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 functioning. 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 through 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* (STA_r). With the support of 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, The Internet of Things (IoTs) are bringing network systems to daily life. Conventional devices are replaced by smart terminals, and in turn collected by central controllers 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 [16] and Microsoft Azure [12]. These architectures are so popular that even small companies are starting to deploy their own private cloud systems.

In modern systems, the component-based method is widely used to speed up the development process. Long-tested functional units are encapuslated as *components*, and get integrated in different systems through *connectors*. Under this developing model, a connector often implements the core software protocol, and consequently, suffers frequently from different kind of vulnerabilities.

Reo [2], as one of the most popular coordination languages, was designed to formalize the hierachy and communication pattern 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 three primitive channels, *Map*, *StochasticChoice* and *pTimer*, to capture data evolution, real-time and stochastic behavior. A new semantics named STA_r is also provided as the theoretical basis of the primitive channels.

Compared with existing timed and stochastic (or probabilistic) semantics of Reo [4, 7, 14, 15], our work provides a more powerful and universal solution.

1. *Both timed behavior and stochastic behavior are supported, but declared separately.* This makes it convenient to model various coordination scenarios by different combination patterns.
2. *Timelocks are avoided in this semantics, making the timed connectors fully implementable.* In the common semantics of timed Reo [3, 14], *Timer* channels may get trapped in *timelock*. Under such cases, the behavior of *Timer* is undefined, which makes it impossible to obtain a equivalent implementation.

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 STA_r as its formal semantics. Related work and comparison are discussed in Section 4. Section 5 presents several examples. 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 [2], where concurrency protocols are manifested as *connectors*. Basically, connectors are constructed through a compositional 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.

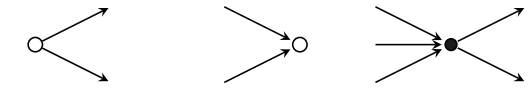


Fig. 1. Three Types of Nodes

Essentially, a *source node* performs *replicating* behavior. That is, any coming data values will be broadcasted synchronously if and only if all its successors are ready to accept. A *sink node* performs *merging* behavior, accepting data

values from its predecessors randomly (this can be a non-deterministic choice if all predecessors are ready to write). And a *mixed node*, literally, performs both behavior at the same time, randomly picking one input and broadcasting it to all outputs.

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

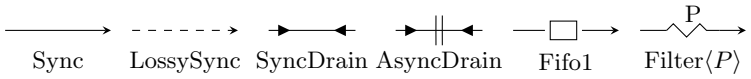


Fig. 2. Primitive Channels

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 the primitive channels shown in Fig.2 are specified as follows.

Sync(A:source,B:sink) is a *synchronous* channel that delivers 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 being 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 data comes to end A, first we have to check whether 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 broadcasts 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 sends 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 a finite number of replacements, all the mixed nodes can be simplified as nodes with one incoming edge and one outgoing edge, which are called *flow-through*. When processing the semantics of connectors, we assume that all the mixed nodes are *flow-through* ones. But in the figures we still draw the mixed nodes in their original form to make it clear and easy to understand.

2.2 Capturing Timed and Stochastic Behavior

In this subsection, we come up with some 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 *time delay*.

In the following definitions, we use $\langle p \rangle$ to denote the parameter of a channel. Value of parameters should be provided while declaring the channel, and would never be updated during the execution.

Map$\langle f \rangle$*(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. A *Map* channel accepts incoming values from its source end A (only when B is ready for writing) and then it writes $f(dA)$ to B simultaneously (dA denotes the data accepted from A).

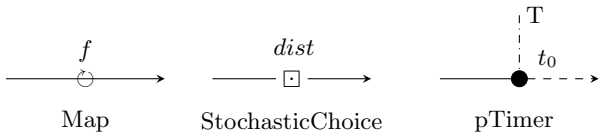


Fig. 4. Extended Primitive Channels

StochasticChoice⟨dist⟩(A:source,B:sink) is a synchronous randomizer channel that accepts data values from its source node A (only when B is ready for writing) and writes a random value to B simultaneously. The random value is sampled from the distribution parameter *dist*.

pTimer⟨t_0⟩(A T:source,B:sink) is a *parameterized* version of t-Timer in [14]. The channel accepts data values from its source end A and starts counting down. Then after a certain delay, it will send a TIMEOUT signal to B if writable, and otherwise do nothing. In both cases, the channel will reset itself and prepare to accept the next incoming value.
Value of the delay is initialized by the parameter t_0, and can be overridden by incoming values from the source end T. When the *pTimer* is not in counting down stage, an incoming value from T will simply update the delay value. Otherwise the incoming value will reset it, update the delay value, and write nothing to B. When the new delay value is provided exactly at the same time when counting down process terminates, the channel will still generate the timeout signal.

3 Stochastic Timed Automata for Reo

In this section, first we introduce the formal model 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 STA_r, and explain how to construct STA_r for complex connectors by applying the *product* and *hiding* operators to STA_r of simpler ones.

3.1 STA_r

Stochastic timed automata (STA) [8] is a powerful formalism to describe stochastic behavior and real-time behavior. Both continuous distributions and discrete distributions are supported in STA. In this paper, we slightly adapt STA as STA_r so that Reo channels can be depicted more naturally and clearly. Before touching the technical details of 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 either *a) any finite set*, *b) the set of real numbers* $\mathbb{R}$, or their union. Namely, $\mathbb{D}$ is finite, or $\mathbb{D}\backslash\mathbb{R}$ is finite (if $\mathbb{R} \subseteq \mathbb{D}$). When the data scope is restricted to finite sets, stochastic assignments are no longer supported. For example, if

$\mathbb{D} = \{1, 2\}$, $v := norm(e, \sigma)$ is an invalid assignment while $v' := 1 + \mathrm{B}(1, 0.5)$ is acceptable, where *norm* and B stand for normal and binomial distribution respectively. 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 its valuation. 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 is provided by the environment, there must be a data value coming along, and assigned to its adjoint variable dA. Adjoint variables are used to describe the channels' behavior, where the basic idea is: channel ends are triggered iff. data values come (or leave).

Definition 2 ($\mathbf{STA}_r$). *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,*
- *V is a finite set of variables that satisfies $\forall A \in Acts, dA \in V$,*
- *$V_0 \in EV$ 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,*
  - *$acts \in P(Acts)$ is a finite set of actions (internal action is denoted by the empty set),*
  - *$g : EV \to Bool$ 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 $l \xrightarrow{\mathbf{acts},g,u} l'$ if it does not lead to ambiguity. Meanwhile, in a STA_r graph we use $[\mathbf{acts},\ g]u$ to label such a transition (see in Fig. 5). For simplicity reasons, tautology guards and internal actions are omitted.

3.2 Semantics of Primitive Channels

As mentioned before, the STA_r for a given Reo connector is constructed in a compositional way. In this subsection, we provide semantics of the primitive channels as 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 STA_r.

The following figures (Fig. 5 and Fig. 6) provide a graphical representations of the primitive channels' semantics (both standard and extended ones included). In these figures, we use a set of assignments to represent an update u. For example, $dA, dB := exp_1, exp_2$ leads to a new evaluation where dA and dB are updated to exp_1 and exp_2 (both are calculated under the original evaluation).

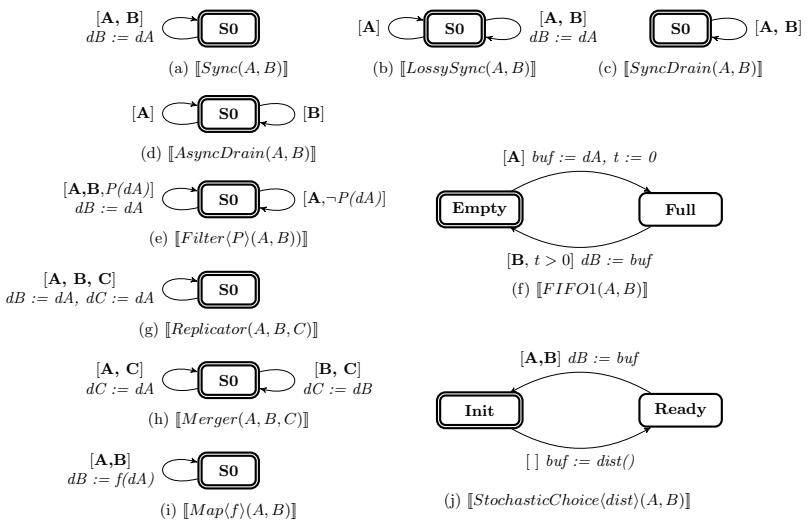


Fig. 5. Semantics of Primitive Channels

$[\![Sync(A, B)]\!]$ 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 which is only triggered when B is not writable.

$[\![SyncDrain(A, B)]\!]$ 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 an asynchronous variation of *SyncDrain* where we can only drop from one end 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.

$[\![FIFO1(A, B)]\!]$ 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 a synchronous channel 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 simultaneously. 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.

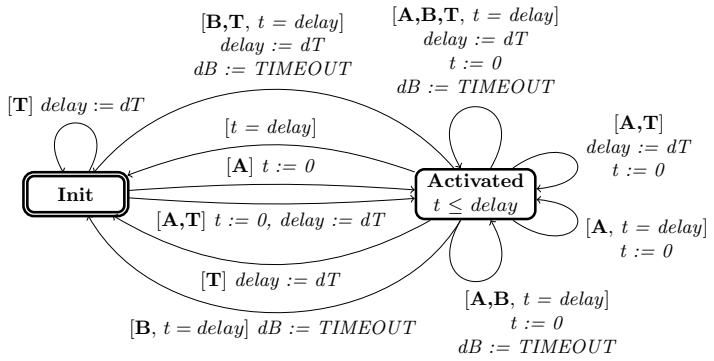


Fig. 6. Semantics of pTimer

[1] Different interpretations of *AsyncDrain* have been proposed in [2, 6]. For simplicity we choose the later one in [6], and don't consider fairness issues.

When the channel is in *init* state, it is able to accept values from T and update its delay time, or accept values from A, jump to the *activated* state and start counting down process. In the *activated* state, the channel may accept values from A, B and T:

- *When counting down process is not finished yet,* only T is writable and any incoming values from T will reset the timer to *init* state.
- *When counting down finishes,* all combinations in $P(A, B, T)$ is acceptable. If B is writable, a TIMEOUT signal will be sent to B. If A has an incoming value, it will trigger a new counting down process immediately. And if T has an incoming value, the delay time will be overridden.

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 rule (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 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).

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

We consider a simple example in Fig. 7, where we use *product* and *hiding* operators to combine a *Sync* and a *FIFO1* channel. In Fig. 7, we show the combined connector in different stages and its corresponding STA_r step by step.

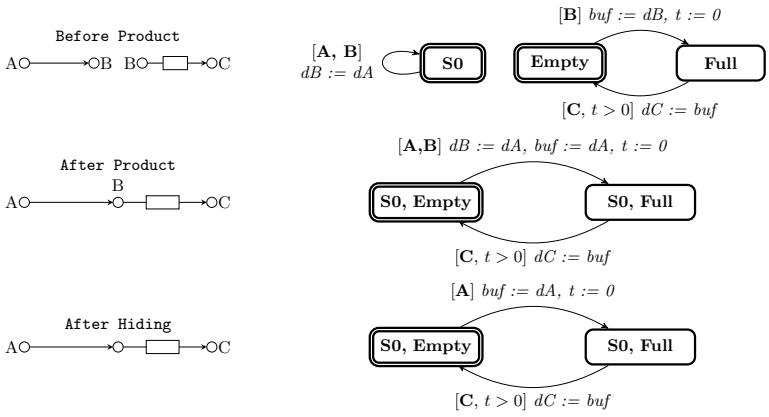


Fig. 7. *Product* and *Hiding* of *Sync(A,B)* and *FIFO1(B,C)*

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 STA_r.

Definition 7 (Isomorphism). *Two STA_r are* isomorphic *($\mathscr{A}_1 \cong \mathscr{A}_2$, where $\mathscr{A}_i = \langle L_i, l_{0,i}, Acts, V_i, V_{0,i}, C_i, Inv_i, E_i \rangle$), if the 1-to-1 mappings $f_L : L_1 \to L_2$, $f_V : V_1 \to V_2$ and $f_C : C_1 \to C_2$ 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* $e' \in E_2 = \langle f_L(l), acts, g', u', f_L(l') \rangle$ *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 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[2]). *Let $\mathscr{A}_1, \mathscr{A}_2$ be two STA_r, $\mathscr{A}_1 \bowtie \mathscr{A}_2 \cong \mathscr{A}_2 \bowtie \mathscr{A}_1$.*

[2] Proof can be found at https://github.com/liyi-david/ReoSTA.

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 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 Discussion

In the coordination community, Reo is well known for its variety on extensions and semantics [11]. This paper is not the first work on its timed or stochastic extension. Here we take *timed Reo*, *probabilistic Reo*, and *stochastic Reo* as examples to illustrate how our model differs from its predecessors.

Timed Reo. Time was natively involved in Reo from its very beginning [2], where *FIFO1* channel needs time constraints to ensure its retardancy. However, time was involved only implicitly in [2] instead of syntax. And in some other semantic models like *constraint automata*, time is even simplified as temporal order. A set of raw *t-Timer* channels was proposed in [3] to capture timed delays. This work was then followed and extended in [15] and [14] with different types of timed models.

The *pTimer* channel proposed in this paper is basically an improvement of *t*-Timer in [14]. A *t*-Timer channel accepts data values and produce timeout signals after a certain delay t. However, it does not describe what happens if the timeout signal fails to deliver. In some cases, this may lead to timelock.

Timelock has different meaning in different semantic models. Informally speaking, a timed model falls into timelock if and only if there is no possible evolution that satisfies the model constraints, and hence the model execution is forced to stop. From a practical perspective, a connector suffering from timelock can not be simulated or implemented. And even in theoretical scenarios [13], it may also lead to inconsistency in proving frameworks as an unsatisfiable proposition *False* derives everything.

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 data items again. Furthermore, *pTimer* channels also support reconfiguration of delays, which make it able to encode other timer channels (such as *EXPTimer, OFFTimer and RSTTimer* in [14]) through simple combination patterns (refer to Example 2, Section 5).

Probabilistic Reo. a probabilistic extension of constraint automata was proposed in [5] to formalize the potential lossy behavior in connectors. In [5], probabilistic loss of data may happen while the data is being transmitted or waiting in the buffer. Definition of the former case is rather trivial, but in the latter one, the

discrete time model is required. The authors assume that in each time unit, a buffer failure may happen with a probability τ, and data items may get lost due to this failure.

Probabilistic lossy behavior of connectors can be represented by combination of *StochasticChoice* channels and *SyncDrain* channels (an example of which can be found in Section 5). Actually, *pTimer* channels can also produce discrete time signals. Thus, we can use probabilistic lossy channels and discrete time counters to reproduce probabilistic connectors like the *LossyFIFO1* in [5].

Stochastic Reo. Baier and Wolf proposed the first stochastic extension for Reo in [7] based on Continuous-time Constraint Automata (CCA). The work was later extended with different semantics. For example, Quantitative Intentional Automata in [4] and interactive Markov Chain in [17].

Basically, most of those stochastic semantics are based on continuous-time Markov Chains. Delays and arrival rates are attached to primitive channels, giving them randomized or unreliable behavior. This approach has a defect that random behavior is bounded with time. We can produce random delays but not random values. That is 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 with stochastic behavior.

There are also other coordination models that supports stochastic and timed behavior except for Reo. For example, Probabilistic KLAIM in [18], Stochastic π-caculus in [19], etc. However, in most of them timed and stochastic behavior are supported in components, instead of coordinators. 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.

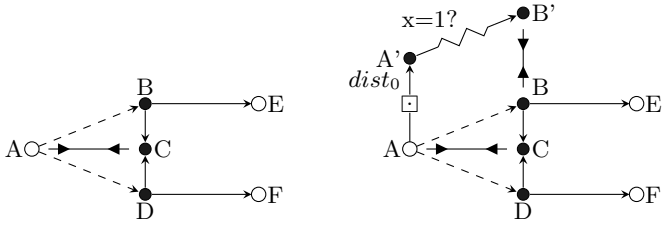
5 Case Studies

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

Example 1 (Probabilistic Router). Router is a widely used connector example [6, 3]. 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.

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, numeric value 0 or 1 will be generated, and in turn passed to the *Filter* channel. If the value is 1, it will be sent to the

($dist_0$) follows a binomial distribution B(1, 0.5)



(a) Router (b) Probabilistic Router

Fig. 8. From Router to Probabilistic Router

SyncDrain channel BB'. In this case, the incoming value has to go through the path $A \rightarrow B \rightarrow E$. Otherwise, if the sampled value is 0, it will be dropped by the *Filter*, the incoming value will be sent to F as B cannot accept any data from A.

The corresponding 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 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 STA_r, after *hiding* all the internal nodes except *A,E,F*, is shown in Fig. 9.

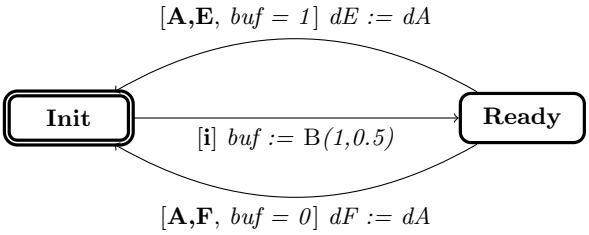


Fig. 9. STA_r of Probabilistic Router

Example 2 (Expiring Timer).

In Timed Reo [14], different types of timer channels are proposed to capture real-time behaviors in different practical scenarios, including: *OffTimer* that allows the timer to terminate the counting process when a certain signal is received, *RSTTimer* that allows the timer to reset and restart its counting process, and

EXPTimer that makes the timer produce the TIMEOUT signal immediately when a certain signal is received.

In this paper we take *EXPTimer* as an example to show how *pTimer* is used to encode the previous timers. In this example, the default delay time is denoted by t. (See Fig. 10)

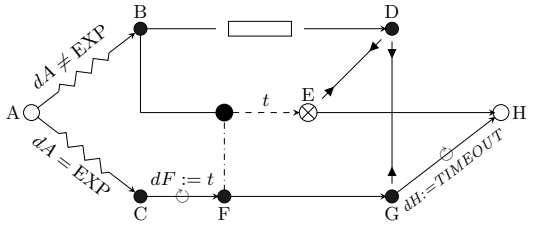


Fig. 10. Encode *EXPTimer* with *pTimer*

Basically, this connector divide the incoming values into two classes: expiring signals and normal values. For a normal value, it goes into the *pTimer* channel, and its copy is temporarily stored in the *FIFO1* channel to show that the *pTimer* is activated. When the counting process finished successfully without interruption, the buffered value will be dropped due to the *SyncDrain* channel, and a TIMEOUT signal will be sent to H.

On the other hand, when an expiring signal is caught, it will be replaced by the default delay value t and sent to the T end of the *pTimer*. According to the semantics, the *pTimer* will be reset immediately, and the buffered value will be dropped while sending out the TIMEOUT signal.

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. Theoretically, our approach supports partial reconfiguration (by rewritable *pTimer*) and various stochastic distributions (by highly customizable *StochasticChoice*). The case studies illustrate its capacity to formalize complex coordination scenarios in the real world. We use 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 existing popular tools for formal modeling and verification. Currently, our plan is to encode STA_r in JANI (JSON Automata Network Interface) [1], which is a unified analysis framework including a shared model specification that covers STA, and a standard analyzing interface supported by various probabilistic model checking tools (Modest[10], IscasMC[9], etc.).

Acknowledgements

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

References

1. The JANI specification of the jani-model format and the jani-interaction protocol, http://www.jani-spec.org/
2. Arbab, F.: Reo: a channel-based coordination model for component composition. Mathematical Structures in Computer Science 14(3), 329–366 (2004)
3. Arbab, F., Baier, C., de Boer, F.S., Rutten, J.J.M.M.: Models and temporal logical specifications for timed component connectors. Software and System Modeling 6(1), 59–82 (2007)
4. Arbab, F., Chothia, T., van der Mei, R., Meng, S., Moon, Y., Verhoef, C.: From coordination to stochastic models of QoS. In: COORDINATION 2009. LNCS, vol. 5521, pp. 268–287. Springer (2009)
5. Baier, C.: Probabilistic models for Reo connector circuits. Journal of Universal Computer Science 11(10), 1718–1748 (2005)
6. Baier, C., Sirjani, M., Arbab, F., Rutten, J.: Modeling component connectors in Reo by constraint automata. Science of Computer Programming 61(2), 75–113 (2006)
7. Baier, C., Wolf, V.: Stochastic reasoning about channel-based component connectors. In: Proceedings of COORDINATION 2006, LNCS, vol. 4038, pp. 1–15. Springer (2006)
8. Hahn, E.M., Hartmanns, A., Hermanns, H.: Reachability and reward checking for stochastic timed automata. Electronic Communication of the European Association of Software Science and Technology 70 (2014)
9. Hahn, E.M., Li, Y., Schewe, S., Turrini, A., Zhang, L.: IscasMC: A web-based probabilistic model checker. In: Proceedings of FM 2014. LNCS, vol. 8442, pp. 312–317. Springer (2014)
10. Hartmanns, A.: Modest a unified language for quantitative models. In: Proceedings of FDL'12. pp. 44–51. IEEE (2012)
11. Jongmans, S.S.T.Q., Arbab, F.: Overview of thirty semantic formalisms for Reo. Scientific Annals of Computer Science 22(1), 201–251 (2012)
12. Li, H.: Introducing Windows Azure. Apress, Berkely, CA, USA (2009)
13. Li, Y., Sun, M.: Modeling and verification of component connectors in Coq. Science of Computer Programming 113, 285–301 (2015)
14. Meng, S.: Connectors as designs: The time dimension. In: Proceedings of TASE 2012. pp. 201–208. IEEE Computer Society (2012)
15. Meng, S., Arbab, F.: On resource-sensitive timed component connectors. In: Proceedings of FMOODS 2007. LNCS, vol. 4468, pp. 301–316. Springer (2007)
16. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How amazon web services uses formal methods. Communications of the ACM 58(4), 66–73 (2015)
17. Oliveira, N., Barbosa, L.S.: An enhanced model for stochastic coordination. Electronic Proceedings in Theoretical Computer Science 228, 35–45 (2016)
18. Pierro, A.D., Hankin, C., Wiklicky, H.: Probabilistic KLAIM. In: de Nicola, R., et al. (eds.) COORDINATION 2004. LNCS, vol. 2949, p. 119134. Springer (2004)
19. Priami, C.: Stochastic pi-calculus. The Computer Journal 38(7), 578–589 (1995)

Appendix

A. Proof Sketch of the Associative Law

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 $\mathscr{A}_i = \langle L_i, l_{0,i}, Acts, V_i, V_{0,i}, C_i, Inv_i, E_i \rangle$. 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,

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

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

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

As mentioned in Definition. 7, we need to construct the four *1-to-1* mapping functions, and prove that these mapping functions satisfy certain constraints. Now we 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 naturally defined as the identity function $\mathrm{id}_{V_1 \cup V_2 \cup V_3}$,

With the definition of these mapping functions provided, we show why these functions satisfy the requirements 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 is 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 we try to unfold it and recover its original form, we can 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.