# Capturing Stochastic and Real-Time Behavior in Reo Connectors

Yi Li, Xiyue Zhang, Yuanyi Ji and Meng Sun

Department of Informatics and LMAM, School of Mathematical Sciences,
Peking University
{liyi_math,zhangxiyue,jyy,sunm}@pku.edu.cn

**Abstract.** Modern distributed systems are often coupled with flexible architectures, composed of heterogenous components, and deployed on different execution nodes. Under such frameworks, connectors (or middlewares) are widely used to organize the separated components and make them functional. Apparently, reliability of such systems highly depends on the correctness of their connectors. Reo is a channel-based coordination language where complex connectors are constructed from simpler ones in a compositional approach. In this paper, we propose a stochastic and real-time extension of Reo, including a set of new primitive channels and an expressive semantics named *Stochastic Timed Automata for Reo* ($\text{STA}_r$). With the support of $\text{STA}_r$, different coordination scenarios in existing Reo extensions can be easily encoded, integrated, and analyzed.

**Keywords:** Coordination, Stochastic, Real-time, Distributed Systems

## 1 Introduction

Distributed systems have been booming everywhere in the past decades. On the one hand, Internet of Things (IoTs) are bringing network systems to daily life. Conventional devices are replaced by smart terminals, and in turn collected by central controller to construct 'Smart Cities'. On the other hand, high-performance computation is being adapted from local workstations and clusters to cloud platform and elastic computation frameworks like Amazon EC2[**?**] and Microsoft Azure[**?**]. These architectures are so popular that even small companies are starting to deploy their own private cloud systems.

In modern systems, component-based method is widely used to speed up the development process. Long-tested functional units are encapuslated as *components*, and get integrated in various systems through the *connectors*. Under this developing model, a connector often implements the core software protocol, and consequently, suffers frequently from different kind of bugs. Due to the distributed nature, it is really hard to have a bug-free connector designed manually. Unexpected scheduling and asynchronous clocks often lead to inconsistency, and on the other hand, unreliable connections and unstable delays make it even worse. A powerful framework is strongly required to formalize the connectors, and provided a basis for further formal analysis.

Reo[?], as one of the most popular coordination languages, was designed to formalize the hierachy and communication process between components. Based on channels and nodes, Reo provides a compositional approach where complex connectors are built from simpler ones. In this paper, we extend Reo with new primitive channels and the enhanced semantics $\text{STA}_r$, so that real-time and stochastic behavior can be easily handled.

Compared with existing timed and stochastic (or probabilistic) semantics of Reo [?,?,?,?], our work provides a more powerful and universal solution.

1. *Both timed behavior and stochastic behavior are supported, but declared separately.* This makes it free to model various coordination senarios by different combination patterns.
2. *Timelocks is avoided in this semantics, making the timed connectors fully implementable.* In the common semantics of timed Reo [?,?], *Timer* channels may get trapped in timelock, and in turn lead to unrealizable connectors.

The paper is organized as follows. Section 2 introduces Reo, the coordination language, and shows how we extend this language by adding new primitive channels. Then, in Section 3 we provide an adapted stochastic timed automata $\text{STA}_r$ as its formal semantics. Section 4 presents several examples. Related works and comparison are discussed in Section 5. Finally, Section 6 summarizes the paper and comes up with some future work we are going to work on.

## 2 Extending Reo for Stochastic and Timed Behavior

### 2.1 Reo

Reo is a channel-based exogenous coordination language proposed by F. Arbab in [?], where concurrency protocols are manifested as *connectors*. Basically, connectors are constructed in a recursive approach: complex ones are composed of simpler ones, where the atomic ones are called *channels*. Channels are glued on *nodes*, and they together perform the behavior of connectors.

*Nodes.* There are three types of nodes in Reo: *source nodes*, *sink nodes* and *mixed nodes*, as shown in Fig. 1.

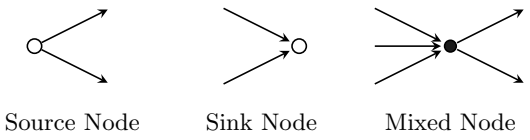



**Fig. 1.** Three Types of Nodes

Essentially, a *source node* performs *replicate* behavior. That is, any coming data values will be broadcasted sychronously if and only if all its successors are

ready to accept. A *sink node* performs *merge* behavior, accepting data values from its predecessors once at a time (this can be a non-deteministic choice if all predecessors are ready to write). And *mixed node*, literally, performs both behavior at the same time, randomly picking one input and broadcast it to all outputs.

*Channels.* As the basic functional units in Reo, channels are supposed to describe basic coordination behavior among *channel ends*. Channel ends can be either *source end* or *sink end*, indicating the direction of data flow. A set of primitive channels can be found in Fig. 2, where we use arrows to indicate the type of channel ends.

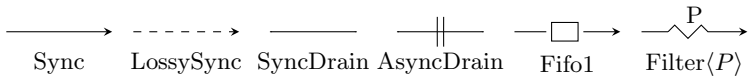



**Fig. 2.** Primitive Channels

Channels can be either *synchronous* or *asynchronous*. A channel is *synchronous* if and only if the read and write operations on its channel ends are always performed simultaneously. The behavior of some primitive channels are specified as follows.

*Sync(A:source,B:sink)* is a *synchronous* channel that deliver data values from its source end $A$ to its sink end $B$. A synchronous channel is fired only when $A$ is prepared for reading and $B$ is ready for writing.

*LossySync(A:source,B:sink)* is an *input-enabled synchronous* channel with a source end $A$ and a sink end $B$. Such channels are always prepared to accept data from $A$. However, the transmission process could be unreliable. If $B$ is also ready for writing, the received value will be sent to $B$. Otherwise the value will be dropped immediately.

*SyncDrain(A B:source)* is a *synchronous* channel with two source ends $A$ and $B$. It only accepts input from both $A$ and $B$ simultaneously and drop them together after received.

*AsyncDrain(A B:source)* is an asynchronous variation of *SyncDrain*. The most important difference is that it accepts data only from one end at a time. If both ends are ready to read, one of them (randomly picked) should wait.

*FIFO1(A:source,B:sink)* is an asynchronous channel with a source end $A$ and a sink end $B$. A FIFO1 channel can temporarily store one data value from its source end $A$ for arbitrary duration, and deliver it anytime when $B$ is ready to write. When the buffer is full, a FIFO1 cannot accept any more data values.

*Filter⟨P⟩(A:source,B:sink)* is a synchronous channel with a source end $A$, a sink end $B$ and a boolean function $P$ as its parameter. When some process (or other channels) try to write a data value to $A$, first we have to check if

the value satisfies the filter predicate $P$. If the answer is yes, the channel will behave just as *Sync*, otherwise the value will be simply dropped.

*Composition.* Formalization of nodes sometimes becomes rather complicated, as arbitrary number of incoming and outgoing edges may be involved. Usually, we tend to introduce two ternary channels *Replicator*, *Merger* and use their combinations to capture the behavior of mixed nodes.

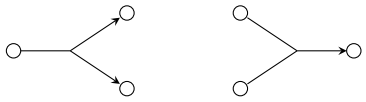


**Fig. 3.** Replicator and Merger

*Replicator(A:source,B C:sink)* is a *synchronous* broadcast channel with a source end $A$ and two sink ends $B, C$. The channel accepts data values from $A$, and broadcast them to $B, C$ iff. both $B$ and $C$ are ready to write.

*Merger(A B:source,C:sink)* is an *asynchronous* channel that collects inputs from either $A$ or $B$ and send them to $C$ simultaneously if $C$ is prepared.

*Replicator*s and *Merger*s can reduce the number of incoming and outgoing edges for mixed nodes. For example, if we replace two outgoing edges with a *Replicator* channel, the number of edges would be reduced by 1. After finite number of replacements, all the mixed nodes can be simplified as nodes with one incoming edge and one outgoing edge, which is called *flow-through*. When processing the semantics of connectors, we assume that all the mixed nodes are *flow-through* ones. However, to make it easy to understand, in the figures we still draw the mixed nodes in its original form.

### 2.2 Capturing Timed and Stochastic Behavior

In this subsection, we come up with some new primitive channels, which extend Reo and make it capable to specify timed and stochastic behavior. Compared with other formal languages, Reo provides a framework which can be easily extended by adding new channel types to the primitive channel set. Usually, new channels should be simple enough, and orthogonal to the existing ones. Following this idea, here we propose three channel types, capturing *data evolution*, *stochastic choice*, and *timed delay*.

*Map⟨f⟩(A:source,B:sink)* is a synchronous channel with a source end $A$, a sink end $B$ and a mapping function $f$ as its parameter. The function $f$ has exactly one argument. However, the argument may not always be used (for example, the function can be $f(x) = 1$). A *Map* channel behaves similar to a *Sync* channel. But it will apply $f$ to all the incoming data values and write the results to $B$.

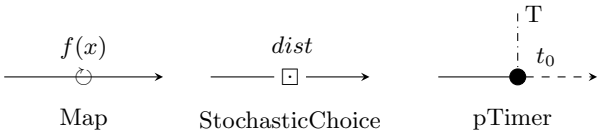



**Fig. 4.** Extended Primitive Channels

*StochasticChoice⟨dist⟩(A:source,B:sink)* is a combination of a *Sync* channel and a randomizer. It accepts data values from source node $A$ only when $B$ is writable. Then, a random value following distribution *dist* will be sent to $B$, where *dist* is provided as a parameter.

*pTimer⟨$t_0$⟩(A T:source,B:sink)* is a *parameterized* version of $t$-Timer in [**?**]. The channel accepts a data value from $A$ and throw out a timeout signal to $B$ after a certain delay. The delay is initialized by $t_0$, and can be rewritten by values coming from the source end $T$. In the beginning, a *pTimer* channel is ready to accept a numeric value from $T$ and reset its timeout as that value. However, after the channel accepts a data value from $A$ and start counting, any new value from $T$ will reset the *pTimer*, force it jumping to the initial state and forget the current counting status. If the counting process is not terminated by $T$, it will finish after the delay. Then a timeout signal will be sent to $B$ if it is writable, or simply be ignored otherwise.

## 3 Stochastic Timed Automata for Reo

In this section, first we introduce the formal model $\text{STA}_r$ that yields the basis for reasoning about timed and stochastic behavior of connectors. Then we define the new semantics for primitive Reo channels based on $\text{STA}_r$, and explain how to construct the $\text{STA}_r$ for complex connectors by applying the *product* and *hiding* operators to the $\text{STA}_r$ for simpler ones.

### 3.1 $\text{STA}_r$

Stochastic timed automata (STA) [**?**] is a powerful formalism to describe stochastic behavior and real-time behavior. Both continuous distribution and discrete distribution are supported in STA. In this paper, we slightly adapt STA as $\text{STA}_r$ so that Reo channels can be depicted more naturally and clearly. Before touching the technical details of $\text{STA}_r$, we first introduce some notations that will be used later.

During the rest of this paper, we will use $\mathbb{D}$ to denote the *data scope*, which can be any finite set, the set of real numbers $\mathbb{R}$ or their union. Namely, $\mathbb{D}$ is finite, or $\mathbb{R} \subseteq \mathbb{D}$ and $\mathbb{D} \backslash \mathbb{R}$ is finite. When the data scope is limited to finite sets, stochastic assignments are, obviously, not supported. For example, if $\mathbb{D} = \{1, 2\}$, $v := norm(e, \sigma)$ is an invalid assignment while $v' := 1 + \text{B}(1, 0.5)$ is acceptable. We use $Dist(S)$ to denote the set of continuous or discrete distributions on $S$.

**Definition 1 (Evaluations).** *Suppose $V$ is a finite set of variables, an* evaluation *on $V$ is defined as a function $ev_V : V \to \mathbb{D}$ that maps a variable identifier to a value. Similarly, we can also define clock evaluations on $C$ as $ev_C : C \to \mathbb{R}$, where $C$ is a set of clock variables. Natually, we can use $EV_V$ to denote the set of all evaluations on $V$, $EV_C$ to denote the set of all clock evaluations on $C$, and $EV$ to denote their combination, i.e.*

$$EV = \left\{ ev : V \cup C \to \mathbb{D} \cup \mathbb{R} | ev(v) = \begin{cases} ev_v(v) & v \in V \\ ev_c(v) & v \in C \end{cases}, ev_v \in EV_V, ev_c \in EV_C \right\}$$

In practice, evaluations are usually represented by a set of assignment statements. E.g., $\{a := TIMEOUT,\ b := 1,\ c := 0.5,\ \cdots\}$.

In STA$_r$, there is a very different concept named *adjoint variable*. That is, for each external action $A$, when it shows up, there must be a data value coming along, and assigned to its adjoint variable $dA$. Adjoint variables are used to describe the channels' behavior : channel ends are triggered if and only if a data value comes (or leaves).

**Definition 2 (STA$_r$).** *Stochastic Timed Automata for Reo (STA$_r$) is defined as an 8-tuple $\langle L, l_0, Acts, V, V_0, C, Inv, E \rangle$ where:*

- *$L$ is a finite set of locations,*
- *$l_0 \in L$ is an initial location,*
- *$Acts$ is a finite set of actions with the internal action $i \in Acts$ (we use $Acts_e$ to denote the set of external actions $Acts \backslash \{i\}$),*
- *$V$ is a finite set of variables that satisfies $\forall A \in Acts_e, dA \in V$,*
- *$V_0 \in EV$ is an initialized function for variables,*
- *$C$ is a finite set of clocks (we always assume that $V \cap C = \varnothing$),*
- *$Inv : L \to (EV \to Bool)$ is a function that maps locations to their corresponding invariants,*
- *$E$ is a finite set of edges. An* edge *of a STA$_r$ is defined as a 5-tuple $\langle l, acts, g, u, l' \rangle$ where*
  - *$l \in L$ is the source location,*
  - *$acts \in P(Acts_e) \cup \{\{i\}\}$ is a set of actions (external actions and internal actions do not appear on the same edge simultaneously). The transition can be fired either $acts = \{i\}$ or all external actions in $acts$ are prepared,*
  - *$g : EV \to Bool$ is the guard constraint that maps an evaluation (for both variables and clocks) to a boolean value* true *or* false*,*
  - *$u : EV \to Dist(EV)$ is a random assignment that updates the current evaluation with a random sample following a certain distribution of $Dist(EV)$,*
  - *$l' \in L$ is the target location.*

In the following, we write $l \xrightarrow{\mathbf{acts}, g, u}_E l'$ instead of $\langle l, acts, g, u, l' \rangle \in E$, or simply $l \xrightarrow{\mathbf{acts}, g, u} l'$ under ambiguity-free context. Meanwhile, in a STA$_r$ graph we use $[\mathbf{acts},\ g]u$ to label such a transition (see in Fig. 5).

### 3.2 Semantics of Primitive Channels

As mentioned before, $\text{STA}_r$ for a given Reo connector is constructed in a compositional way. In this subsection, we provide semantics of the primitive channels as $\text{STA}_r$, including both original and extended ones. The $[\![\cdot]\!]$ operator is used to denote *semantics map* which maps a Reo connector to its semantics as $\text{STA}_r$.

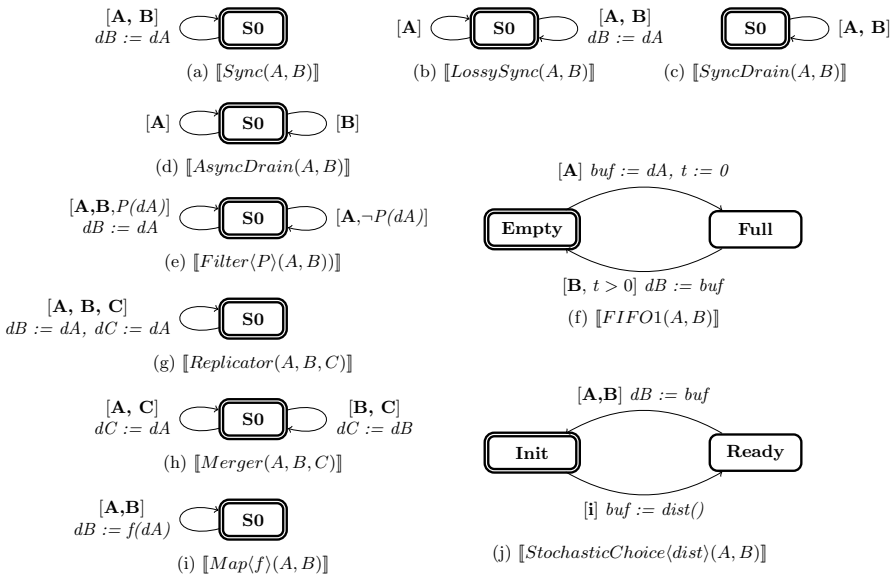



**Fig. 5.** Semantics of Primitive Channels

$[\![Sync(A, B)]\!]$ in Fig. 5(a) has only one single location and a self-loop edge indicating the read-and-write operation. No variables are involved in this automata.

$[\![LossySync(A, B)]\!]$ in Fig. 5(b) is a variation of *Sync* where data may be lost when flowing through. Here we use an extra edge to indicate this *lossy* behavior.

$[\![SyncDrain(A, B)]\!]$ in Fig. 5(c) also consists of only one location. However, unlike *Sync*, there are no assignments here since all the data items are simply dropped in this channel.

$[\![AsyncDrain(A, B)]\!]$ in Fig. 5(d) is a variation of *SyncDrain* where we can only drop one value at a time[1].

$[\![Filter\langle P\rangle(A, B)]\!]$ in Fig. 5(e) has one location and two edges. One is for the satisfaction of filter predicate and the other is for failure.

[1] Different interpretations of *AsyncDrain* have been proposed in [**?**,**?**]. For simplicity we choose the later one in [**?**], and don't consider fairness issues.

$[\![FIFO1(A,B)]\!]$ in Fig. 5(f) consists of two locations and two edges, one for reading and one for writing. Variable *buf* is used to store the value in the buffer. As mentioned earlier, data items are supposed to stay in the buffer for a positive delay, so we also need a clock $t$ even if it's not a timed channel.

$[\![Replicator(A,B,C)]\!]$ in Fig. 5(g) has only one location and one edge. The edge can be triggered if and only if a data value is obtained from $A$ and broadcast to both $B$ and $C$.

$[\![Merger(A,B,C)]\!]$ in Fig. 5(h) also has only one location, but two edges for two sink ends, respectively.

$[\![Map\langle f\rangle(A,B)]\!]$ in Fig. 5(i) is also a variation of *Sync* where the mapping function $f$ will be applied to any flow-through data item.

$[\![StochasticChoice\langle dist\rangle(A,B)]\!]$ in Fig. 5(j) consists of two locations: *init* and *ready*. As this is a synchronous channel, you may find it kind of confusing. The idea is that the random sampling should **NOT** be performed synchronously. Otherwise, you may meet the case where two edges can be triggered at the same time, but guard of the later edge relies on the random assignment of the previous one. It's hard to describe such semantics natually, so we always assume that the random process is done before triggering the assignment.

$[\![pTimer\langle t_0\rangle(A,T,B)]\!]$ in Fig. 6 consists of two locations and a large family of edges. Various border behavior is covered in this semantic model, making it capable to meet different requirements.

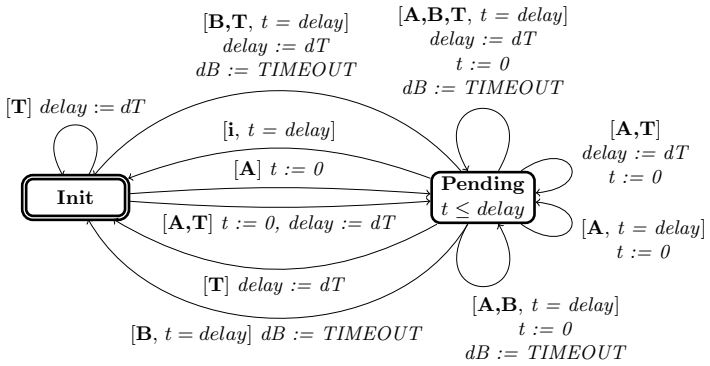



**Fig. 6.** Semantics of pTimer

### 3.3 Composition of Connectors as STA$_r$

As mentioned before, connectors in Reo are constructed from simpler ones in a compositional approach. Now we show how connectors are composed by *product* and *hiding* operations on STA$_r$.

The *product* operator is used to combine two connectors by joining their *shared nodes* (*shared actions* in STA$_r$). In *product* operations, we always assume

that shared actions have the same identifiers, while other variables and clocks are all named without repetition. Before showing the formal definition of the *product* operator, first we introduce a predicate *compatible*.

**Definition 3 (Compatible STA$_r$).** *Let $\mathscr{A}_i = \langle L_i, l_{0,i}, Acts_i, V_i, V_{0,i}, C_i, Inv_i, E_i\rangle$ be two STA$_r$ (i=1,2), they are compatible if*

- *there's no conflicting initialization on shared variables, formalized as $\forall v \in V_1 \cap V_2, V_{0,1}(v) = V_{0,2}(v)$, and*
- *shared variables can only be assigned in one of them, i.e. if $v \in V_1 \cap V_2$ and $v := expr$ (expr is an expression) appears in the assignments of $\mathscr{A}_1$, then $\forall e \in E_2$, e should not contain any assignment on v, and vice versa.*

*In other words, we don't allow two connectors to write on the same node.*

**Definition 4 (Product).** *Let $\mathscr{A}_i = \langle L_i, l_{0,i}, Acts_i, V_i, V_{0,i}, C_i, Inv_i, E_i\rangle (i = 1, 2)$ be two compatible STA$_r$, their product $\mathscr{A} = \mathscr{A}_1 \bowtie \mathscr{A}_2$ is defined as:*

$$\mathscr{A}_1 \bowtie \mathscr{A}_2 = \langle L_1 \times L_2, (l_{0,1}, l_{0,2}), Acts_1 \cup Acts_2, V_1 \cup V_2, V_0, C_1 \cup C_2, Inv, E\rangle$$

*where*

- *$V_0(v)$ is equal to $V_{0,1}(v)$ if $v \in V_1 \backslash V_2$, or $V_{0,2}(v)$ otherwise,*
- *$Inv(l_1, l_2)(ev) = Inv_1(l_1)(ev \restriction_{V_1 \cup C_1}) \wedge Inv_2(l_2)(ev \restriction_{V_2 \cup C_2})$, where $\restriction$ is used to restrict a function on certain domain,*
- *E is obtained through the following rules:*

$$\frac{l_1 \xrightarrow{\boldsymbol{acts}_1, g_1, u_1}_{E_1} l_1', acts_1 \cap Acts_2 = \varnothing}{\langle l_1, l_2\rangle \xrightarrow{\boldsymbol{acts}_1, g_1, u_1}_{E} \langle l_1', l_2\rangle} \tag{1}$$

$$\frac{l_2 \xrightarrow{\boldsymbol{acts}_2, g_2, u_2}_{E_2} l_2', acts_2 \cap Acts_1 = \varnothing}{\langle l_1, l_2\rangle \xrightarrow{\boldsymbol{acts}_2, g_2, u_2}_{E} \langle l_1, l_2'\rangle} \tag{2}$$

$$\frac{l_1 \xrightarrow{\boldsymbol{acts}_1, g_1, u_1}_{E_1} l_1', l_2 \xrightarrow{\boldsymbol{acts}_2, g_2, u_2}_{E_2} l_2', acts_1 \cap Acts_2 = acts_2 \cap Acts_1}{\langle l_1, l_2\rangle \xrightarrow{\boldsymbol{acts}_1 \cup \boldsymbol{acts}_2, g, u}_{E} \langle l_1', l_2'\rangle} \tag{3}$$

In equation 3, guard formula is the logical conjunction of $g_1$ and $g_2$, formally $g(ev) = g_1(ev \restriction_{V_1 \cup C_1}) \wedge g_2(ev \restriction_{V_2 \cup C_2})$ is defined simply following *Inv*. However, the definition of $u$ is much more complicated. For example, in Fig. 7 we may have $dB := dA$ as $u_1$, and $dC := dB$ as $u_2$. Their direct product is $dB := dA$, $dC := dB$. Obviously, we need an order here to resolve the dependency between variables (otherwise these statements could become a great mess), which is provided as follows.

1. Check all the assignment statements $v := expr$ in $u_1$, and use expression $expr$ to replace all the existence of $v$ in both $u_2$ and $g_2$,
2. Reversely, check all $v := expr$ in $u_2$, and replace their existence in both $u_1$ and $g_1$ (note that this replacement will also affect $g$),

3. Repeat the previous steps until nothing can be replaced,
4. Suppose $u'_1$ and $u'_2$ are the resolved assignment statements, we have

$$u(v) = \begin{cases} u'_1(v) & v \text{ is assigned in } v_1, \\ u'_2(v) & otherwise \end{cases}$$

With the *product* operator, we can obtain a rough combination of Reo connectors (as $\text{STA}_r$). But there are still redundant statements that should have been simplified. We now introduce the *hiding* operator which can be used to omit such unnecessary parts.

**Definition 5 (Hideable Action).** *Let $\mathscr{A} = \langle L, l_0, Acts, V, V_0, C, Inv, E\rangle$ be a $STA_r$ and $A \in Acts$ is an action. We say $A$ is hideable in $\mathscr{A}$ if a) all the assignment statements do not depend on the value of $dA$ (i.e. $dA$ never appears on the right-hand side of any assignment statement), and b) $dA$ doesn't appear in any guard or invariant.*

**Definition 6 (Hiding).** *Let $\mathscr{A} = \langle L, l_0, Acts, V, V_0, C, Inv, E\rangle$ be a $STA_r$ and $A \in Acts$ is a* hideable *action in $\mathscr{A}$. The hiding operator $\mathscr{A}\backslash\{A\}$ is defined as*

$$\mathscr{A}\backslash\{A\} = \langle L, l_0, V\backslash\{dA\}, V_0 \restriction_{V\backslash\{dA\}}, C, Acts\backslash\{A\}, Inv, E'\rangle$$

*where* $E' = \{\langle l, acts\backslash\{A\}, g, u \restriction_{V\backslash\{dA\}}, l'\rangle | \langle l, acts, g, u, l'\rangle \in E\}$.

Hiding operation can be also used to remove multiple hideable actions at a time. For example, we introduce the following notation, and it it easy to prove that this notation is well-defined, and satisfies the law of commutation (since all we do in hiding is to remove things from existing terms).

$$\mathscr{A}\backslash\{A_1, \cdots, A_n\} := \mathscr{A}\backslash\{A_1\}\backslash\{A_2\}\cdots\backslash\{A_n\}$$

Let's consider an simple example in Fig. 7, where we use *product* and *hiding* operators to combine a *Sync* and a *FIFO1*. In the figure, we shows the combined connector in different stages and its corresponding $\text{STA}_r$ step by step.

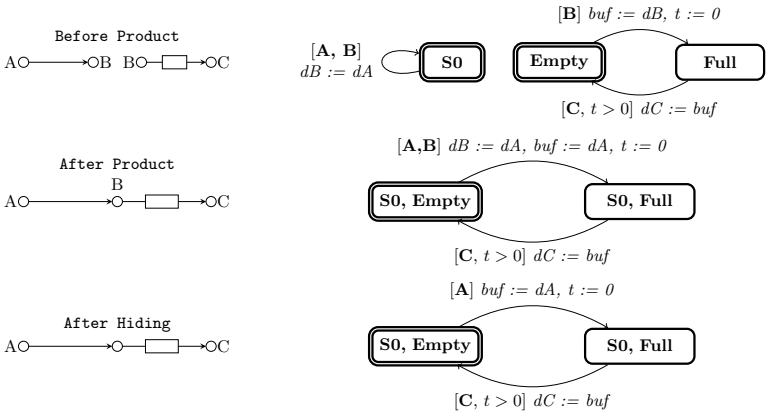



**Fig. 7.** *Product* and *Hiding* of *Sync(A,B)* and *FIFO1(B,C)*

### 3.4 Well-definedness of Composition Operators

To specify the well-definedness of composition operators listed above, here we present the *commutative* law and the *associative* law for them. Before starting, first we introduce the *isomorphism* of $\text{STA}_r$.

**Definition 7 (Isomorphism).** *Two $STA_r$ are* isomorphic *($\mathscr{A}_1 \cong \mathscr{A}_2$, where $\mathscr{A}_i = \langle L_i, l_{0,i}, Acts_i, V_i, V_{0,i}, C_i, Inv_i, E_i \rangle$), if the 1-to-1 mappings $f_L : L_1 \to L_2$, $f_V : V_1 \to V_2$, $f_C : C_1 \to C_2$, $f_{act} : Acts_1 \to Acts_2$ exist and satisfy:*

- $f_L(l_{0,1}) = l_{0,2}$,
- $\forall v \in V_1, V_{0,1}(v) = V_{0,2}(f_V(v))$,
- $\forall l \in L_1, Inv_1(l)$ *and* $Inv_2(f_L(l))$ *can be obtained from each other by variables' replacement specified by* $f_V$ *and* $f_C$,
- $\forall e = \langle l, acts, g, u, l' \rangle \in E_1$, *we can find a corresponding exclusive edge* $e' \in E_2 = \langle f_L(l), \{f_{act}(a) | a \in acts\}, g', u', f_L(l') \rangle$ *where* $g', u'$ *and* $g, u$ *can be obtained from each other by variables' replacement specified by* $f_V$ *and* $f_C$.

Informally speaking, two $\text{STA}_r$ are isomorphic if they have the same graphical structure and homologous behavior, despite the slight difference of location labels or variable identifiers. The *commutative* and *associative* laws we present in the following are essentially based on the definition of *isomorphism*.

**Theorem 1 (Commutative).** *Let $\mathscr{A}_1, \mathscr{A}_2$ be two $STA_r$, $\mathscr{A}_1 \bowtie \mathscr{A}_2 \cong \mathscr{A}_2 \bowtie \mathscr{A}_1$.*

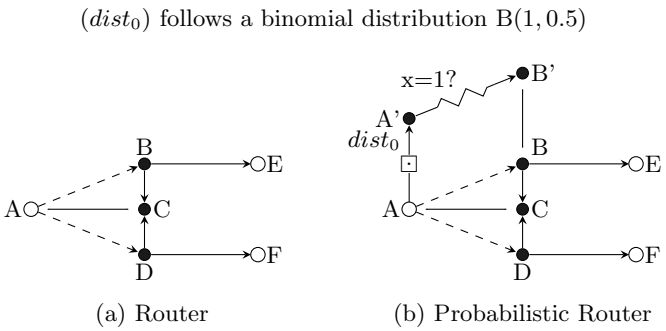
**Theorem 2 (Associative).** *Let $\mathscr{A}_1, \mathscr{A}_2, \mathscr{A}_3$ be three $STA_r$, $(\mathscr{A}_1 \bowtie \mathscr{A}_2) \bowtie \mathscr{A}_3 \cong \mathscr{A}_1 \bowtie (\mathscr{A}_2 \bowtie \mathscr{A}_3)$.*

From the two theorems above, it's clear that orders make only little difference in composition of $\text{STA}_r$. No matter how we label the identifiers and write the composing expression, finally the connectors we obtain have the same behavior. Similar to the isomorphism of graphs, these laws can be easily proved through a constructive approach.

## 4 Case Studies

With support of the composition operators, Reo can be used to capture various coordination senarios in the real world. In this section, we present two examples: *Probabilistic Router* and *Embedded Control System*.

*Example 1 (Probabilistic Router). Router* is an old, but also popular example that has shown up in several Reo papers [**?**,**?**]. As shown in Fig. 8 (a), a *Router* uses two *LossySync* channels and a *SyncDrain* channel to make sure that a coming data value is only sent to one of its sink ends. This choice is made nondeterministically at $C$, where the *Merger* channel exists. Here we show how the nondeterministic behavior is resolved as probabilistic behavior through the *StochasticChoice* channel.




(a) Router (b) Probabilistic Router

**Fig. 8.** From Router to Probabilistic Router

We attach a new path $A \to A' \to B' \to B$ to the original *Router*, including a *StochasticChoice*, a *Filter* and a *SyncDrain*, as depicted in Fig. 8 (b). When the *StochasticChoice* channel is triggered, a data value 0 or 1 will be generated, and in turn passed to the *Filter* channel. If the value is equal to 1, it will be sent to the *SyncDrain* channel $BB'$. In this case, the incoming value has to go through the path $A \to B \to E$. Otherwise, if the sampled value equals 0 and get lost in the *Filter*, the incoming value will be sent to $F$ as $B$ won't accept any values from $A$.

The corresponding $\text{STA}_r$ of a *Probabilistic Router* can be deduced on the basis of primitive channels' semantics and product operators. There are two locations in the product $\text{STA}_r$, since the primitive channels in the connector are all synchronous (including only one location) except the *StochasticChoice* channel (having two locations). According to the *product* operator, the locations should be labelled as tuples like *(S0,...,Init,...)*. Here for simplicity we use *Init* and *Ready* instead. The final result $\text{STA}_r$, after *hiding* all the internal nodes except *A,E,F*, is shown in Fig. 9. Analogous to *Replicator* and *Merger*, *Probabilistic Router* can also be regarded as a ternary channel afterwards.

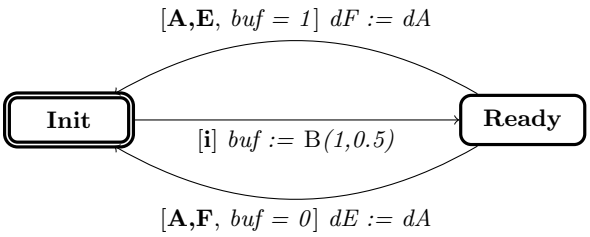



**Fig. 9.** $\text{STA}_r$ of Probabilistic Router

*Example 2 (Embedded Control System).* A common embedded control system usually comprises a set of sensors to obtain information from the environment, a set of actuators to operate on the environment, and a main processor to process information and give instructions.

In the following we show how to use Reo connectors to formalize the coordination part of an embedded control system, which is a simplified version of the embedded controller model with modular redundancy in [**?**].

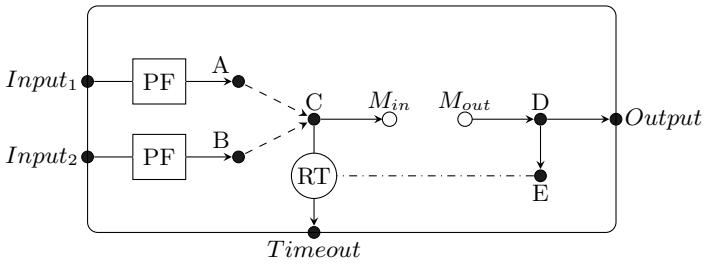



**Fig. 10.** Embedded Control System

Sensors, actuators and the main processor are regarded as components in this example. The assumption is, all the components are reliable but their communication is not. Such behavior is captured by a *Probabilistic Filter* connector **PF** which is defined in Fig. 11. The main processor, connected to ports $M_{in}$ and $M_{out}$, reads data coming from sensors, and passes instructions to the actuator through port *Input* and *Output*, respectively. The system fails to obtain an input only when both *Probabilistic Filter*s fail.

Normally, the main processor will wait for potential input. However, due to the unreliable channels, we have to set a timeout mechanism to report system failures. In this model, a complete cycle (including data acquisition, data processing and instruction transmission) should be finished within a certain duration. Otherwise a *TIMEOUT* signal will be generated by a *Reset Timer* connector **RT**.

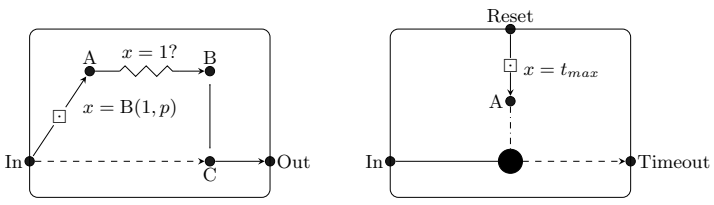



**Fig. 11.** Probablistic Filter (left) and Reset Timer (right)

A *Probabilistic Filter* drops data with a certain probability, i.e. $1-p$ in this example; while the *Reset Timer* with time bound $t_{max}$ is a timer that supports reset operation (triggered by an extra source end *Reset*). Reseting a *Reset Timer* will prevent it from generating *timeout* signals until it receives new value. The formal definition of *Probabilistic Filter* and *Reset Timer* is provided in Fig. 11. This also shows how Reo connectors are encapsulated and reused.

There are 8 locations in the $\text{STA}_r$ of the embedded system, consisting of triples which symbolizes the configuration of two probabilistic filters and the reset timer. The corresponding $\text{STA}_r$ represented in JANI format[**?**] is provided in Appendix.

## 5 Discussion

In the coordination world, Reo is well known for its variety on extensions and semantics[**?**]. And this is, of course, not the first work on its timed or stochastic semantics. Here we take *timed Reo*, *probabilistic Reo*, and *stochastic Reo* as examples, to illustrate how our new model differs from its predecessors.

*Timed Reo.* Time was natively involved in Reo from its very beginning[**?**], where *FIFO1* channel needs time constraints to ensure its retardancy. However, time was involved, only implicitly, in this semantics instead of syntax. And in some other semantics like *constraint automata*, time is even simplified as logic order. Then [**?**] proposed the raw *t-Timer* channels to capture timed delays. This work was then followed and extended in [**?**] and [**?**] with different types of timed channels.

The *pTimer* channel here in our work is, basically, enhanced from *Timer*t in [**?**]. A *Timer*t channel accepts data values and proceed timeout signals after a certain delay. However, it does not describe what happens if the timeout signal fails to deliver. In some cases, this will lead to timelock.

*Timelock* have different meaning in different semantics. Informally speaking, a timed model falls into timelock if and only if there's no possible evolution that satisfies the model constraints, and hence the model execution is forced to stop. From a practical view, a connector suffering from timelock can not be simulated or implemented. And even in theoretical senarios [**?**], it may also lead to inconsistency in proving frameworks as an unsatisfiable proposition *False* derives everything.

*Example 3 (Timelock in Timed Reo).* The *Timer*t channels may easily lead to timelock. For example, in Fig. 3, there are two *Timer*ts, one is located between $A$ and $C$, the other between $B$ and $D$. According to the original definition of *Timer*t in [**?**] and [**?**], we can derive that,

$$\forall i \in \mathbb{N}, t_i(C) = t_i(A) + 0.5, t_i(D) = t_i(B) + 1, t_i(A) = t_i(B), t_i(C) = t_i(D) \quad (*)$$

where $t(X)$ indicates the time stream on node $X$. For example, if $A$ accepts the first value at 0, and the second value at 1, then $t_0(A) = 0, t_1(A) = 1, \cdots$. From

equation(*), it's easy to derive that $t(A) = t(A) + 0.5$. This connector will be trapped in timelock once $A$ starts accepting values.

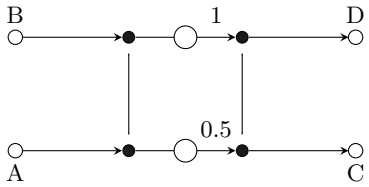



**Fig. 12.** Timelock Caused by Abuse of *Timer*t

In comparison, *pTimer* channels are timelock-free. If its sink end is not ready, the timeout signal will be dropped, and it channel will become available to accept new values again.

Further more, *pTimer* channels also support reconfiguration of delay. In an original *Timer*t channel, the delay is assigned when encoding the connector, and cannot be rewritten during execution. As a result, when formalizing specific timed connectors, we still need other channels like *reset-timer*, *expired-timer* and so on. In our framework, *pTimer* is capable to represent all these channels with only simple combination patterns.

*Probabilistic Reo.* [**?**] came up with a probabilistic extension of constraint automata, to formalize the potential lossy behavior in connectors.

In [**?**], probabilistic loss may happens while being transmitted or waiting in buffer. Definition of the former case is rather trivial, but in the latter one, discrete time is required and formalized as time units. The authors assume that in each time unit, a buffer failure may happens with probability $\tau$, and data values will get lost due to this failure.

In Section 4, we have already shown that probabilistic lossy channels can be represented by combination of *StochasticChoice* channels and *SyncDrain* channels. Actually, *pTimer* channels is also capable to produce discrete time signals. Thus, we can use probablistic lossy channels and discrete time counters to reproduce the *LossyFIFO1* in [**?**].

*Stochastic Reo.* Baier and Wolf proposed the first stochastic extension for Reo in [**?**] based on continuous-time constraint automata (CCA). The work was later extended with different semantics. For example, quantitative constraint automata in [**?**] and interactive markov chain in [**?**].

Basically, most of those stochastic semantics are based on continuous-time markov chains. Delays and arrival rates are attached to all the primitive channels, giving them randomized and unreliable behavior. This approach has a defect that random behavior is bounded with time. We can produce random delays but not random values. That's the reason why we split off stochastic delays

as *StochasticChoice*. *StochasticChoice* has nothing to do with time, but we can always combine it with *pTimer* to produce different timed connectors.

There are also other coordination models that supports stochastic and timed behavior except for Reo. For example, Probabilistic KLAIM in [**?**], Stochastic $\pi$-caculus in [**?**], etc. However, most of them provide support for modeling not only connecors, but also components. This makes it possible to model timed and stochastic behavior in the component part, and do not need extra coordination primitives. Compared with these approaches, our framework supports more complicated coordination behaviors and more intuitive modeling interfaces (graphical representation), which also keep connector designers away from potential failures.

## 6 Conclusion and Future Work

This paper comes up with an approach using Reo connectors to capture stochastic and real-time behavior in distributed systems. With an extended set of primitive channels, stochastic choices and timed delays are encapsulated as individual channels. As a theoretical framework, our approach supports partial reconfiguration (by rewritable *pTimer*) and various statistical distributions (by highly customizable *StochasticChoice*). And the case studies also prove its capacity to formalize complex coordination senarios in the real world. We use $\text{STA}_r$ as the formal semantics of stochastic and timed connectors, which is purely operational and timelock-free.

The framework, however, is still in its infancy. We need an implementation to make it compatible with popular formal tools. Currently, our plan is to encode $\text{STA}_r$ as STA so that it can be supported by JANI. JANI (JSON Automata Network Interface) [**?**] is an unified analysis framework including a shared model specification that covers STA, and a standard analyzing interface supported by various probabilistic model checking tools (Modest[**?**], IscasMC[**?**], etc.).

## Acknowledgements

The work is partially supported by ...

# Appendix

## A. Proof Sketch of Commutative and Associative Laws

*Thoerem* 2. (*Associative*) Let $\mathscr{A}_1, \mathscr{A}_2, \mathscr{A}_3$ be three STA$_r$, $(\mathscr{A}_1 \bowtie \mathscr{A}_2) \bowtie \mathscr{A}_3 \cong \mathscr{A}_1 \bowtie (\mathscr{A}_2 \bowtie \mathscr{A}_3)$.

*Proof.* We provide the sketch of this proof in a constructive style. Suppose $\mathscr{A}_i = \langle L_i, l_{0,i}, Acts_i, V_i, V_{0,i}, C_i, Inv_i, E_i \rangle$. First we calculate the two result STA$_r$, the first one $(\mathscr{A}_1 \bowtie \mathscr{A}_2) \bowtie \mathscr{A}_3$ is denoted by,

$$\begin{aligned}\langle (L_1 \times L_2) \times L_3, ((l_{0,1}, l_{0,2}), l_{0,3}), (Acts_1 \cup Acts_2) \cup Acts_3,\\ (V_1 \cup V_2) \cup V_3, V_0', (C_1 \cup C_2) \cup V_3, Inv', E' \rangle\end{aligned}$$

Similarly, the other STA$_r$ $\mathscr{A}_1 \bowtie (\mathscr{A}_2 \bowtie \mathscr{A}_3)$ is,

$$\begin{aligned}\langle L_1 \times (L_2 \times L_3), (l_{0,1}, (l_{0,2}, l_{0,3})), Acts_1 \cup (Acts_2 \cup Acts_3),\\ V_1 \cup (V_2 \cup V_3), V_0'', C_1 \cup (C_2 \cup V_3), Inv'', E'' \rangle\end{aligned}$$

As mentioned in Definition. 7, we need to construct the four *1-to-1* mapping, and prove that these mapping satisfies the certain constraint. Now let's define the mapping functions first.

- $f_L : (L_1 \times L_2) \times L_3 \to L_1 \times (L_2 \times L_3)$, and $f_L(((l_1, l_2), l_3)) = (l_1, (l_2, l_3))$. It is easy to prove that $f_L$ is a bijection,
- Since the *union* operation on sets is associative, we have $(V_1 \cup V_2) \cup V_3 = V_1 \cup (V_2 \cup V_3)$. And $f_V$ is natually defined as the identity function $\mathrm{id}_{V_1 \cup V_2 \cup V_3}$,
- Similarly, we use $\mathrm{id}_{C_1 \cup C_2 \cup C_3}$ as $f_C$, and $\mathrm{id}_{Acts_1 \cup Acts_2 \cup Acts_3}$ as $f_{act}$.

With the definition of mappings provided, we show why these functions satisfy the requirement in Definition. 7.

1. $f_L(((l_{0,1}, l_{0,2}), l_{0,3})) = (l_{0,1}, (l_{0,2}, l_{0,3}))$. This can be directly derived from its definition,
2. Suppose $\mathscr{A}_i \bowtie \mathscr{A}_j$ is denoted by $\mathscr{A}_{i \bowtie j}$. According to the definition,

   $$V_0'(v) = \begin{cases} V_{0,1}(v), v \in V_1 \backslash (V_2 \cup V_3), \\ V_{0,2 \bowtie 3}(v), v \in V_2 \cup V_3 \end{cases}, \text{ and } V_0''(v) = \begin{cases} V_{0,1 \bowtie 2}(v), v \in (V_1 \cup V_2) \backslash V_3, \\ V_{0,3}(v), otherwise \end{cases}$$

   Then we unfold $V_{0,2 \bowtie 3}$ and $V_{0,1 \bowtie 2}$, and find that the two functions are exactly the same.

   $$V_0'(v) = V_0''(v) = \begin{cases} V_{0,1}(v), v \in V_1 \backslash (V_2 \cup V_3), \\ V_{0,2}(v), v \in V_2 \backslash V_3, \\ V_{0,3}(v), otherwise \end{cases}$$

3. Variables and clock variables in the two STA$_r$ are also the same, which lead to a conclusion that variable replacements are even not required, and they share the same invariants.

4. For edges, as shown in the original definition of *product* operator, all the edges in $E'$ are added following the three rules. In a nutshell, there are two types of edges, *synchronous* or *asynchronous*. In $(\mathscr{A}_1 \bowtie \mathscr{A}_2) \bowtie \mathscr{A}_3$, an *asynchronous* edge either an edge of $\mathscr{A}_3$, or an edge of $(\mathscr{A}_1 \bowtie \mathscr{A}_2)$. And a *synchronous* edge only comes from combining an edge of $\mathscr{A}_3$ and an edge of $(\mathscr{A}_1 \bowtie \mathscr{A}_2)$. Similar to the last case, when you try to unfold it and recover its original form, you will find that $E'$ and $E''$ are almost the same, except for the different location identifiers.

The *commutative* law can be also simply proved using the similar steps.

## B. Semantics of the Embedded Control System

```
// the embedded system
{
  locations: [
    // loc_num [inv] <initial> : loc_conf
    1 <initial>: (init, init, waiting),
    2: (init, ready, waiting),
    3: (ready, init, waiting),
    4: (ready, ready, waiting),
    5 [t <= t_max] : (init, init, timing),
    6 [t <= t_max] : (init, ready, timing),
    7 [t <= t_max] : (ready, init, timing),
    8 [t <= t_max] : (ready, ready, timing)
  ],
  edges: [
    // src_loc -> dst_loc (act) [guard] <: update_func>
    1 -> 2 (i) : buf_2 = B(1, p),
    2 -> 1 (Input_2),
    1 -> 3 (i) : buf_1 = B(1, p),
    3 -> 1 (Input_1),
    1 -> 4 (i) : buf_1 = B(1, p), buf_2 = B(1, p),
    4 -> 1 (Input_1, Input_2),
    2 -> 4 (i) : buf_1 = B(1, p),
    4 -> 2 (Input_1),
    3 -> 4 (i) : buf_2 = B(1, p),
    4 -> 3 (Input_2),
    2 -> 3 (i, Input_2) : buf_1 = B(1, p),
    3 -> 2 (i, Input_1) : buf_2 = B(1, p),

    5 -> 6 (i) : buf_2 = B(1, p),
    6 -> 5 (Input_2),
    5 -> 7 (i) : buf_1 = B(1, p),
    7 -> 5 (Input_1),
    5 -> 8 (i) : buf_1 = B(1, p), buf_2 = B(1, p),
```

```
8 -> 5 (Input_1, Input_2),
6 -> 8 (i) : buf_1 = B(1, p),
8 -> 6 (Input_1),
7 -> 8 (i) : buf_2 = B(1, p),
8 -> 7 (Input_2)
6 -> 7 (i, Input_2) : buf_1 = B(1. p),
7 -> 6 (i, Input_1) : buf_2 = B(1, p),

1 -> 1 (M_out, Output) : Output = M_out,
2 -> 2 (M_out, Output) : Output = M_out,
3 -> 3 (M_out, Output) : Output = M_out,
4 -> 4 (M_out, Output) : Output = M_out,

7 -> 5 (Input_1, M_in) [buf_1 = 1, t = t_max] : t = 0,
7 -> 5 (Input_1, M_in, Timeout) [buf_1 = 1, t = t_max] :
    t = 0, M_in = Input_1, Timeout = TIMEOUT,
7 -> 5 (Input_1, M_in, M_out, Output) [buf_1 = 1] :
    t = 0, M_in = Input_1, Output = M_out,
7 -> 5 (Input_1, M_in, M_out, Output, Timeout)
    [buf_1 = 1, t = t_max] :
    t = 0, M_in = Input_1, Output = M_out, Timeout = TIMEOUT,
6 -> 5 (Input_2, M_in) [buf_2 = 1, t = t_max] : t = 0,
6 -> 5 (Input_2, M_in, Timeout) [buf_2 = 1, t = t_max] :
    t = 0, M_in = Input_2, Timeout = TIMEOUT,
6 -> 5 (Input_2, M_in, M_out, Output) [buf_2 = 1] :
    t = 0, M_in = Input_2, Output = M_out,
6 -> 5 (Input_2, M_in, M_out, Output, Timeout)
    [buf_2 = 1, t = t_max] :
    t = 0, M_in = Input_2, Output = M_out, Timeout = TIMEOUT,
8 -> 6 (Input_1, M_in) [buf_1 = 1, t = t_max] : t = 0,
8 -> 6 (Input_1, M_in, Timeout) [buf_1 = 1, t = t_max] :
    t = 0, M_in = Input_1, Timeout = TIMEOUT,
8 -> 6 (Input_1, M_in, M_out, Output) [buf_1 = 1] :
    t = 0, M_in = Input_1, Output = M_out,
8 -> 6 (Input_1, M_in, M_out, Output, Timeout)
    [buf_1 = 1, t = t_max] :
    t = 0, M_in = Input_1, Output = M_out, Timeout = TIMEOUT,
8 -> 7 (Input_2, M_in) [buf_2 = 1, t = t_max] : t = 0,
8 -> 7 (Input_2, M_in, Timeout) [buf_2 = 1, t = t_max] :
    t = 0, M_in = Input_2, Timeout = TIMEOUT,
8 -> 7 (Input_2, M_in, M_out, Output) [buf_2 = 1] :
    t = 0, M_in = Input_2, Output = M_out,
8 -> 7 (Input_2, M_in, M_out, Output, Timeout)
    [buf_2 = 1, t = t_max] :
    t = 0, M_in = Input_2, Output = M_out, Timeout = TIMEOUT,
```

```
    8 -> 5 (Input_1, Input_2, M_in) [buf_1 = 1, t = t_max] :
        t = 0, M_in = Input_1,
    8 -> 5 (Input_1, Input_2, M_in) [buf_2 = 1, t = t_max] :
        t = 0, M_in = Input_2,
    8 -> 5 (Input_1, Input_2, M_in, M_out, Output) [buf_1 = 1] :
        t = 0, M_in = Input_1, Output = M_out,
    8 -> 5 (Input_1, Input_2, M_in, M_out, Output) [buf_2 = 1] :
        t = 0, M_in = Input_2, Output = M_out,
    8 -> 5 (Input_1, Input_2, M_in, Timeout) [buf_1 = 1, t = t_max] :
        t = 0, M_in = Input_1, Timeout = TIMEOUT,
    8 -> 5 (Input_1, Input_2, M_in, Timeout) [buf_2 = 1, t = t_max] :
        t = 0, M_in = Input_2, Timeout = TIMEOUT,
    8 -> 5 (Input_1, Input_2, M_in, M_out, Output, Timeout)
        [buf_1 = 1, t = t_max] :
        t = 0, M_in = Input_1, Output = M_out, Timeout = TIMEOUT,
    8 -> 5 (Input_1, Input_2, M_in, M_out, Output, Timeout)
        [buf_2 = 1, t = t_max] :
        t = 0, M_in = Input_2, Output = M_out, Timeout = TIMEOUT,

    5 -> 1 (M_out, Output) [t < t_max] : Output = M_out,
5 -> 1 (i) [t = t_max]
    5 -> 1 (Timeout) [t = t_max] : Timeout = TIMEOUT,
    5 -> 1 (M_out, Output, Timeout) [t = t_max] :
        Output = M_out, Timeout = TIMEOUT,
    6 -> 2 (M_out, Output) [t < t_max] : Output = M_out,
6 -> 2 (i) [t = t_max]
    6 -> 2 (Timeout) [t = t_max] : Timeout = TIMEOUT,
    6 -> 2 (M_out, Output, Timeout) [t = t_max] :
        Output = M_out, Timeout = TIMEOUT,
    7 -> 3 (M_out, Output) [t < t_max] : Output = M_out,
7 -> 3 (i) [t = t_max]
    7 -> 3 (Timeout) [t = t_max] : Timeout = TIMEOUT,
    7 -> 3 (M_out, Output, Timeout) [t = t_max] :
        Output = M_out, Timeout = TIMEOUT,
    8 -> 4 (M_out, Output) [t < t_max] : Output = M_out,
8 -> 4 (i) [t = t_max]
    8 -> 4 (Timeout) [t = t_max] : Timeout = TIMEOUT,
    8 -> 4 (M_out, Output, Timeout) [t = t_max] :
        Output = M_out, Timeout = TIMEOUT,

    3 -> 5 (Input_1, M_in) [buf_1 = 1] : t = 0, M_in = Input_1,
    3 -> 5 (Input_1, M_in, M_out, Output) [buf_1 = 1] :
        t = 0, M_in = Input_1, Output = M_out,
    2 -> 5 (Input_2, M_in) [buf_2 = 1] : t = 0, M_in = Input_2,
    2 -> 5 (Input_2, M_in, M_out, Output) [buf_2 = 1] :
```

```
        t = 0, M_in = Input_2, Output = M_out,
    4 -> 6 (Input_1, M_in) [buf_1 = 1] : t = 0, M_in = Input_1,
    4 -> 6 (Input_1, M_in, M_out, Output) [buf_1 = 1] :
        t = 0, M_in = Input_1, Output = M_out,
    4 -> 7 (Input_2, M_in) [buf_2 = 1] : t = 0, M_in = Input_2,
    4 -> 7 (Input_2, M_in, M_out, Output) [buf_2 = 1] :
        t = 0, M_in = Input_2, Output = M_out,
    3 -> 6 (i, Input_1, M_in) [buf_1 = 1] :
        t = 0, M_in = Input_1, buf_2 = B(1, p),
    3 -> 6 (i, Input_1, M_in, M_out, Output) [buf_1 = 1] :
        t = 0, M_in = Input_1, buf_2 = B(1, p), Output = M_out,
    2 -> 7 (i, Input_2, M_in) [buf_2 = 1] :
        t = 0, M_in = Input_2, buf_1 = B(1, p),
    2 -> 7 (i, Input_2, M_in, M_out, Output) [buf_2 = 1] :
        t = 0, M_in = Input_2, buf_1 = B(1, p), Output = M_out
    4 -> 5 (Input_1, Input_2, M_in) [buf_1 = 1]: t = 0, M_in = Input_1
    4 -> 5 (Input_1, Input_2, M_in) [buf_2 = 1]: t = 0, M_in = Input_2
    4 -> 5 (Input_1, Input_2, M_in, M_out, Output) [buf_1 = 1]:
        t = 0, M_in = Input_1, Output = M_out,
    4 -> 5 (Input_1, Input_2, M_in, M_out, Output) [buf_2 = 1]:
        t = 0, M_in = Input_2, Output = M_out,
  ]
}
```