internal action and time evolution,
- $\to \subseteq L \times \Sigma \times L$ is the labeled transition relation constructed from the following rules. A non-timed transition $(l, \sigma, l') \in \to$ is also written as $l \xrightarrow{\sigma} l'$ and a timed transition $(l, t, l') \in \to$ is also written as $l \to_t l'$.

Some impotent detailed definitions of the transitions $\to$ are listed as follows.

When initialized, a DMA is not executing any transition, i.e. $l_0.S = [\cdot]$. In this case, the DMA may keep waiting if all transitions cannot be fired (i.e. their guard conditions are not satisfied, captured by Wait) or select a transition to fire

[4] Although in Section 2 we have mentioned that transitions can be organized in both ordered or non-deterministic groups, it is easy to modify their guard formulae and regroup them to non-deterministic ones.

(Select). When the DMA is waiting for a transition which is able to be fired, nothing but time changes.

$$\textsc{Wait}\ \frac{t \in \mathbb{R}_{\geq 0}, \forall (g, S) \in T, t' \in [0, t] : ev \oplus ep \oplus (ec + t') \not\models g}{(ev, ev, ep, ec, [\cdot], (\varnothing, \varnothing, \varnothing), Q) \rightarrow_t (ev, ev, ep, ec + t, [\cdot], (\varnothing, \varnothing, \varnothing), Q)}$$

When a transition is selected, its statements will be put into the execution sequence. Here we use $ec + t$ to denote the time evolution t of an evaluation ec on clock variables, and $e = e_1 \oplus e_2$ to denote the combination of two evaluations (assuming that their domains are disjoint).

$$\textsc{Select}\ \frac{(g, S) \in T, ev \oplus ep \oplus ec \models g}{(ev, ev, ep, ec, [\cdot], (\varnothing, \varnothing, \varnothing), Q) \xrightarrow{\tau} (ev, ev, ep, ec, S, (\varnothing, \varnothing, \varnothing), Q)}$$

Execution of different statements is described by the following rules. Some of them consume messages from the queue and some others generate new messages and send them through the send actions.

A synchronizing statement sync p on an input port p may have two different behavior. When a synchronizing message $syn(p)$ is already on the top of the message queue, the statement consumes $syn(p)$ and succeeds, as shown in Sync-In.

$$\textsc{Sync-In}\ \frac{p.\texttt{reqRead} \wedge p.\texttt{reqWrite}, p.io = in, \gamma = (P_s, P_a, \varnothing), \gamma' = (P_s \cup \{p\}, P_a, \varnothing)}{(ev, \overline{ev}, ep, ec, [\texttt{sync}\ \ p]^\frown S, \gamma, [syn(p)]^\frown Q) \xrightarrow{\text{send }\ syn(p)} (ev, \overline{ev}, ep, ec, S, \gamma', Q)}$$

Otherwise, the statement will block the DMA until the synchronizing message finally comes (Sync-WaitIn).

$$\textsc{Sync-WaitIn}\ \frac{p.\texttt{reqRead} \wedge p.\texttt{reqWrite}, p.io = in, t \in \mathbb{R}_{\geq 0}}{(ev, \overline{ev}, ep, ec, [\texttt{sync}\ \ p]^\frown S, \gamma, [\cdot]) \rightarrow_t (ev, \overline{ev}, ep, ec, [\texttt{sync}\ \ p]^\frown S, \gamma, [\cdot])}$$

When the sync statement performs on an output port p, it always succeeds immediately and sends out a corresponding synchronizing message $syn(p)$.

$$\textsc{Sync-Out}\ \frac{p.\texttt{reqRead} \wedge p.\texttt{reqWrite}, p.io = out, \gamma = (P_s, P_a, \varnothing), \gamma' = (P_s \cup \{p\}, P_a, \varnothing)}{(ev, \overline{ev}, ep, ec, [\texttt{sync}\ \ p]^\frown S, \gamma, Q) \xrightarrow{\text{send }\ syn(p)} (ev, \overline{ev}, ep, ec, S, \gamma', Q)}$$

Assignments on variables behave differently based on their types. When a local variable is assigned, nothing happens except for the change $\overline{ev}$. Otherwise, the DMA sends out a upd message, telling all the other entities who contain a copy of this variable to update their values.

$$\textsc{Assign-Ext}\ \frac{e = \overline{ev} \oplus ep \oplus ec, v \in var_w(P), m = upd(v, expr(e))}{(ev, \overline{ev}, ep, ec, [v\ =\ expr]^\frown S, \gamma, Q) \xrightarrow{\text{send }\ m} (ev, \overline{ev}, ep[v \mapsto expr(e)], ec, S, \gamma, Q)}$$

$$\textsc{Assign-Int}\ \frac{e = \overline{ev} \oplus ep \oplus ec, v \in V}{(ev, \overline{ev}, ep, ec, [v\ =\ expr]^\frown S, \gamma, Q) \xrightarrow{\tau} (ev, \overline{ev}[v \mapsto expr(e)], ep, ec, S, \gamma, Q)}$$

A clock is reset by the reset statement.

$$\textsc{Reset}\ \frac{c \in C}{(ev, \overline{ev}, ep, ec, [\texttt{reset}\ \ c]^\frown S, \gamma, Q) \xrightarrow{\tau} (ev, \overline{ev}, ep, ec[c \mapsto 0], S, \gamma, Q)}$$

A sleep statement forces the DMA to wait for a certain number of time units, which is calculated by evaluating the following expression.

$$\textsc{Sleep}\ \frac{e = \overline{ev} \oplus ep \oplus ec, t = expr(e)}{(ev, \overline{ev}, ep, ec, [\texttt{sleep}\ \ expr]^\frown S, \gamma, Q) \rightarrow_t (ev, \overline{ev}, ep, ec + t, S, \gamma, Q)}$$

An if-then-else statement evaluates the condition expression and selects the following branch to execute. The selected sequence of statements will be pushed on the top of S.

$$\textsc{If-Then-Else}\ \frac{\begin{array}{c} e = \overline{ev} \oplus ep \oplus ec, S = [\texttt{if}\ \ (g)\ \ \texttt{then}\ \ S_1\ \ \texttt{else}\ \ S_2]^\frown S_0 \\ e \models g \rightarrow (S' = S_1{}^\frown S_0), e \not\models g \rightarrow (S' = S_2{}^\frown S_0) \end{array}}{(ev, \overline{ev}, ep, ec, S, \gamma, Q) \xrightarrow{\tau} (ev, \overline{ev}, ep, ec, S', \gamma, Q)}$$

Receive shows how a DMA receive messages from its environment through the `recv` actions. All messages received will be stored in the queue. Update shows how the evaluation of port variables are modified according to the *upd* messages.

$$\textsc{Receive}\ \frac{m \in \mathbb{M}_P}{(ev, \overline{ev}, ep, ec, S, \gamma, Q) \xrightarrow{\texttt{recv}\ m} (ev, \overline{ev}, ep, ec, S, \gamma, Q^\frown[m])}$$

$$\textsc{Update}\ \frac{}{(ev, \overline{ev}, ep, ec, S, \gamma, [upd(v, x)]^\frown Q) \xrightarrow{\tau} (ev, \overline{ev}, ep[v \mapsto x], ec, S, \gamma, Q)}$$

The following rules show how a transition successfully finishes as a transaction. After all statements being executed, the DMA tells the environment that all synchronized ports have been acknowledged by Ack. After acknowledging all of them, the DMA waits for *cmt* messages from the context, as illustrated in Commit. When all the ports involved are committed, the transition will be successfully committed and evaluation of local variables ev will be overwritten by $\overline{ev}$.

$$\textsc{Ack}\ \frac{\gamma = (P_s, P_a, P_f),\ p \in P_s,\ \gamma' = (P_s \backslash \{p\}, P_a \cup \{p\}, P_f)}{(ev, \overline{ev}, ep, ec, [\cdot], \gamma, Q) \xrightarrow{\text{send } ack(p)} (ev, \overline{ev}, ep, ec, [\cdot], \gamma', Q)}$$

$$\textsc{Commit}\ \frac{\gamma = (\varnothing, P_a, \varnothing),\ p \in P_a,\ \gamma' = (\varnothing, P_a \backslash \{p\}, \varnothing)}{(ev, \overline{ev}, ep, ec, [\cdot], \gamma, [cmt(p)]^\frown Q) \xrightarrow{\tau} (ev, \overline{ev}, ep, ec, [\cdot], \gamma', Q)}$$

$$\textsc{Succeed}\ \frac{ev \neq \overline{ev}}{(ev, \overline{ev}, ep, ec, [\cdot], (\varnothing, \varnothing, \varnothing), Q) \xrightarrow{\tau} (\overline{ev}, \overline{ev}, ep, ec, [\cdot], (\varnothing, \varnothing, \varnothing), Q)}$$

On the other hand, a DMA may face a port failure on the following cases: trying to *synchronize an unprepared output port* (Fail-1), handling a *syn* message that *synchronizes an unprepared or wrong input port* (Fail-2) and a currently acknowledged port being noticed as failed (Fail-3) by the environment. In the first two cases a failure message should be sent to its environment.

$$\textsc{Fail-1}\ \frac{\neg(p.\texttt{reqRead} \wedge p.\texttt{reqWrite}) \qquad \gamma = (P_s, \varnothing, \varnothing),\ \gamma' = (P_s, \varnothing, \{p\})}{(ev, \overline{ev}, ep, ec, [\mathbf{sync}\,p]^\frown S, \gamma, Q) \qquad \xrightarrow{\text{send } fail(p)} (ev, \overline{ev}, ep, ec, [\cdot], \gamma', Q)}$$

$$\textsc{Fail-2}\ \frac{(S = [\mathbf{sync}\,p]^\frown S',\ p' \neq p) \vee \neg(p'.\texttt{reqRead} \wedge p'.\texttt{reqWrite})}{(ev, \overline{ev}, ep, ec, S, \gamma, [syn(p')]^\frown Q) \qquad \xrightarrow{\text{send } fail(p')} (ev, \overline{ev}, ep, ec, S, \gamma, Q)}$$

$$\textsc{Fail-3}\ \frac{\gamma = (\varnothing, P_a, P_f),\ p \in P_a,\ \gamma' = (\varnothing, P_a \backslash \{p\}, P_f \cup \{p\}),}{(ev, \overline{ev}, ep, ec, S, \gamma, [fail(p)]^\frown Q) \xrightarrow{\tau} (ev, \overline{ev}, ep, ec, [\cdot], \gamma', Q)}$$

When the environment (the system where this DMA is declared) receives a failure message, it is supposed to mark the corresponding transaction as *failed* and broadcast failure messages to all the other ports in this transaction, as in Section 5.2. And in the DMA, when all ports currently synchronized are labeled as *failed*, the transition as a transaction will be aborted, hence all modification on local variables will be rolled back (Abort).

$$\textsc{Abort}\ \frac{P_f \neq \varnothing}{(ev, \overline{ev}, ep, ec, [\cdot], (\varnothing, \varnothing, P_f), Q) \xrightarrow{\tau} (ev, ev, ep, ec, [\cdot], (\varnothing, \varnothing, \varnothing), Q)}$$

5.2. Semantics of Mediator *systems*

An entity $\mathcal{E}$ could be either a DMA or a system. Considering the definition of DMA and systems, we assume that $\mathcal{E}$ is a tuple that contains at least a set of ports, denoted by $\mathcal{E}.P \in \mathbb{P}$, and semantics of $\mathcal{E}$ is captured by LTS $[\![\mathcal{E}]\!]$.

Definition 5.3 *(Systems).* A system $\mathcal{S}$ is defined as a 3-tuple $\mathcal{S} = (P, \{\mathcal{E}_i\}, \Rightarrow)$ where $P \subseteq \mathbb{P}$ is a finite set of ports, $\{\mathcal{E}_i\}$ is a finite set of entities (which will be referred as *sub-entities* hereinafter) and $\Rightarrow \subseteq P_{all} \times P_{all}$ is a binary relation that describes the dataflow direction between the ports (both ports of the system itself and ports of its sub-entities) where $P_{all} = P \cup \bigcup_i \mathcal{E}_i.P$. For all $(p, p') \in \Rightarrow$, we have

$$\begin{aligned} (p \in P \rightarrow p.io = in \wedge p \notin P \rightarrow p.io = out) \quad \wedge \\ (p' \in P \rightarrow p'.io = out \wedge p' \notin P \rightarrow p'.io = in) \end{aligned}$$

For simplicity we also denote $(p, p') \in \Rightarrow$ by $p \Rightarrow p'$ which indicates that p and p' are connected and p writes data to p'. If $p \in P$ we require $p.io = in$, otherwise we require $p.io = out$ because an output port of a sub-entity actually provides data to the system.

Definition 5.4 *(Dependency between port variables).* Given a system $\mathcal{S} = (P, \{\mathcal{E}_i\}_{i=1,\cdots,n}, \Rightarrow)$ and two port variables $v, v' \in var(P_{all})$, we say v *depends on* v' under $\mathcal{S}$ iff

$$\begin{aligned} &\exists p \Rightarrow p'.(v = p.\texttt{reqRead} \wedge v' = p'.\texttt{reqRead}) \\ \vee\, &\exists p' \Rightarrow p.(v = p.\texttt{reqWrite} \wedge v' = p'.\texttt{reqWrite}) \\ \vee\, &\exists p' \Rightarrow p.(v = p.\texttt{value} \wedge v' = p'.\texttt{value}) \end{aligned}$$

Definition 5.5 *($[\![\mathcal{S}]\!]$).* Semantics of a system $\mathcal{S} = (P, \{\mathcal{E}_i\}_{i=1,\cdots,n}, \Rightarrow)$, denoted by $[\![\mathcal{S}]\!]$, is represented by a labeled transition system $(L, l_0, \Sigma, \rightarrow)$ where:

- $L = \{(st, \Gamma)\}$ where st is a function that maps each sub-entity $\mathcal{E}_i$ to a location in $[\![\mathcal{E}_i]\!].L$, and Γ is a set of transactions,
- $l_0 = (st_0, \varnothing)$ where $st_0(\mathcal{E}_i) = [\![\mathcal{E}_i]\!].l_0$,
- $\Sigma = \{\texttt{send}\ m | m \in \mathbb{M}_P\} \cup \{\texttt{recv}\ m | m \in \mathbb{M}_P\} \cup \{\tau\} \cup \mathbb{R}_{\geq 0}$ is the set of actions that denotes message passing, internal action and time evolution (the same as actions in semantics of DMA),
- $\rightarrow$ is the transition relation constructed from the following rules. A non-timed transition $(l, \sigma, l') \in\rightarrow$ is denoted by $l \xrightarrow{\sigma} l'$ and a timed transition $(l, t, l') \in\rightarrow$ is denoted by $l \rightarrow_t l'$.

Update messages sent from a sub-entity can be received by another sub-entity (Update-1) or forwarded to the external environment (Update-2), depending on how the ports are connected. Similarly, the system can also forward an update message from the environment to its sub-entities (Update-3).

$$\textsc{Update-1}\ \frac{\begin{array}{c} i, j \in \{1, \cdots, n\}\ l_i \in [\![\mathcal{E}_i]\!].L, l_j \in [\![\mathcal{E}_j]\!].L, v' \textit{ depends on } v \\ st(\mathcal{E}_i) \xrightarrow{\texttt{send}\ upd(v,x)} l_i,\ st(\mathcal{E}_j) \xrightarrow{\texttt{recv}\ upd(v',x)} l_j \end{array}}{(st, \Gamma) \xrightarrow{\tau} (st[\mathcal{E}_i \mapsto l_i][\mathcal{E}_j \mapsto l_j], \Gamma)}$$

$$\textsc{Update-2}\ \frac{i \in \{1, \cdots, n\}, l \in [\![\mathcal{E}_i]\!].L, v' \in var(P) \textit{ depends on } v \in var(\mathcal{E}_i.P), st(\mathcal{E}_i) \xrightarrow{\texttt{send}\ upd(v,x)} l}{(st, \Gamma) \xrightarrow{\texttt{send}\ upd(v',x)} (st[\mathcal{E}_i \mapsto l], \Gamma)}$$

$$\textsc{Update-3}\ \frac{i \in \{1, \cdots, n\}, l \in [\![\mathcal{E}_i]\!].L, v \in var(\mathcal{E}_i.P) \textit{ depends on } v' \in var(P), st(\mathcal{E}_i) \xrightarrow{\texttt{recv}\ upd(v,x)} l}{(st, \Gamma) \xrightarrow{\texttt{recv}\ upd(v',x)} (st[\mathcal{E}_i \mapsto l], \Gamma)}$$

The Sync rules updates the transaction set Γ according to the synchronized ports. Sync-1 and Sync-2 describe how ports of sub-entities are synchronized. Sync-3 and Sync-4 show how *syn* messages of external ports are forwarded to the environment. Sync-5 and Sync-6 show how *syn* messages of external ports are received from the environment.

$$\textsc{Sync-1}\ \frac{\begin{array}{c} i, j \in \{1, \cdots, n\}, p \in \mathcal{E}_i.P, p' \in \mathcal{E}_j.Pp \Rightarrow p', l_i \in [\![\mathcal{E}_i]\!].L, l_j \in [\![\mathcal{E}_j]\!].L \\ \Gamma' = join(\Gamma \cup \{(\{p, p'\}, \varnothing, \varnothing)\}, p),\ st(\mathcal{E}_i) \xrightarrow{\texttt{send}\ syn(p)} l_i, st(\mathcal{E}_j) \xrightarrow{\texttt{recv}\ syn(p')} l_j \end{array}}{(st, \Gamma) \xrightarrow{\tau} (st[\mathcal{E}_i \mapsto l_i][\mathcal{E}_j \mapsto l_j], \Gamma')}$$

$$\textsc{Sync-2}\ \frac{i, j \in \{1, \cdots, n\}, p \in \mathcal{E}_i.P, p' \in \mathcal{E}_j.P, p' \Rightarrow p, l \in [\![\mathcal{E}_i]\!].L, st(\mathcal{E}_i) \xrightarrow{\texttt{send}\ syn(p)} l}{(st, \Gamma) \xrightarrow{\tau} (st[\mathcal{E}_i \mapsto l], join(\Gamma, p'))}$$

$$\textsc{Sync-3}\ \frac{\begin{array}{c} i \in \{1, \cdots, n\}, p \in \mathcal{E}_i.P, p' \in P, p \Rightarrow p', \Gamma' = join(\Gamma \cup \{(\{p, p'\}, \varnothing, \varnothing)\}, p) \\ l \in [\![\mathcal{E}_i]\!].L, st(\mathcal{E}_i) \xrightarrow{\texttt{send}\ syn(p)} l \end{array}}{(st, \Gamma) \xrightarrow{\texttt{send}\ syn(p')} (st[\mathcal{E}_i \mapsto l], \Gamma')}$$

$$\textsc{Sync-4}\ \frac{i \in \{1, \cdots, n\}, p \in \mathcal{E}_i.P, p' \in P, p' \Rightarrow p, l \in [\![\mathcal{E}_i]\!].L, st(\mathcal{E}_i) \xrightarrow{\texttt{send}\ syn(p)} l}{(st, \Gamma) \xrightarrow{\texttt{send}\ syn(p')} (st[\mathcal{E}_i \mapsto l], \Gamma)}$$

$$\textsc{Sync-5}\ \frac{\begin{array}{c} i \in \{1,\cdots,n\}, p \in P, p' \in \mathcal{E}_i.P, p \Rightarrow p' \\ \Gamma' = join(\Gamma \cup \{(\{p,p'\},\varnothing,\varnothing)\}, p), l \in [\![\mathcal{E}_i]\!].L, st(\mathcal{E}_i) \xrightarrow{\texttt{recv}\ syn(p')} l \end{array}}{(st,\Gamma) \xrightarrow{\texttt{recv}\ syn(p)} (st[\mathcal{E}_i \mapsto l], \Gamma')}$$

$$\textsc{Sync-6}\ \frac{i \in \{1,\cdots,n\}, p \in P, p' \in \mathcal{E}_i.P, p' \Rightarrow p}{(st,\Gamma) \xrightarrow{\texttt{recv}\ syn(p)} (st,\Gamma)}$$

There are two different types of *syn* messages. For an output port p, $syn(p)$ indicates that a write request of p is sent. In *Mediator*, each port p is uniquely connected to another port p'. So we can add the two ports p and p' to a new transaction γ and try to join γ with all existing transactions where the owner of p is involved (since p' has not responded yet). And when p' responds, $syn(p')$ is sent and the system then tries again to join all transactions where the owner of p' is involved. These rules guarantee that only when a connection is successfully established the corresponding transactions can be joint.

A system accepts *ack* messages from its sub-entities (Ack-1). When ports of the system are involved in a transaction, the system acknowledges them after all the ports of sub-entities being acknowledged (Ack-2).

$$\textsc{Ack-1}\ \frac{\begin{array}{c} i \in \{1,\cdots,n\}, p \in \mathcal{E}_i.P, (P_s,P_a,P_f) \in \Gamma, p \in P_s, l_i \in [\![\mathcal{E}_i]\!].L, \\ st(\mathcal{E}_i) \xrightarrow{\texttt{send}\ ack(p)} l_i \Gamma' = \Gamma \backslash \{(P_s,P_a,P_f)\} \cup \{(P_s\backslash\{p\}, P_a \cup \{p\}, P_f)\} \end{array}}{(st,\Gamma) \xrightarrow{\tau} (st[\mathcal{E}_i \mapsto l_i], \Gamma')}$$

$$\textsc{Ack-2}\ \frac{(P_s,P_a,\varnothing) \in \Gamma, P_s\backslash P = \varnothing, p \in P_s, \Gamma' = \Gamma\backslash\{(P_s,P_a,\varnothing)\} \cup \{(P_s\backslash\{p\}, P_a \cup \{p\}, \varnothing)\}}{(st,\Gamma) \xrightarrow{\texttt{send}\ ack(p)} (st,\Gamma')}$$

A transaction (on system level) fails iff at least one of its ports fails. In this case a $fail$ message is sent from its sub-entity where the failed port belongs (Fail-1) or the external environment (Fail-2) to the system. Once a failure is captured, the system sends $fail$ messages to all the other ports involved in this transaction (Fail-3 and Fail-4).

$$\textsc{Fail-1}\ \frac{\begin{array}{c} i \in \{1,\cdots,n\}, p \in \mathcal{E}_i.P, (P_s,P_a,P_f) \in \Gamma, p \in P_s, l_i \in [\![\mathcal{E}_i]\!].L \\ \Gamma' = \Gamma\backslash(P_s,P_a,P_f) \cup \{(P_s\backslash\{p\}, P_a, P_f \cup \{p\})\}, st(\mathcal{E}_i) \xrightarrow{\texttt{send}\ fail(p)} l_i \end{array}}{(st,\Gamma) \xrightarrow{\tau} (st[\mathcal{E}_i \mapsto l_i], \Gamma')}$$

$$\textsc{Fail-2}\ \frac{p \in P, (P_s,P_a,P_f) \in \Gamma, p \in P_s, \Gamma' = \Gamma\backslash(P_s,P_a,P_f) \cup \{(P_s\backslash\{p\}, P_a, P_f \cup \{p\})\}}{(st,\Gamma) \xrightarrow{\texttt{recv}\ fail(p)} (st,\Gamma')}$$

$$\textsc{Fail-3}\ \frac{\begin{array}{c} (\varnothing,P_a,P_f) \in \Gamma, P_f \neq \varnothing, p \in P_a, i \in \{1,\cdots,n\}, l \in [\![\mathcal{E}_i]\!].L \\ \Gamma' = \Gamma\backslash(\varnothing,P_a,P_f) \cup \{(\varnothing, P_a\backslash\{p\}, P_f \cup \{p\})\}, st(\mathcal{E}_i) \xrightarrow{\texttt{recv}\ fail(p)} l \end{array}}{(st,\Gamma) \xrightarrow{\tau} (st[\mathcal{E}_i \mapsto l], \Gamma')}$$

$$\textsc{Fail-4}\ \frac{(\varnothing,P_a,P_f) \in \Gamma, P_f \neq \varnothing, p \in P_a, \Gamma' = \Gamma\backslash(\varnothing,P_a,P_f) \cup \{(\varnothing, P_a\backslash\{p\}, P_f \cup \{p\})\}}{(st,\Gamma) \xrightarrow{\texttt{send}\ fail(p)} (st,\Gamma')}$$

A system receives commit messages $cmt(p)$ from its environment indicating that the higher-level transaction (where p is involved) has been committed (Cmt-1). If all the external ports of a transaction are committed, the system broadcast commit messages to all the internal ports (Cmt-2).

$$\textsc{Cmt-1}\ \frac{(\varnothing,P_a,\varnothing) \in \Gamma, p \in P_a, \Gamma' = \Gamma\backslash(\varnothing,P_a,\varnothing) \cup \{(\varnothing, P_a\backslash\{p\}, \varnothing)\}}{(st,\Gamma) \xrightarrow{\texttt{recv}\ cmt(p)} (st,\Gamma')}$$

$$\textsc{Cmt-2}\ \frac{\begin{array}{c} (\varnothing,P_a,\varnothing) \in \Gamma, P_a \cap P = \varnothing, p \in P_a i \in \{1,\cdots,n\}, l \in [\![\mathcal{E}_i]\!].L, st(\mathcal{E}_i) \xrightarrow{\texttt{recv}\ cmt(p)} l, \\ \Gamma' = \Gamma\backslash(\varnothing,P_a,\varnothing) \cup \{(\varnothing, P_a\backslash\{p\}, \varnothing)\} \end{array}}{(st,\Gamma) \xrightarrow{\tau} (st[\mathcal{E}_i \mapsto l], \Gamma')}$$

When a transaction is successfully committed or aborted, it will be removed from the transaction set Γ, as shown in Destroy.

$$\textsc{Destroy}\ \frac{(\varnothing,\varnothing,\varnothing) \in \Gamma}{(st,\Gamma) \xrightarrow{\tau} (st, \Gamma\backslash\{(\varnothing,\varnothing,\varnothing)\})}$$

The following rules describe the *location evolving* behavior including time elapse and interleaving internal transitions.

$$\textsc{Wait}\ \frac{t \in \mathbb{R}_{\geq 0}, \forall i \in \{1, \cdots, n\}.st(\mathcal{E}_i) \rightarrow_t st'(\mathcal{E}_i)}{(st, \Gamma) \rightarrow_t (st', \Gamma)}$$

$$\textsc{Internal}\ \frac{i \in \{1, \cdots, n\}, l \in [\![\mathcal{E}_i]\!].L, st(\mathcal{E}_i) \xrightarrow{\tau} l}{(st, \Gamma) \xrightarrow{\tau} (st[\mathcal{E}_i \mapsto l], \Gamma)}$$

6. Properties as CTL* formulae

As a super set of computation tree logic (CTL) and linear temporal logic (LTL), CTL* [6] is used as the property specification language for *Mediator* here. Both syntax and semantics of CTL* for *Mediator* are presented in the followings.

CTL* is a branching-time temporal logic where formulae of CTL* have two forms: *state formulae* and *path formulae*, denoted by $\langle \Phi \rangle$ and $\langle \phi \rangle$. Formal syntax of CTL* formulae are as follows:

$$\boxed{\begin{array}{rcl}
\langle \Phi \rangle & ::= & \langle ap \rangle \mid \texttt{!}\ \langle \Phi \rangle \mid \langle \Phi \rangle\ \texttt{\&\&}\ \langle \Phi \rangle \mid \langle \Phi \rangle\ \texttt{||}\ \langle \Phi \rangle \mid \langle \Phi \rangle\ \texttt{=>}\ \langle \Phi \rangle \mid \langle \Phi \rangle\ \texttt{<=>}\ \langle \Phi \rangle \mid \\
 & & \texttt{A}\ \langle \phi \rangle \mid \texttt{E}\ \langle \phi \rangle \\
\langle \phi \rangle & ::= & \langle \Phi \rangle \mid \texttt{!}\ \langle \phi \rangle \mid \langle \phi \rangle\ \texttt{\&\&}\ \langle \phi \rangle \mid \langle \phi \rangle\ \texttt{||}\ \langle \phi \rangle \mid \langle \phi \rangle\ \texttt{=>}\ \langle \phi \rangle \mid \langle \phi \rangle\ \texttt{<=>}\ \langle \phi \rangle \mid \\
 & & \texttt{X}\ \langle \phi \rangle \mid \texttt{F}\ \langle \phi \rangle \mid \texttt{G}\ \langle \phi \rangle \mid \langle \phi \rangle\ \texttt{U}\ \langle \phi \rangle \\
\langle ap \rangle & ::= & \langle term \rangle \mid \texttt{sync}\ \texttt{p}_1\texttt{,...,p}_n
\end{array}}$$

State formulae describe properties of *all possible paths starting from the specified* states. A state formula can be an atomic proposition denoted by $\langle ap \rangle$, a logic combination of other state formulae or a path formula decorated with branching-time temporal operator `A` or `E`.

- $\langle ap \rangle$: *Logical terms* and *synchronizing flags* are two types of supported atomic propositions. A logical term t is a normal term that is typed `bool` according to the typing rules in Section 2.2. If t is evaluated to `true` under the given state, we say the state satisfies t. A synchronizing flag is composed of a set of ports. If all ports in the synchronizing flag are synchronized in the last transition preceeded in an automaton or a system, we say the flag is satisfied.
- The *forall* operator `A`: `A` $\langle \phi \rangle$ is satisfied iff all the paths starting from the current state satisfy the path formula $\langle \phi \rangle$.
- The *exists* operator `E`: `E` $\langle \phi \rangle$ is satisfied iff at least one path starting from the current state satisfies the path formula $\langle \phi \rangle$.

Path formulae, on the other hand, capture the properties of paths where a path is an infinite sequence of states that logs how an automaton evolves. A path formula can be a state formula $\langle \Phi \rangle$, a logic combination of other path formulae or a temporal formula with linear-time temporal operators `X, G, F`, and `U`.

- $\langle \Phi \rangle$: A path satisfies a state formula $\langle \Phi \rangle$ iff the first state in the path satisfies it.
- The *next* operator `X`: `X` $\langle \phi \rangle$ means that the path formula $\langle \phi \rangle$ holds for the subsequent paths starting from the second state on the path.
- The *globally* operator `G`: `G` $\langle \phi \rangle$ means that the path formula $\langle \phi \rangle$ holds for all the subsequent paths starting from the current state.
- The *finally* operator `F`: `F` $\langle \phi \rangle$ means that the path formula $\langle \phi \rangle$ holds for some of the subsequent paths starting from the current state.
- The *until* operator `U`: $\langle \phi_1 \rangle\, \texttt{U}\, \langle \phi_2 \rangle$ holds for a path iff there exists a prefix in this path where $\langle \phi_1 \rangle$ holds for all subsequent paths starts inside the prefix and $\langle \phi_2 \rangle$ holds for the corresponding suffix.

As mentioned above, the satisfiability of CTL* formulae is declared on *states* and *paths*. The formal definition of states has already been presented in Section 3. Here we give the formal definition of paths as follows.

Definition 6.1 *(Path).* Given a LTS $L = \langle S, \Sigma, \rightarrow, s_0 \rangle$, a path that starts at $s \in S$ is an infinite sequence $\pi = s_1, s_2, \cdots$ which satisfies:

$$s = s_1 \wedge (\forall i \in \mathbb{N}^+ : s_i \rightarrow s_{i+1})$$

We denote the set of all paths starting from s by $path_L(s)$.

A *Mediator* automaton satisfies a CTL* state formula $\langle\Phi\rangle$ iff its corresponding LTS satisfies $\langle\Phi\rangle$. And whether a LTS satisfies $\langle\Phi\rangle$ depends on its initial state.

$$L = (S, \Sigma, \rightarrow, s_0) \models \langle\Phi\rangle \iff s_0 \models_L \langle\Phi\rangle$$

The notation $s \models_L \langle\Phi\rangle$ means *a state s satisfies* $\langle\Phi\rangle$ *under the LTS L* which is formally defined as follows.

$$\begin{aligned}
(ev, P) \models_L t &\iff t : \texttt{bool} \wedge ev(t) = \texttt{true} \\
(ev, P) \models_L \texttt{sync p}_1\texttt{,...,p}_n &\iff \{p_1, \cdots, p_n\} \subseteq P \\
(ev, P) \models_L \texttt{ ! } \langle\Phi\rangle &\iff (ev, P) \not\models_L \langle\Phi\rangle \\
(ev, P) \models_L \langle\Phi_1\rangle \texttt{ \&\& } \langle\Phi_2\rangle &\iff (ev, P) \models_L \langle\Phi_1\rangle \wedge (ev, P) \models_L \langle\Phi_2\rangle \\
(ev, P) \models_L \langle\Phi_1\rangle \texttt{ || } \langle\Phi_2\rangle &\iff (ev, P) \models_L \langle\Phi_1\rangle \vee (ev, P) \models_L \langle\Phi_2\rangle \\
(ev, P) \models_L \langle\Phi_1\rangle \texttt{ => } \langle\Phi_2\rangle &\iff (ev, P) \not\models_L \langle\Phi_1\rangle \vee (ev, P) \models_L \langle\Phi_2\rangle \\
(ev, P) \models_L \langle\Phi_1\rangle \texttt{ <=> } \langle\Phi_2\rangle &\iff (ev, P) \models_L \langle\Phi_1\rangle \texttt{ => } \langle\Phi_2\rangle \wedge (ev, P) \models_L \langle\Phi_2\rangle \texttt{ => } \langle\Phi_1\rangle \\
(ev, P) \models_L \texttt{ A } \langle\phi\rangle &\iff \forall \pi \in path_L((ev, P)) : \pi \models_L \langle\phi\rangle \\
(ev, P) \models_L \texttt{ E } \langle\phi\rangle &\iff \exists \pi \in path_L((ev, P)) : \pi \models_L \langle\phi\rangle
\end{aligned}$$

Similarly, for a path $\pi = s_1, s_2, \cdots$, $\pi \models_L \langle\phi\rangle$ means π *satisfies* $\langle\phi\rangle$ *under the LTS L*.

$$\begin{aligned}
\pi \models \langle\Phi\rangle &\iff s_1 \models \langle\Phi\rangle \\
\pi \models \texttt{ ! } \langle\phi\rangle &\iff \pi \not\models \langle\phi\rangle \\
\pi \models \langle\phi_1\rangle \texttt{ \&\& } \langle\phi_2\rangle &\iff \pi \models \langle\phi_1\rangle \wedge \pi \models \langle\phi_2\rangle \\
\pi \models \langle\phi_1\rangle \texttt{ || } \langle\phi_2\rangle &\iff \pi \models \langle\phi_1\rangle \vee \pi \models \langle\phi_2\rangle \\
\pi \models \langle\phi_1\rangle \texttt{ => } \langle\phi_2\rangle &\iff \pi \not\models \langle\phi_1\rangle \vee \pi \models \langle\phi_2\rangle \\
\pi \models \langle\phi_1\rangle \texttt{ <=> } \langle\phi_2\rangle &\iff \pi \models \langle\phi_1\rangle \texttt{ => } \langle\phi_2\rangle \wedge \pi \models \langle\phi_2\rangle \texttt{ => } \langle\phi_1\rangle \\
\pi \models \texttt{ X } \langle\phi\rangle &\iff s_2, s_3, \cdots \models \langle\phi\rangle \\
\pi \models \texttt{ G } \langle\phi\rangle &\iff \forall i \in \mathbb{N}^+ : s_i, s_{i+1}, \cdots \models \langle\phi\rangle \\
\pi \models \texttt{ F } \langle\phi\rangle &\iff \exists i \in \mathbb{N}^+ : s_i, s_{i+1}, \cdots \models \langle\phi\rangle \\
\pi \models \langle\phi_1\rangle \texttt{ U } \langle\phi_2\rangle &\iff \exists i \in \mathbb{N}^+ : s_{i+1}, s_{i+2}, \cdots \models \langle\phi_2\rangle \wedge (\forall j \in [1, i] : s_j, s_{j+1}, \cdots \models \langle\phi_1\rangle\,)
\end{aligned}$$

The model checking technology for CTL* on LTS is quite mature [6] and will not be discussed here.

7. Case study

In modern distributed computing frameworks (e.g. MPI [12] and ZooKeeper [16]), *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 [13] the authors proposed a classical algorithm for a typical leader election scenario, as shown in Fig. 6. 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. 6 (a)).

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 acknowledgment 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

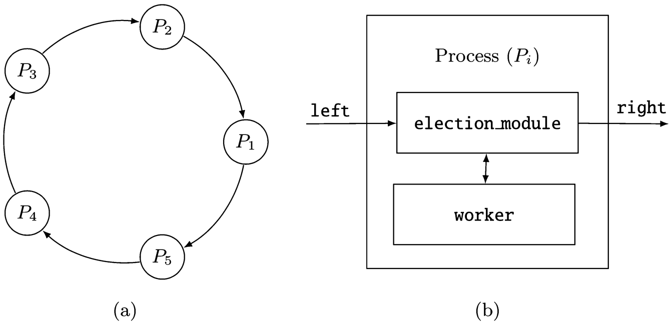


Fig. 6. (a) Topology of an asynchronous ring and (b) structure of a process.

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 7.1 *(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, root:bool> election_module ( left : in msgVote, right : out msgVote, query : out
    msgLocal
) {
  variables {
    leaderStatus : enum { pending, acknowledged } init pending;
    buffer : (voteMsg | NULL) init root ? {vtype: vote, id:id} : null;
    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, true>;
    E2 : election_module<2, true>;
    E3 : election_module<3, true>;
    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].

The most important feature of a leader election algorithm is to guarantee that the system always converges to a 'consistent' state where all election modules acknowledge the same leader. The property can be represented as follows.

```
A F (E1.leaderStatus == acknowledged && E2.leaderStatus == acknowledged &&
     E3.leaderStatus == acknowledged &&
     E1.leaderId == E2.leaderId && E2.leaderId == E3.leaderId)
```

To check this property, we rewrite a simplified version of the `election_module` and the `system` in nuXmv [10], the widely-used symbolic model checker. The automata and systems can be naturally mapped into nuXmv modules, except for a little more work on extracting the structure and union types. We provide part of the nuXmv codes here, including the declarations of election modules and the main system. The full nuXmv code including initializations and transitions can be found in [1].

```
MODULE election_module ( id,
    left_val_vtype, left_val_id, left_reqread, left_reqwrite,
    right_val_vtype, right_val_id, right_reqread, right_reqwrite
    )
    VAR
        leaderStatus : {pending, acknowledged};
        leaderId    : {0, 1, 2};
        leader_isnull : boolean;
        buffer_vtype : {vote, ack};
        buffer_id   : {0, 1, 2};
        buffer_isnull : boolean;

MODULE main ()
    VAR
        nodes_val_vtype : array 0 .. 2 of {vote, ack};
        nodes_val_id    : array 0 .. 2 of {0, 1, 2};
        nodes_reqread : array 0 .. 2 of boolean;
        nodes_reqwrite : array 0 .. 2 of boolean;

        elec0 : election_module (0,
            nodes_val_vtype[0], nodes_val_id[0],
            nodes_reqread[0], nodes_reqwrite[0],
            nodes_val_vtype[1], nodes_val_id[1],
            nodes_reqread[1], nodes_reqwrite[1]
        );

        elec1 : election_module (1,
            nodes_val_vtype[1], nodes_val_id[1],
            nodes_reqread[1], nodes_reqwrite[1],
            nodes_val_vtype[2], nodes_val_id[2],
            nodes_reqread[2], nodes_reqwrite[2]
        );

        elec2 : election_module (2,
            nodes_val_vtype[2], nodes_val_id[2],
            nodes_reqread[2], nodes_reqwrite[2],
            nodes_val_vtype[0], nodes_val_id[0],
            nodes_reqread[0], nodes_reqwrite[0]
        );

    LTLSPEC F G (
        elec0.leaderStatus = acknowledged &
        elec1.leaderStatus = acknowledged &
        elec2.leaderStatus = acknowledged &
        elec0.leaderId = elec1.leaderId &
        elec1.leaderId = elec2.leaderId
    );
```

The verification result (shown as follows) suggests that the property is satisfied under nuXmv 1.1.

```
specification F ( G ((((elec0.leaderStatus = acknowledged & elec1.leaderStatus = acknowledged) & elec2.
     leaderStatus = acknowledged) & elec0.leaderId = elec1.leaderId) & elec1.leaderId = elec2.leaderId)) is
      true
```

8. Conclusion and future work

The modeling language *Mediator* is proposed in this paper to formalize component-based concurrent and distributed system models. The language is equipped with a powerful type system including a set of typing and subtyping rules that supports static type checking. It covers most commonly-used types in popular programming languages and formal modeling languages. 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. Based on *clocks*, *messages* and corresponding newly-add statements, the language is capable for capturing real-time and asynchronous message passing behavior in distributed systems as well. Through the transaction-based semantics, we have successfully mapped the chain synchronous behavior in *Mediator* to the distributed scenario. Properties of *Mediator* models can be easily specified by CTL* and verified using model checkers like nuXmv.

Currently, two rough implementations have been provided. The first tool in [2] is based on an ANTLR grammar parsing system (part of its rules has been shown in the paper as the syntax trees) and implements system scheduling and automatic code generation to the *Arduino* [26] platform, as presented in [21]. The second tool proposed in [30] can be used to generate PRISM codes from Mediator models automatically and cooperates with the PRISM model checker to verify properties of Mediator models.

Our future work will mainly focus on:

1. Extending *Mediator* with hybrid behavior. A major target of *Mediator* is the safety-critical embedded systems, which are usually highly related to physical environments. As the behavior units are still automata, such an extension should be natural for the architecture of *Mediator*. With this extension, *Mediator* would be more powerful and able to cover lots of practical scenarios.
2. A more general-purpose code generation framework, and the corresponding verification for the code generator itself. Currently, our code generator only supports limited programming languages. A potential solution is to provide a formal semantics for the target programming languages and design general synthesis algorithms to generate target codes. Moreover, the code generator itself should be verified to make it more reliable.

Acknowledgement

This work was partially supported by the National Natural Science Foundation of China under grant no. 61772038, 61532019 and 61272160, and the Guangdong Science and Technology Department (Grant no. 2018B010107004).

References

[1] A list of Mediator models, https://github.com/liyi-david/Mediator-Proposal.

[2] The mediator implementation, https://github.com/mediator-team/mediator.

[3] P. Abdulla, J. Deneux, G. Stålmarck, H. Ågren, O. Åkerlund, Designing safe, reliable systems using SCADE, in: M. Tiziana, S. Bernhard (Eds.), Proceedings of ISoLA 2004, in: LNCS, vol. 4313, Springer, 2006, pp. 115–129.

[4] T. Amnell, G. Behrmann, J. Bengtsson, P.R. D'argenio, A. David, A. Fehnker, T. Hune, B. Jeannet, K.G. Larsen, M.O. Möller, Others: UPPAAL - now, next, and future, in: F. Cassez, C. Jard, R. Brigitte, M.D. Ryan (Eds.), Proceedings of MOVEP 2000, in: LNCS, vol. 2067, Springer, 2001, pp. 99–124.

[5] F. Arbab, Reo: a channel-based coordination model for component composition, Math. Struct. Comput. Sci. 14 (3) (2004) 329–366.

[6] C. Baier, J. Katoen, Principles of Model Checking, MIT Press, 2008.

[7] P.A. Bernstein, E. Newcomer, Principles of Transaction Processing for Systems Professionals, Morgan Kaufmann, 1996.

[8] G. Berry, G. Gonthier, The Esterel synchronous programming language: design, semantics, implementation, Sci. Comput. Program. 19 (2) (1992) 87–152.

[9] S. Bliudze, J. Sifakis, The algebra of connectors - structuring interaction in BIP, IEEE Trans. Comput. 57 (10) (2008) 1315–1330, https://doi.org/10.1109/TC.2008.26.

[10] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, S. Tonetta, The nuxmv symbolic model checker, in: A. Biere, R. Bloem (Eds.), Proceedings of CAV 2014, in: LNCS, vol. 8559, Springer, 2014, pp. 334–342.

[11] E. Curry, Message-oriented middleware, in: Q. Mahmoud (Ed.), Middleware for Communications, John Wiley & Sons, Inc., 2004, pp. 1–28.

[12] W. Gropp, E. Lusk, R. Thakur, Using MPI-2: Advanced Features of the Message-Passing Interface, MIT Press, Cambridge, MA, USA, 1999.

[13] A. Hagit, W. Jennifer, Distributed Computing: Fundamentals, Simulations, and Advanced Topics, John Wiley & Sons, Inc., 2004.

[14] B. Hahn, D.T. Valentine, SIMULINK toolbox, in: Essential MATLAB for Engineers and Scientists, Academic Press, 2016, pp. 341–356.

[15] C.A.R. Hoare, Communicating Sequential Processes, Prentice-Hall, 1985.

[16] F.P. Junqueira, B.C. Reed, M. Serafini, Zab: high-performance broadcast for primary-backup systems, in: Proceedings of DSN 2011, IEEE Compute Society, 2011, pp. 245–256.

[17] H. Kim, E.A. Lee, D. Broman, A toolkit for construction of authorization service infrastructure for the Internet of things, in: Proceedings of IoTDI 2017, ACM, 2017, pp. 147–158.

[18] M. Kwiatkowska, G. Norman, D. Parker, PRISM 4.0: verification of probabilistic real-time systems, in: G. Gopalakrishnan, S. Qadeer (Eds.), Proceedings of CAV 2011, in: LNCS, vol. 6806, Springer, 2011, pp. 1–6.

[19] E. Levy, H.F. Korth, A. Silberschatz, An optimistic commit protocol for distributed transaction management, SIGMOD Rec. 20 (2) (1991) 88–97.

[20] Y. Li, M. Sun, Component-based modeling in Mediator, in: J. Proença, M. Lumpe (Eds.), Proceedings of FACS 2017, in: LNCS, vol. 10487, Springer, 2017, pp. 1–19.

[21] Y. Li, M. Sun, Generating Arduino C codes from Mediator, in: It's All About Coordination: Essays to Celebrate the Lifelong Scientific Achievements of Farhad Arbab, in: LNCS, vol. 10865, Springer, 2018, pp. 174–188.

[22] Y. Li, M. Sun, Distributed Mediator, in: Proceedings of TASE 2019, IEEE, 2019, pp. 17–24.

[23] Z. Liu, C. Morisset, V. Stolz, rCOS: theory and tool for component-based model driven development, in: F. Arbab, M. Sirjani (Eds.), Proceedings of FSEN 2009, in: LNCS, vol. 5961, Springer, 2010, pp. 62–80.

[24] Z. Liu, C. Morisset, V. Stolz, rCOS: theory and tool for component-based model driven development, in: Proceedings of FSEN 2009, in: LNCS, vol. 5961, Springer, 2010, pp. 62–80.

[25] Q. Ma, Z. Duan, N. Zhang, X. Wang, Verification of distributed systems with the axiomatic system of MSVL, Form. Asp. Comput. 27 (1) (2014) 103–131.

[26] M. Margolis, Arduino Cookbook, O'Reilly Media, Inc., Sebastopol, USA, 2011.

[27] R. Milner, Communication and Concurrency, PHI Series in Computer Science, Prentice Hall, 1989.

[28] National Instruments: Labview, http://www.ni.com/zh-cn/shop/labview.html.

[29] M. Snir, MPI–the Complete Reference: The MPI Core, vol. 1, MIT Press, 1998.

[30] W. Sun, M. Sun, Prism code generation for verification of mediator models, in: A. Perkusich (Ed.), Proceedings of SEKE 2019, KSI Research Inc. and Knowledge Systems Institute Graduate School, 2019, pp. 271–274.

[31] J.R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M.D. Ernst, T. Anderson, Verdi: a framework for implementing and formally verifying distributed systems, ACM SIGPLAN Not. 50 (6) (2015) 357–368.

[32] L. Zou, N. Zhan, S. Wang, M. Fränzle, S. Qin, Verifying Simulink diagrams via a hybrid Hoare logic prover, in: Proceedings of EMSOFT 2013, IEEE, 2013, pp. 9:1–9:10.