Synthesis of Reo circuits from scenario-based interaction specifications

Sun Meng a,⁎, Farhad Arbab a, Christel Baier b

a CWI, Amsterdam, The Netherlands
b Institute for Theoretical Computer Science, Technische Universität Dresden, Dresden, Germany

A R T I C L E   I N F O

Article history:
Received 29 January 2009
Received in revised form 19 February 2010
Accepted 2 March 2010
Available online 20 March 2010

Keywords:
Connector
Reo circuits
Scenario-based specification
UML
Synthesis

A B S T R A C T

It is difficult to construct correct models for distributed large-scale service-oriented applications. Typically, the behavior of such an application emerges from the interaction and collaboration of multiple components/services. On the other hand, each component, in general, takes part in multiple scenarios. Consequently, not only components, but also their interaction protocols are important in the development process for distributed systems. Coordination models and languages, like Reo, offer powerful “glue-code” to encode interaction protocols. In this paper we propose a novel synthesis technique, which can be used to generate Reo circuits directly from scenario specifications. Inspired by the way UML2.0 sequence diagrams can be algebraically composed, we define an algebraic framework for merging connectors generated from partial specifications by exploiting the algebraic structure of UML sequence diagrams.

© 2010 Elsevier B.V. All rights reserved.

doi:10.1016/j.scico.2010.03.002

1. Introduction

Service-oriented applications consisting of services that may run on large-scale distributed platforms are notoriously difficult to construct. It is well-known that most service-oriented applications rely on a collaborative behavior among their constituent services/components, and this implies complex coordination. Therefore, construction of these applications crucially depends on deriving a correct coordination model that specifies the precise order and causality of the actions of their constituent services. For example, in an online banking scenario, a user can log into the system only after the account information such as the account number and password are verified to be valid. Given the strong role that coordination of components/services plays in such applications, important questions from the software engineering perspective include:

• What are the connectors in an application that coordinate the behavior of its components/services?
• What does a service oriented development process look like?
• How can one systematically generate connectors from interaction specifications?

In this paper we address these questions by using Reo as the coordination language in service oriented applications, and show how correct Reo circuits (connectors) can be synthesized automatically from scenario-based interaction specifications.

Scenarios represent a global view of interactions among the components (in the broadest sense) within a system. Each scenario corresponds to a single temporal sequence of interactions among system components/services and provides a partial system description. Scenarios are close to users’ understanding and they are often employed to refine use cases and provide an abstract view of the system behavior. In recent years, scenario based languages such as UML Sequence Diagrams (SDs) [30], message sequence charts (MSCs) [17,18], and Live Sequence Charts (LSCs) [13], have become popular.

⁎ Corresponding author.
E-mail addresses: M.Sun@cwi.nl (S. Meng), Farhad.Arbab@cwi.nl (F. Arbab), baier@tcs.inf.tu-dresden.de (C. Baier).
for expressing behavioral requirements of applications. In this paper we focus on scenarios represented as UML sequence diagrams. However, our synthesis approach can be easily generalized to use other alternatives, such as HMSC [28].

The idea of using scenario descriptions, such as UML SDs, to generate operational models and/or executable code, of course, is not new [15,16,25,32]. We briefly describe some related work in this area in Section 6. However, most of the existing work takes an endogenous approach for coordination.

As an example, consider a use case scenario in a simple bank ATM, that involves a user, an ATM, and a number of remote processes, including a PIN verifier. The scenario describes that after feeding his card into the ATM, (1) the ATM asks the user to enter his PIN; (2) the ATM sends the user ID and his PIN to the PIN verifier; (3) the PIN verifier verifies the PIN to determine the validity of the access request; (4) the PIN verifier sends its (Allow/Deny/Confiscate) response back to the ATM; and (5) depending on the content of the response, the ATM proceeds to either allow or deny user access, or confiscates his card, presumably, after 3 unsuccessful attempts entering a wrong PIN. The common view of the transformation of this scenario to executable code yields an ATM process and a PIN verifier process that directly communicate with each other: the ATM process contains a "send ⟨ID, PIN⟩ to PINverifier" instruction somewhere in its code, implementing step 2; and the PIN verifier process contains a "send response to ATM" instruction somewhere in its code, implementing step 4, above. These direct communication instructions implement the coordination protocol described in the scenario in an endogenous form.

Endogenous models implement/express a protocol only implicitly, through fragments of code in disparate entities that are hardwired to specifically realize that protocol. Suppose now that in a later version of this system, it is decided that the messages sent to the PIN verifier process must also be sent to some monitoring process, or instead of the PIN verifier to another more sophisticated process (e.g., one that tells the ATM to confiscate the user's card if the number of successive wrong PIN access attempts by the same card ID through all involved ATMs within, say, a 24 h sliding window, exceeds a threshold). Such changes to the protocol can easily be reflected in the SD specifications. However, implementing them requires invasive changes to a variety of independent software units that comprise the participating processes; worse, these changes may necessitate other less obvious changes that affect other software units and processes that are not directly involved in the modified portion of the protocol. Thus, small, “local” changes to a protocol can propagate through large spans of software units, touching them in ways that may invalidate their previously verified properties. Not only are such invasive modifications generally undesirable, in many cases they are impractical or even impossible, e.g., when they involve legacy code or third party providers.

Alternatively, the exogenous view of a scenario (e.g., the EnterPwd SD in the top right corner of Fig. 4) imposes a purely local interpretation on each inter-process communication, implementing it as a pure I/O operation on each side, that allows processes to communicate anonymously, through the exchange of untargeted passive data. For instance, Fig. 1 shows the behavior skeleton of the four processes involved in the EnterPwd SD, mentioned above. Observe that these processes are not hardwired to directly communicate with each other. Replacing exchanges of targeted messages with simple I/O localizes the range of their impact. This makes processes engaged in a protocol oblivious to changes in the protocol and their peers that do not directly impact their own behavior. Having expunged all communication/coordination concerns out of the parties involved, exogenous models relegate the task of conducting the required coordination to a (centralized or distributed) coordinator glue code that establishes the necessary communication links among the parties and engages them in the specified protocol. Reo is a good example of an exogenous coordination language that can be used to develop such glue code.

The scheme that we advocate in this paper for generating coordination models from UML sequence diagrams uses the exogenous view in its interpretation of these scenario specifications. To our knowledge, this approach is novel and no other work has considered scenario specifications for exogenous coordination. Our approach is structural and embodies the advantages inherent in the exogenous models of coordination: coordinated processes are strictly isolated from the dependencies on their environment and the details of the protocol that do not involve their individual behavior. This leads to more reusable processes, and an independent, explicit coordination protocol that can in turn be reused to coordinate these or similar processes.

In [4] the problem of synthesizing Reo circuits from given automata specifications is discussed. In [7] we provide an approach for synthesizing constraint automata from scenario specifications represented as UML sequence diagrams. However, taking constraint automata as the bridge between scenarios and connectors, may introduce superfluous serialization in the synthesis process and result in unnecessarily complex Reo circuits, even for simple scenarios. Our work
in the remainder of this paper goes one step further toward bridging the gap between low-level implementations and
scenario-based specifications, by generating Reo circuits directly from UML SDs. This approach can reduce the redundancy
in our previous work and make the resulting Reo circuits more compact and efficient. Furthermore, the proposed translation
from UML SDs to Reo circuits in this paper is structural and therefore preserves the nature of the interaction specification
inherent in a UML SD. This is an advantage over the translation from constraint automata to Reo circuits which cannot
recover parallelism. In fact, in constraint automata, parallelism is lost and impossible to retract.

There is plenty of research on providing semantics for UML 2.0 sequence diagrams [14,26,33]. Here we use the coalgebraic
approach in [33] for defining the semantics of basic sequence diagrams and formally define the operators for combining
sequence diagrams. As pointed out in [33], the coalgebraic approach is compositional, and leads to the coinductive proof
style which provides an elegant way to check the bisimulation and refinement relations between models. On the other
hand, the coalgebraic semantics of Reo is investigated in [6]. However, in [6], the semantics of a Reo circuit is given not
as a single coalgebra, but as a relation on the timed data streams (final coalgebras) on different nodes. This semantics
precisely specifies the initial intuition behind Reo connectors. The declarative, relational nature of this semantics is one
of its strengths; nevertheless, it also makes it difficult to operationalize and build formal proofs of bisimulation/simulation
relations. Fortunately, the operational semantics of Reo has been investigated in [9] by using an automata-based formalism,
called constraint automata. In a constraint automaton, states represent Reo configurations (which are determined by the
contents of the buffers) and transitions encode maximally-parallel stepwise evolution. Transition labels show maximal sets
of active nodes and sets of data constraints. It is well known that state-based systems such as automata and transition
systems, can be described by coalgebras [19,31]. Therefore, in this paper we adopt such a coalgebraic interpretation of Reo
circuits, where the state space of a coalgebra has the same meaning as in its constraint automata semantics, which turns the
coalgebra into an abstraction of its corresponding constraint automaton. One benefit of using this coalgebraic interpretation
is that the correctness of the mapping from UML to Reo in our synthesis approach can, in principle, be formally judged by
comparing their semantics.

The remainder of this paper is organized as follows: Section 2 contains a brief summary of Reo. In Section 3 we present the
relevant features of UML Sequence Diagrams. We explain the construction of Reo circuits from given scenario specifications
represented by UML Sequence Diagrams in Section 4. In Section 5 we prove the correctness of our synthesis approach by
providing a bisimulation between the coalgebras for UML Sequence Diagrams and the synthesized Reo circuits. In Section 6,
we present related work and compare it with our approach. Finally, Section 7 concludes the paper.

2. Reo

Reo [2] is a channel-based exogenous coordination model wherein complex coordinators, called connectors, are
compositionally constructed from simpler ones. We summarize only the main concepts in Reo here. Further details about
Reo and its semantics can be found in [2,6,9].

Complex connectors in Reo consist of a network of primitive connectors, called channels. A connector provides the
protocol that controls and organizes the communication, synchronization and cooperation among the components/services
that it interconnects. Each channel has two channel ends. There are two types of channel ends: source and sink. A source
channel end accepts data into its channel, and a sink channel end dispenses data out of its channel. It is possible for the
ends of a channel to be both sinks or both sources. Reo places no restriction on the behavior of a channel and thus allows
an open-ended set of different channel types to be used simultaneously together. Each channel end can be connected to at
most one component instance at any given time. Fig. 2 shows the graphical representation of some simple channel types in
Reo. A FIFO1 channel represents an asynchronous channel with one buffer cell which is empty if no data item is shown in
the box (this is the case in Fig. 2). If a data element $d$ is contained in the buffer of a FIFO1 channel then $d$ is shown inside
the box in its graphical representation. A synchronous channel has a source and a sink end and no buffer. It accepts a data
item through its source end iff it can simultaneously dispense it through its sink. A lossy synchronous channel is similar to
a synchronous channel except that it always accepts all data items through its source end. The data item is transferred if
it is possible for the data item to be dispensed through the sink end, otherwise the data item is lost. For a filter channel, its
pattern $P \subseteq Data$ specifies the type of data items that can be transmitted through the channel. Any value $d \in P$ is accepted
through its source end iff its sink end can simultaneously dispense $d$; all data items $d \notin P$ are always accepted through the
source end, but are immediately lost. The $P$-producer is a variant of a synchronous channel whose source end accepts any
data item, but the value dispensed through its sink end is always a data element $d \in P$.

There are some more exotic channels permitted in Reo: (A)synchronous drains have two source ends and no sink end. No
data value can be obtained from drains since they have no sink end. A synchronous drain can accept a data item through
one of its ends iff a data item is also available for it to simultaneously accept through its other end as well, and all data

\begin{center}
\begin{figure}
\centering
\includegraphics[width=\textwidth]{fig2.png}
\caption{Some basic channels in Reo.}
\end{figure}
\end{center}
Example 1 (Sequencer). Fig. 3(a) shows an implementation of a sequencer by composing five synchronous channels and four FIFO1 channels together. The first (leftmost) FIFO1 channel is initialized to have a data item in its buffer, as indicated by the presence of the symbol e in the box representing its buffer cell. The actual value of the data item is irrelevant. The connector provides only the four nodes A, B, C and D for other entities (connectors or component instances) to take from. The take operation on nodes A, B, C and D can succeed only in the strict left-to-right order. This connector implements a generic sequencing protocol: we can parameterize this connector to have as many nodes as we want simply by inserting more (or fewer) Sync and FIFO1 channel pairs, as required.

Fig. 3(b) shows a simple example of the utility of the sequencer. The connector in this figure consists of a two-node sequencer, plus a pair of Sync channels and a SyncDrain channel connecting each of the nodes of the sequencer to the nodes A and C, and B and C, respectively. The behavior of the connector can be seen as imposing an order on the flow of the data items written to A and B, through C: the sequence of data items obtained by successive take operations on C consists of the first data item written to B, followed by the first data item written to A, followed by the second data item written to B, followed by the second data item written to A, and so on.

In the remainder of the paper, we discuss the synthesis problem of Reo circuits where the input specification of the desired coordination is given by UML sequence diagrams, as presented in the next section.

3. UML sequence diagrams

UML Sequence Diagrams are used to model the dynamic behavior of systems. Graphically, a UML SD has two dimensions: an horizontal dimension representing the components participating in the scenario, and a vertical dimension representing time. Every component has a vertical dashed line called its lifeline. SDs focus on the message interchange among a number of lifelines. An SD describes an interaction by focusing on the sequence of messages exchanged during a system run. See Fig. 4 as an example of sequence diagrams that describe the interactions in the login phase of an online banking scenario. A UML SD is represented as a rectangular frame labeled by the keyword sd followed by the name of the interaction. The vertical lines in the SD represent lifelines for the individual participants in the interaction.

A message defines a particular communication between lifelines of an interaction. It can be either asynchronous (represented by an open arrow head) or synchronous (represented by a filled arrow head). Additionally, there are two special kinds of messages: lost (respectively, found) messages (not shown in Fig. 4), which are denoted by a small black circle at the
end (respectively, the start) of the arrow representing the message. Note that what we are interested in is the coordination between components/services, so we consider only a subset of the UML2.0 SDs. What is left out of this subset consists of the following two parts:

1. The internal behavior of processes represented as actions within the same lifelines in SDs (like the check action in Fig. 4) will not be considered in the synthesis process. This is justified because we intend the synthesized coordination in our approach to reflect only the inter-process interaction in a system. We specifically do not wish to obtain a global state machine that intertwines both the behavior of the components in a system and the interactions among them.

2. The neg, ignore and critical operators on sequence diagrams. These are not operators for composition of sequence diagrams, since they take as input a single sequence diagram and “refine” its semantics either by permitting more behaviors or by ruling out certain behaviors.

The internal behavior or internal actions within a lifeline in an SD (such as check in SD EnterPwd in Fig. 4) constitute an assumption that the component (Bank in Fig. 4) must fulfill. For the example, the check in EnterPwd means that after receiving the message verifywithBank, the bank must check the validity of the account number and password pair. This can be modeled by a constraint automaton for the bank. For the synthesis of the Reo circuit that captures the inter-process communication in the system, this constraint automaton is irrelevant. However, to verify the UML SD of a system via Reo model checkers, the automata for the system components (like the bank) may become important as well, for instance, to prove certain properties using the assume/guarantee paradigm. Given that in this paper we focus on the construction of the Reo circuit, we ignore such internal behavior, and assume that the automata for the components still can (and will) be used for the analysis of UML SDs.

3.1. Syntax

The signature of a basic UML sequence diagram is defined as follows:

**Definition 1.** A basic sequence diagram $sd$ is given by a tuple

$$ (I, \text{Loc}^*, \text{Mes}, \text{Loc}_{ini}, \text{loc}, E, \leq) $$
where

- \( l \) is a set of instance identifiers corresponding to the participants in the interaction described by the diagram;
- \( Loc \) is a set of locations, and \( Loc^* = Loc \cup \{ \bullet \} \);
- \( Mes = M \times Mtype \) denotes the set of messages, where \( M \) is a set of message labels, and \( Mtype = \{ \rightarrow, \leftarrow, \text{opt}, \text{seq}, \text{par} \} \), where \( \rightarrow \) and \( \leftarrow \) denote asynchronous and synchronous messages, respectively;
- \( Loc_{\text{ini}} \subseteq Loc \) is a set of initial locations;
- \( loc : I \rightarrow 2^{loc} \) associates to each instance a set of locations. The function satisfies the following conditions expressing disjointness and conformity with the initial constraints, respectively,

\[
\forall i, j \in I, i \neq j \ . \ loc(i) \cap loc(j) = \emptyset \tag{1}
\]

\[
\forall i \in I . \ card(loc(i) \cap Loc_{\text{ini}}) = 1 \tag{2}
\]

where for a set \( S \), \( card(S) \) returns the cardinality of \( S \).

- \( E \subseteq Loc^* \times Mes \times Loc^* \) is a relation such that a tuple \( (l_1, (m, t), l_2) \in E \) represents a message \( m \) exchanged between \( l_1 \) and \( l_2 \), and \( t \) denotes the type of the message (synchronous or asynchronous).
- \( \leq \subseteq Loc \times Loc \) is a partial order capturing the relative positions of locations within each linelife in the diagram. \( \square \)

Note that in general, for a relation in \( E \) to represent a communication between participants in a sequence diagram, its source and target locations cannot be the same, i.e., the following property is assumed:

\[ \forall (l_1, (m, t), l_2) \in E . \ l_1 \neq l_2 \]

Observe that this assumption does not rule out a process sending a message to itself: it only requires for the source and the target locations, which may still reside on the same linelife, to be different.

Within this model the following function returns the next location in a particular linelife. Formally,

- \( next : Loc \rightarrow Loc \) is defined as

\[ next(l) = l' \iff \exists i \in I . l \in loc(i) \land l < l' \land \forall l'' \in loc(i). l < l'' \Rightarrow l' \leq l'' \]

where \( < \) stands, as usual, for the largest irreflexive subset of \( \leq \).

Let \( l_1, l_2 \) range over \( Loc \), and \( \Sigma_m \) be the set of communication events executed concerning messages exchanged in a sequence diagram \( sd \). For every relation \( (l_1, (m, t), l_2) \in E \), where \( (m, t) \in Mes \), there are two events in \( \Sigma_m \), corresponding to the sending and receiving of \( m \), which occur at locations \( l_1 \) and \( l_2 \) respectively. Each event \( e \in \Sigma_m \) has one of the following forms:

1. \( (l_1 \rightarrow l_2, m) \) - \( l_1 \) sends the asynchronous message \( m \) to \( l_2 \).
2. \( (l_1 \leftarrow l_2, m) \) - \( l_1 \) receives the asynchronous message \( m \) from \( l_2 \).
3. \( (l_1 \rightarrow l_2, m) \) - \( l_1 \) sends the synchronous message \( m \) to \( l_2 \), and
4. \( (l_1 \leftarrow l_2, m) \) - \( l_1 \) receives the synchronous message \( m \) from \( l_2 \).

The cases for \textit{lost} and \textit{found} messages can be represented by replacing \( l_2 \) by \( \bullet \) in the first two cases. For an arbitrary event \( e \in \Sigma_m \),

\[ type(e) \in \{ \rightarrow, \leftarrow, \text{opt}, \text{seq}, \text{par} \} \]

denotes the type of the event (sending/receiving for asynchronous messages and sending/receiving of synchronous messages). The direction of the arrow for \( type(e) \) specifies whether \( e \) is the sending or receiving event of a message, while solid-head and open-head arrows represent synchronous and asynchronous messages, respectively. For example, if \( e = (l_1 \leftarrow l_2, m) \), then \( type(e) = \leftarrow \), which means that \( e \) is a receiving event of an asynchronous message.

Since the sending and receiving events for a synchronous message must happen simultaneously, for \( e = (l_1 \rightarrow l_2, m) \) and its corresponding receiving event \( \overline{e} = (l_2 \leftarrow l_1, m) \), we use \( \langle e, \overline{e} \rangle \) to denote the occurrence of such a pair of events, and

\[ \Sigma = \Sigma_m \setminus \{ e \mid type(e) = \rightarrow \lor type(e) = \leftarrow \} \cup \{ (e, \overline{e}) \mid type(e) = \rightarrow \} \]

UML SDs may contain sub-interactions called \textit{interaction fragments} that can be structured and combined using \textit{interaction operators}. There are several possible operators in UML for composing sequence diagrams, such as \textit{alt}, \textit{opt}, \textit{strict}, \textit{par}, \textit{seq} and \textit{loop}. Depending on the operator used, an interaction fragment contains one or more operands. For \textit{loop} and \textit{opt}, the fragment has exactly one operand, while the other operators have several operands.\(^1\)

\(^1\) As mentioned earlier, a few more operators are given in [30] than the ones we consider here, for instance \textit{neg}, \textit{ignore} and \textit{critical}. These operators take as input a single sequence diagram and “refine” its semantics either by permitting more behaviors or by ruling out certain behaviors, but they are not used for composition of sequence diagrams. The negative operator \textit{neg} designates that the fragment represents traces that are invalid; the \textit{ignore} operator designates that there are some messages that are not shown within the fragment, which are insignificant and can appear anywhere in the traces; the \textit{critical} operator designates that the fragment represents a critical region, which means that the traces of the region cannot be interleaved by other event occurrences (on those lifelines covered by the region). Such operators provide some sort of coercion which restricts or expands the underlying possible behaviors. They are useful, for example, for verifying system properties and test case construction. We can easily handle the cases for these operators in our framework. For simplicity, we will not consider these operators further in this paper.
Definition 2. The syntax for sequence diagrams is defined as follows:

\[ SD ::= sd \ | \ \text{alt}(g_1 : SD, g_2 : SD) \ | \ \text{par}(SD, SD) \ | \ \text{strict}(SD, SD) \ | \ \text{seq}(SD, SD) \ | \ \text{opt}(SD) \ | \ \text{loop}(g : SD) \]

where \( sd \) denotes a basic sequence diagram, and \text{alt}, \text{par}, \text{strict}, \text{seq}, \text{opt} and \text{loop} are the interaction composition operators. \( \square \)

3.2. Semantics

The semantics for a sequence diagram \( sd = (l, \text{Loc}, \text{Mes}, \text{Loc}_{\text{init}}, \text{loc}, E, \leq) \) can be defined in terms of coalgebras as \((C, \langle e, \overline{e} \rangle : C \rightarrow \mathcal{P}(\Sigma_m) \times C^2)\), where \( C \) is the set of the possible configurations of \( sd \), \( e : C \rightarrow \mathcal{P}(\Sigma_m) \) denotes the set of active events in a configuration, and \( \overline{e} : C \rightarrow C^2 \). Thus, the semantics of a sequence diagram is a coalgebra of the functor

\[ T = \mathcal{P}(\Sigma_m) \times -^\Sigma \]

(3)
together with an initial configuration \( c_0 \), which is given by the tuple of initial locations \( c_0 \) = \( \prod_{l \in \text{Loc}_{\text{init}}} l \). The set of initially active events is

\[ \epsilon(c_0) = \{ e \mid \pi(e) \in \text{Loc}_{\text{init}} \wedge \text{type}(e) \neq \leftarrow \} \]

A configuration of a sequence diagram denotes a global state, composed of the local states of its participants. For every configuration, there is a set of active events that may happen in that configuration.

Definition 3. A configuration \( c \) of a sequence diagram is a tuple of the local states (locations) of its participants. \( \square \)

Suppose \( C \) denotes the set of all possible configurations. For any event \( e \in \Sigma_m \), the location at which \( e \) happens is defined by \( \pi(e) = l \) iff \( e = (l, \cdots) \). This notation generalizes for a set of events \( \Sigma' \subseteq \Sigma_m \) as \( \pi(\Sigma') = \{ \pi(e) \mid e \in \Sigma' \} \). Here \( \pi(e) \in c \) means \( \pi(e) \) is a location in \( c \). A configuration \( c \) is called final and denoted by \( c_f \) if \( \epsilon(c) = \emptyset \).

We now define \( \alpha \), the curried version of \( \overline{e} \), by enumerating all possible transition schemes. For synchronous messages, the events modelling both sending and receiving occur simultaneously (i.e., in an atomic, non interruptible way): no other event can occur in between. So if the current configuration is \( c \) and both the sending event \( e = (l_1 \rightarrow l_2, m) \) and its corresponding receiving event \( \overline{e} = (l_2 \leftarrow l_1, m) \) are active, i.e., \( e \in c \), \( \overline{e} \in e(c) \), then we have

\[ \alpha(c, \langle e, \overline{e} \rangle) = c[\text{next}(l_1)/l_1, \text{next}(l_2)/l_2] \]

and

\[ \epsilon(\alpha(c, \langle e, \overline{e} \rangle)) = \epsilon(c) \setminus \{ e, \overline{e} \} \cup \left\{ e' \mid \pi(e') = \text{next}(l_k) \wedge \text{type}(e') \neq \leftarrow \right\} \]

For asynchronous messages, however, when the sending event occurs, the location of the sender will be updated to the next location in its lifeline, while the locations of the other participants remain unchanged. The sending event is therefore removed from the set of active events. On the other hand, the corresponding receiving event will be added to this set. Furthermore, the events at the next location of the sender’s lifeline will become active in the new configuration. If \( e = (l_1 \rightarrow l_2, m) \) is active in configuration \( c \), we have

\[ \alpha(c, e) = c[\text{next}(l_1)/l_1] \]

and

\[ \epsilon(\alpha(c, e)) = \epsilon(c) \setminus \{ e \} \cup \{ (l_2 \leftarrow l_1, m) \} \cup \{ e' \mid \pi(e') = \text{next}(l_1) \wedge \text{type}(e') \neq \leftarrow \} \]

Dually, when an asynchronous message is received, the receiver changes to the next location in its lifeline, while the locations of all other participants remain unchanged. Formally, if \( e = (l_1 \leftarrow l_2, m) \) is active in configuration \( c \), we have

\[ \alpha(c, e) = c[\text{next}(l_1)/l_1] \]

and

\[ \epsilon(\alpha(c, e)) = \epsilon(c) \setminus \{ e \} \cup \{ e' \mid \pi(e') = \text{next}(l_1) \wedge \text{type}(e') \neq \leftarrow \} \]

The case of a lost message, represented by the event \( e = (l \rightarrow \bullet, m) \), is similar to the asynchronous communication: the sender updates its location and \( e \) is removed from the set of active events. However, no corresponding receiving event becomes active. Similarly, for a found message, when a receiving event \( e = (l \leftarrow \bullet, m) \) occurs, only the location of the receiver is updated and \( e \) is removed from the set of active events. Both cases are, therefore, handled by

\[ \alpha(c, e) = c[\text{next}(l)/l] \]
and
\[ \epsilon(\alpha(c, e)) = \epsilon(c) \setminus \{e\} \cup \{e' \mid \pi(e') = \text{next}(l) \land \text{type}(e') \neq \epsilon\} \]
assuming the corresponding events are enabled in configuration \(c\).

Additionally, note that for a coregion in one lifeline in a sequence diagram the order of event occurrences is not significant. In our model, a coregion is taken as one location. Let \(l\) be the location for a coregion, we use
\[ \Sigma_l = \{e \mid \pi(e) = l\} \]
to denote all the events that are active in the coregion. If \(e \in \Sigma_l\), then
\[ \alpha(c, e) = \begin{cases} \Sigma_l = \emptyset & \Rightarrow c[\text{next}(l)/l] \\ \Sigma_l \neq \emptyset & \Rightarrow c \end{cases} \]

The semantics of an interaction fragment depends on the operator used in its definition, as informally described in the UML superstructure specification [30]. These operators have been formally investigated in [33], which leads to an algebra for building new sequence diagrams from existing ones. The denotations given in the following paragraphs are similar to those in [33] for the operators. However, since the functor type is different than the one in [33], the definitions of all the operators are also changed. Let \(\llbracket \text{sd}_i \rrbracket = (C_i, \langle \epsilon_i, \alpha_i(\alpha_2), c_0 \rangle)\) for \(i = 1, 2\), the definitions are given as follows:

**Strict sequential composition: strict** \((sd_1, sd_2)\)

The transition structure in\(^3\)
\[ \llbracket \text{strict}(sd_1, sd_2) \rrbracket = (C, \langle \text{strict}(\epsilon_1, \alpha_2), \text{strict}(\alpha_1, \alpha_2), c_0 \rangle) \]
is defined over \(C = C_1 \cup C_2 \setminus \{c \mid c \in C_1 \land \epsilon_1(c) = \emptyset\}\) and \(c_0 = c_0^1\) as follows
\[ \text{strict}(\epsilon_1, \alpha_2)(x) = \begin{cases} x \in C_1 \land \epsilon_1(x) \neq \emptyset & \Rightarrow \epsilon_1(x) \\ x \in C_2 & \Rightarrow \alpha_2(x) \end{cases} \]
\[ \text{strict}(\alpha_1, \alpha_2)(x, e) = \begin{cases} x \in C_1 \Rightarrow \text{let } x' = \alpha_1(x, e) \text{ in } \epsilon_1 x' = \emptyset \Rightarrow c_0^2 \\ \text{otherwise } x' \end{cases} \]

**Parallel: par** \((sd_1, sd_2)\)

In this case we consider \(C = \{(c_1, c_2) \mid \text{for } i = 1, 2, c_i \in C_i\}\) and \(c_0 = (c_0^1, c_0^2)\) in
\[ \llbracket \text{par}(sd_1, sd_2) \rrbracket = (C, \langle \text{par}(\epsilon_1, \alpha_2), \text{par}(\alpha_1, \alpha_2), c_0 \rangle) \]
where the transition structure is defined as
\[ \text{par}(\epsilon_1, \epsilon_2)(x) = \epsilon_1(\pi_1 x) \cup \epsilon_2(\pi_2 x) \]
\[ \text{par}(\alpha_1, \alpha_2)(x, e) = \begin{cases} e \in \Sigma_1 \Rightarrow \text{let } x' = \alpha_1(\pi_1 x, e) \text{ in } (x', \pi_1 x) \\ e \in \Sigma_2 \Rightarrow \text{let } x' = \alpha_2(\pi_2 x, e) \text{ in } (\pi_1 x, x') \\ \text{otherwise } x \end{cases} \]

**Option: opt** \((sd_1)\)

The purpose of \(\text{opt}(sd_1)\) is to offer an alternative between an empty scenario (in which ‘nothing happens’) and the activation of its (sole) operand, \(sd_1\). To formalize its meaning we need to introduce a new event – \textit{skip} – to capture the absence of effective behavior. The event \textit{skip} does nothing but terminates successfully. Then
\[ \llbracket \text{opt}(sd_1) \rrbracket = (C, \langle \text{opt}(\epsilon_1), \text{opt}(\alpha_1), c_0 \rangle) \]
where \(C = C_1\) and \(c_0 = c_0^1\). The transition structure is defined as
\[ \text{opt}(\epsilon_1)(x) = \begin{cases} x = c_0 \Rightarrow \epsilon_1(x) \cup \{\text{skip}\} \\ \text{otherwise } \epsilon_1(x) \end{cases} \]
\[ \text{opt}(\alpha_1)(x, e) = \begin{cases} x \neq c_0 \land e \in \Sigma_1 \Rightarrow \alpha_1(x, e) \\ x = c_0 \land e = \text{skip} \Rightarrow \text{let } x' \in C \text{ in } \epsilon x' = \emptyset \Rightarrow x' \\ x = c_0 \land e \in \Sigma_1 \Rightarrow \alpha_1(c_0, e) \\ \text{otherwise } x \end{cases} \]

\(^2\) The reasons for the difference are: (1) the functor used in [33] also takes internal actions within one lifeline into consideration, but this is not useful for our synthesis of connectors; (2) the functor makes explicit that a set of enabled events is present in the initial state, i.e., before any interaction occurs; and (3) the carrier of the corresponding final coalgebra takes a quite simple form \(v = \mathcal{P}(\Sigma_{m})^{\Sigma_{m}}\), i.e., functions that relate each \(\Sigma\)-trace to the set of enabled events upon completion of its execution.

\(^3\) To avoid an excessive notational burden, we use the same syntax for the combinator over sequence diagrams and its denotation in the proposed semantics.
Semantically, an option is equivalent to a nondeterministic choice between two operands where one operand has non-empty content and the other is empty. So when \( \text{opt}(sd_1) \) is at \( c_0 \), the choice between \( \text{skip} \) and \( e \in \Sigma_1 \) is nondeterministic.

**Choice:** \( \text{alt}(g_1 : sd_1, g_2 : sd_2) \)

Denoting an alternative form of aggregation of sequence diagrams, it requires that \( c_1^0 = c_0^0 \) and \( c_1^1 = c_1^2 \), and that if \( g_i \) is satisfied at the initial configuration \( c_0 \), then all events in \( \Sigma_1^0 \) become active in \( c_0 \). If neither guard condition is satisfied in the initial configuration, then neither \( sd_1 \) nor \( sd_2 \) becomes active, and an empty scenario (represented by the \( \text{skip} \) event) will be executed. Therefore, \( c_0 = c_0^0, c_f = c_1^1 \) and \( C = \{ c_0^0, c_f \} \cup ( C_1 \setminus \{ c_0^1, c_f^1 \} ) \cup ( C_2 \setminus \{ c_0^2, c_f^2 \} ) \).

Formally,

\[
[\text{alt}(g_1 : sd_1, g_2 : sd_2)] = (C, (\text{alt}(\epsilon_1, \epsilon_2), \text{alt}(\alpha_1, \alpha_2)), c_0)
\]

with

\[
\text{alt}(\epsilon_1, \epsilon_2)(x) = \begin{cases} 
    x = c_0 \land c_0 \vdash (g_1 \land g_2) & \Rightarrow \Sigma_1^0 \cup \Sigma_2^0 \\
    x = c_0 \land \exists i \{ c_0 \land g_i \land g_i \neq g \} & \Rightarrow \Sigma_1^0 \\
    x = c_0 \land \exists i \{ c_0 \land g_i \land g_i \neq g \} & \Rightarrow \Sigma_2^0 \\
    x = c_0 \land \land_{i=1,2} c_0 \land g_i & \Rightarrow \{ \text{skip} \} \\
    x \in C_1 \setminus \{ c_1^0, c_f^1 \} & \Rightarrow \epsilon_1(x) \\
    x \in C_2 \setminus \{ c_2^0, c_f^2 \} & \Rightarrow \epsilon_2(x) \\
    x = c_f & \Rightarrow \emptyset
\end{cases}
\]

\[
\text{alt}(\alpha_1, \alpha_2)(x, e) = \begin{cases} 
    x = c_0 \land c_0 \vdash g_i \land e \in \Sigma_i & \Rightarrow \alpha_i(c_i^0, e) \text{ for } i = 1, 2 \\
    x = c_0 \land \land_{i=1,2} c_0 \land g_i \land e = \text{skip} & \Rightarrow c_f \\
    x \in C_1 \setminus \{ c_1^0 \} \land e \in \Sigma_1 & \Rightarrow \alpha_1(x, e) \\
    x \in C_2 \setminus \{ c_2^0 \} \land e \in \Sigma_2 & \Rightarrow \alpha_2(x, e) \\
    \text{otherwise} & \Rightarrow x
\end{cases}
\]

where \( x \) is a configuration in \( C \), and \( e \) is an event in either \( \Sigma_1 \) or \( \Sigma_2 \).

**Weak sequential composition:** \( \text{seq}(sd_1, sd_2) \)

The case for weak sequential composition \( \text{seq}(sd_1, sd_2) \) for \( sd_i, i = 1, 2 \) is a bit more demanding because its definition depends on whether the operands share any lifelines. If such is the case, then for every identifier \( s \in l_1 \cup l_2 \), all event occurrences on \( s \) in \( sd_1 \) must happen before those on \( s \) in \( sd_2 \). However, all other events in \( sd_1 \) and \( sd_2 \) on lifelines not in \( l_1 \cup l_2 \) may occur in any order. Note that if the operands involve disjoint sets of participants (i.e., \( l_1 \cap l_2 = \emptyset \)), the weak sequencing reduces to a parallel merge.

Assume an identifier \( s \), such that \( l_1 \cap l_2 = \{ s \} \), and the functions \( \text{loc}_1 \) and \( \text{loc}_2 \) assigning locations to the instances in \( sd_1 \) and \( sd_2 \), respectively. Let \( \text{loc}(s) = \text{loc}_1(s) \cup \text{loc}_2(s) \). Furthermore, and without loss of generality, let \( C_1 = \text{loc}_1(s) \times L \) and \( C_2 = \text{loc}_2(s) \times K \) be the set of configurations for \( sd_1 \) and \( sd_2 \) respectively, where \( L = \prod_{i \in l_1 \setminus \{ s \}} \text{loc}_1(i) \) and \( K = \prod_{j \in l_2 \setminus \{ s \}} \text{loc}_2(j) \).

Then, define

\[
[\text{seq}(sd_1, sd_2)] = \begin{cases} 
    (C, (\text{seq}(\epsilon_1, \epsilon_2), \text{seq}(\alpha_1, \alpha_2)), c_0) & \text{if } l_1 \cap l_2 = \{ s \} \\
    [\text{par}(sd_1, sd_2)] & \text{if } l_1 \cap l_2 = \emptyset
\end{cases}
\]

with \( C = \text{loc}(s) \times L \times K \) and \( c_0 = (\{ \pi_1, \pi_2 \}, c_0^1, c_2^0) \).

The transition structure is given by

\[
\text{seq}(\alpha_1, \alpha_2)(x, e) = \text{let } [s] = l_1 \cap l_2
\]

\[
\begin{cases} 
    \pi(e) \cap \text{loc}(s) \neq \emptyset \land \langle \pi_1, \pi_2 \rangle x \in C_1 & \Rightarrow \\
    \text{let } x' = \alpha_1((\langle \pi_1, \pi_2 \rangle x, e), \text{ in }
    \begin{cases} 
        \pi_1 x' = \pi_1 c_1^0 & \Rightarrow (\pi_1 c_2^0, \pi_2 x, \pi_3 x) \\
        \text{otherwise} & \Rightarrow (\pi_1 x', \pi_2 x, \pi_3 x)
    \end{cases}
    \pi(e) \cap \text{loc}(s) \neq \emptyset \land \langle \pi_1, \pi_3 \rangle x \in C_2 & \Rightarrow \\
    \text{let } x' = \alpha_2((\langle \pi_1, \pi_3 \rangle x, e), \text{ in }
    \begin{cases} 
        \pi_1 x' = \pi_1 c_1^0 & \Rightarrow (\pi_1 c_2^0, \pi_2 x, \pi_3 x) \\
        \text{otherwise} & \Rightarrow (\pi_1 x', \pi_2 x, \pi_3 x)
    \end{cases}
\end{cases}
\end{cases}
\]

\[
\begin{cases} 
    e \in \Sigma_1 & \Rightarrow \\
    \langle \pi_1, \pi_2 \rangle x \in C_1 & \Rightarrow \text{let } x' = \alpha_1((\langle \pi_1, \pi_2 \rangle x, e), \text{ in }
    \begin{cases} 
        \pi_1 x' = \pi_1 c_1^0 & \Rightarrow (\pi_1 c_2^0, \pi_2 x, \pi_3 x) \\
        \text{otherwise} & \Rightarrow (\pi_1 x', \pi_2 x, \pi_3 x)
    \end{cases}
    \pi(e) \cap \text{loc}(s) = \emptyset & \Rightarrow \\
    e \in \Sigma_2 & \Rightarrow \\
    \langle \pi_1, \pi_3 \rangle x \in C_2 & \Rightarrow \text{let } x' = \alpha_2((\langle \pi_1, \pi_3 \rangle x, e), \text{ in }
    \begin{cases} 
        \pi_1 x' = \pi_1 c_1^0 & \Rightarrow (\pi_1 c_2^0, \pi_2 x, \pi_3 x) \\
        \text{otherwise} & \Rightarrow (\pi_1 x', \pi_2 x, \pi_3 x)
    \end{cases}
\end{cases}
\end{cases}
\]
We use $\epsilon$ as an abbreviation for $\text{seq}(\epsilon_1, \epsilon_2)$, and for any $c \in C$,
$$
\epsilon(c) = \bigcup_{(a_i, \pi_{i+1}c) \in C_i, i=1,2} \epsilon_i((a_i, \pi_{i+1}c)) \setminus \{e \mid \pi(e) \in \text{loc}(s) \land \pi(e) \neq \pi_1(c)\}
$$
where $a_i \in \text{loc}(s), i = 1, 2$ are two locations such that
$$
(a_1, \pi_{2}c) \in C_1 \land (a_2, \pi_{2}c) \in C_2 \land (\pi_1c = a_1 \lor \pi_1c = a_2)
$$
This definition can be easily generalized to an arbitrary number of shared lifelines in $sd_1$ and $sd_2$.

**Loop: loop** ($g : sd_1$)

Finally, the semantics of the iteration combinator is given by
$$
\llbracket \text{loop}(g : sd_1) \rrbracket = (C, \langle \text{loop}(\epsilon_1), \text{loop}(\alpha_1) \rangle, c_0)
$$
over $C = C_j$ and $c_0 = c_j'$, and with the following transition structure

\[
\text{loop}(\epsilon_1)(x) = \begin{cases}
  x = c_0 & \Rightarrow \epsilon_1(x) \cup \{\text{skip}\} \\
  x \in C \setminus \{c_0\} & \Rightarrow \epsilon_1(x)
\end{cases}
\]

\[
\text{loop}(\alpha_1)(x, e) = \begin{cases}
  x = c_0 \land c_0 \neq g \land e = \text{skip} \Rightarrow c_f \\
  x \neq c_0 \lor (x = c_0 \land c_0 \equiv g) \land e \in \epsilon_1(x) \Rightarrow \\
  \begin{cases}
    \text{let } x' = \alpha_1(x, e) \text{ in} \\
    \epsilon(x') = \emptyset \land x' \equiv g \Rightarrow c_0 \\
    \text{otherwise } x'
  \end{cases}
\end{cases}
\]

4. From UML sequence diagrams to Reo

We now address the issue of constructing Reo circuits from scenario specifications represented by UML SDs. Since the source and the sink nodes of a Reo circuit are used for components to exchange data through write and take operations, we first need to identify the node set $\mathcal{N}$ of a circuit involved in an interaction. Assume the participants (components) involved in the interaction are represented by the set of lifelines $L = \{p_1, \ldots, p_n\}$. For simplicity, and without loss of generality, we assume every component has only one input and one output port connected to the corresponding sink and source nodes of the Reo circuit. Therefore, our starting point is a description of a component connector by its source nodes $C_1, \ldots, C_n$ and sink nodes $D_1, \ldots, D_n$, such that each component $p_i$ can write messages to the node $C_i$ and take messages from the node $D_i$. Additionally, the interaction behavior coordinated by the connector is described by a set of UML SDs.

In the sequel, let $\mathcal{N} = \{C_1, \ldots, C_n\} \cup \{D_1, \ldots, D_n\}$ contain all nodes attached to the components involved in a scenario specification, where we assume that the $C_i$'s are source nodes and the $D_i$'s are sink nodes. Our goal is to construct a Reo circuit $R$ with source nodes $C_1, \ldots, C_n$ and sink nodes $D_1, \ldots, D_n$, such that the behavior represented by the scenario specification is permitted by the communication protocol encoded in $R$.

For the construction of $R$, we first consider the construction of Reo circuits for basic sequence diagrams without interaction operators. Assume that there are $n$ lifelines $p_1, \ldots, p_n$ in a basic SD. Every lifeline $p_i$ represents an individual participant in the interaction, and we can derive an order of event occurrences along the lifeline, which is significant as it denotes the order in which these events occur.

4.1. Reo circuits for individual participants

The first step in our synthesis approach consists of deriving a sequencer for every lifeline $p_i$ in a basic sequence diagram. If there are $k$ events (sending and receiving of messages) on a lifeline $p_i$ (and thus $k$ locations $l_1, \ldots, l_k$), then the sequencer corresponding to $p_i$ also has $k$ nodes (e.g., $A_1, \ldots, A_k$), the order of which corresponds to the order of the events/locations on $p_i$. Without loss of generality, we assume here that the component/service/process that implements the behavior described by $p_i$ produces/consumes all of the $k$ messages corresponding to these $k$ events through two separate, dedicated I/O ports (e.g., $C_i$ for output port and $D_i$ for input port). Next, we link each of the two ports of the process that implements $p_i$ to its corresponding nodes of the sequencer using synchronous drains and either synchronous or filter channels. If the event corresponding to a location $l_j$ involves the sending of a message $m$, then a filter with pattern $m$ is used to link the node (output port) $C_i$ to the synchronous drain channel connected to that location’s respective sequencer node $A_j$. On the other hand, if the event involves receiving a message, then a synchronous channel is used to link the node (input port) $D_i$ to the synchronous drain channel connected to that location’s respective sequencer node $A_j$. Note that when a lifeline has only one location, i.e., there is only one sending or receiving event on the lifeline, there is no need for a sequencer; in other words, a one-node sequencer (and its attached synchronous drain) all degenerate into a single node.
Fig. 5. Introducing sequencers for individual lifelines in a scenario (1).

Fig. 6. Introducing sequencers for individual lifelines in a scenario (2).

Fig. 7. Reo circuit for a scenario.

Fig. 5 shows an example of our approach. There are two participants $p_1$ and $p_2$ involved in the scenario. The interaction between them is shown by the messages $m_1$, $m_2$ and $m_3$, which are all synchronous in this example. We first consider $p_1$, whose behavior involves first sending message $m_1$, then receiving message $m_2$ and finally sending message $m_3$, sequentially. There are three events happening on the lifeline, so we introduce a 3-node sequencer, where the first and the last nodes are connected to the node $C_1$ by two filters respectively, and the node in the middle is connected to $D_1$ via a synchronous channel. The patterns $m_1$ and $m_3$ on the filters ensure that $p_1$ can write only these two messages out, and the synchronization between the nodes of the sequencer and the nodes of the channels connected to $C_1$ and $D_1$ ensures that the sending of $m_1$ happens first. Note that on the $p_2$ side, there is no restriction on the channel $A_2D_1$, like a filter, to ensure that the message received through this channel is $m_2$. This is guaranteed by the filter in the synthesized part for $p_2$, as shown in Fig. 6. In other words, the type of a message is always guaranteed by its sender.

4.2. Reo circuits for basic SDs

After we derive the sequencers for all participants from their respective lifelines in isolation, we can connect their respective nodes $A_i$, $B_j$, . . . pairwise by synchronous or asynchronous Reo channels, according to the types and the order of messages, as defined in a basic SD. If $(l_1, (m, t), l_2) \in E$ represents the exchange of a message $m$ between location $l_1$ and $l_2$, $m$ is a synchronous (respectively, asynchronous) message, then a synchronous (respectively, asynchronous) channel is used to link the nodes corresponding to $l_1$ and $l_2$. The direction of the channel is consistent with the direction of the message exchange, i.e., the source (respectively, sink) node of the channel corresponds to the source (respectively, target) location of the message.

For example, the Reo circuit on the right-hand-side in Fig. 7 is the result of composing the Reo circuits in Figs. 5 and 6, according to the basic SD on the left-hand-side of Fig. 7, where all messages are synchronous. In the synthesized connector for the whole SD, source nodes $C_i$ and sink nodes $D_i$ are attached to $p_i$ respectively. Component $p_1$ can write messages $m_1$ and $m_3$ to the source node $C_1$, and receive message $m_2$ from the sink node $D_1$. The filter connected to $C_1$ and the sequencer ensure that $p_1$ receives some message after it sends out the message $m_1$ and before it sends out the message $m_3$. From the synchronous channel between $A_2$ and $B_2$, and the filter $C_2B_2$, we know that the message received by $p_1$ is $m_2$.

Messages in a UML SD can also be asynchronous, which are graphically represented by open arrowheads, such as the message displayindexpage and displayonlineBank in the SD UserArrives in Fig. 4. There are different possibilities for the ordering of events for asynchronous messages, as shown in Fig. 8.
Since the order of asynchronous message passing may be different, it is not possible to use only one asynchronous channel for all asynchronous communications. In Fig. 9, we give the Reo circuits for the scenarios in Fig. 8(a) and (b) respectively. The FIFO1 channels are used for asynchronous messages, where the ordering of events is controlled by the topology of the Reo circuits.

There can be a coregion area in a lifeline in UML SDs where the order of event occurrences on the lifeline is not significant. Fig. 8(c) shows an example of a coregion. In this case, the corresponding Reo circuit is as shown in Fig. 10(a), in which an exclusive router EXR is needed, which is, in turn, composed of five synchronous channels, two lossy synchronous channels and a synchronous drain, as shown in Fig. 10(b).

One participant in a scenario can send the same message multiple times. Fig. 11 shows an example. In this case, an exclusive router EXR can be used on the side of the sender, where the messages through the two sink nodes of EXR are ordered by the sequencer, and connected to the nodes corresponding to the different receivers, respectively. Another possible solution is to use lossy synchronous channels on the sender side where every time when the message \( m_1 \) is sent by \( p_1 \), it can be transmitted through only one of the branches, and lost by the other ones. The order is decided, again, by the sequencer.

Messages in UML SDs can be lost. Lost messages are messages with known sender, but the reception of the message does not happen. Such a situation can be captured by the Lost connector as shown in Fig. 12, where the source node \( B_{\text{lost}} \) can
A message can also be found. A found message is a message whose receiving event occurrence is known, but has no sending event occurrence. This is because the origin of the message is outside the scope of the participants. We can describe such messages by the Found connector in Fig. 12 whose found message is \( m \). The sink node \( A_{\text{found}} \) can be connected to some node \( B_i \) for the receiver of the message \( m \) in the corresponding Reo circuit.

### 4.3. Composing Reo circuits following UML operators

So far our Reo circuits focused on basic SDs. Next, we describe our compositional construction of the Reo circuits of basic SDs following a structural induction approach. To structure the connectors according to the operators in UML SDs, we use a general structure for the Reo circuits as shown in Fig. 13: \( R_{sd} \) is the Reo circuit for a basic SD \( sd \), which is obtained as in the previous section. In this construction, six \((3 \times 2)\) more nodes are added to \( R_{sd} \). \( A_{sd} \) and \( B_{sd} \) are two nodes synchronized with the nodes of the sequencers inside \( R_{sd} \), that correspond to the first sending event and the last receiving event in \( sd \), respectively. If the source node \( A_{sd} \) is fed from outside with some data element, then it is put into the buffer between \( A_{sd} \) and \( B_{sd} \). As soon as \( B_{sd} \) takes the data element from the buffer, the subcircuit \( R_{sd} \) is "activated" by the first message received through some \( C_i \). Similarly, the communication via the subcircuit stops as soon as a data element arrives at \( B_{sd} \), which puts it into the buffer between \( B_{sd} \) and \( B_{sd} \). Thus, data-flow at the sink node \( B_{sd} \) can be viewed as a signal that \( R_{sd} \) has terminated.

As an example, the generalized Reo circuit for the connector in Fig. 7 is shown in Fig. 14.

Assume we have already constructed the circuits for the interaction fragments (basic SDs) of a sequence diagram SD. We now explain how to construct a Reo circuit \( R_{SD} \) for the whole diagram. Note that in the transformation rules for \texttt{alt}, \texttt{par}, \texttt{strict} and \texttt{seq}, we only consider the case for two operands. In fact, using the graph transformation approach proposed in [20,24] to instantiate parameterized Reo circuits, these rules extend to an arbitrary number of operands.

---

4 For simplicity, we assume here a total order on the events. In fact, we can handle the general case as well. If there are \( n \) sending events and any of them can happen as the first one, then we can use an exclusive router \( \text{EXR} \) with \( n \) sink nodes as in Fig. 10, and connect each of the sink nodes in \( \text{EXR} \) with one node corresponding to one such sending event by a synchronous drain. Similarly, we can deal with the case for multiple receiving events.
For $SD = \text{alt}(g_1 : sd_1, g_2 : sd_2)$, the Reo circuit $R_{SD}$ is obtained by combining $R_{sd_1}$ and $R_{sd_2}$ with a replicator connected to three filters and one exclusive router. The patterns on the filters correspond to the two guard conditions $g_1$, $g_2$, and the conjunction of $\neg g_1$ and $\neg g_2$. The data item $d$ to be transmitted via the channel $A_{SD} \bar{A}_{SD}$ is related to the guard condition that may be obtained from other nodes (i.e., $g_1$ and $g_2$ in Fig. 15). In this case, there is another Reo circuit $R$ connected to $A_{sd}$, in which a FIFO channel $\bar{A}_{sd} \bar{B}_{sd}$ is used for the control flow (see Fig. 14 as an example). If a data item (message) $d$ is used as a parameter in the guard condition $g_1$ and $g_2$, and it is transmitted through node $A_{sd}$ in $R$, then we can move the source channel end of the FIFO1 channel $\bar{A}_{sd} \bar{B}_{sd}$ in $R$ from node $A_{sd}$ to node $A_{1}$ to get the data item $d$ related to the guard conditions, so that it can be used in the alternative choice. If only one guard condition is satisfied, then the corresponding Reo circuit ($R_{sd_1}$ or $R_{sd_2}$) will be activated. If both conditions are satisfied, the token will flow through the exclusive router, then a non-deterministic choice will be made at the exclusive router, and one of the two subcircuits will be activated by the corresponding filter and synchronous drain. If neither condition is satisfied, the token will go through the filter with pattern $\neg g_1 \land \neg g_2$ from $\bar{A}_{sd}$ to $\bar{B}_{sd}$ directly.

For $SD = \text{opt}(sd)$, the Reo circuit $R_{SD}$ is obtained by combining $R_{sd}$ and a FIFO1 channel with an exclusive router that chooses to either activate the behavior of the operand $sd$ or skip the fragment while nothing happens (Fig. 16).

For $SD = \text{par}(sd_1, sd_2)$, the Reo circuit is obtained by combining $R_{sd_1}$ and $R_{sd_2}$ with a replicator, which represents a parallel activation of both operands, where the internal FIFO channels in $R_{sd_1}$ and $R_{sd_2}$ (those connected to $A_{sd_1}$ and $B_{sd_1}$ inside the boxes, which are not drawn in the picture) ensure that in the combination, the events in the two branches can be interleaved.

In parallel composition, if there exists some common participant $p_i$ in $sd_1$ and $sd_2$, then $R_{sd_1}$ and $R_{sd_2}$ should also have shared nodes $C_i$ and $D_i$, which are obtained by merging the nodes with the same name in the two Reo circuits (as shown in Fig. 18(a)). For the source node $C_i$, if there is some message $m$ sent by $p_i$ in both $sd_1$ and $sd_2$ (as shown in Fig. 18(c)), then in the resulting Reo circuit $R_{SD}$, the filters will be replaced by one filter and one exclusive router, as shown in Fig. 18(b). For all the filters with source end $C_i$ whose pattern $P$ does not appear in another operand, they will be kept the same in the resulting circuit $R_{SD}$.

For $SD = \text{strict}(sd_1, sd_2)$, the Reo circuit is obtained by combining $R_{sd_1}$ and $R_{sd_2}$ as in Fig. 19, where the FIFO channels in $R_{sd_1}$ and $R_{sd_2}$ ensure that in the combined connector none of the events in $sd_2$ can happen before the communication in $sd_1$ has finished.

The case for weak sequencing $\text{seq}(sd_1, sd_2)$ is more complex, because the definition of the $\text{seq}$ operator depends on whether the operands share some lifelines. If a common lifeline $p_i$ exists in both $sd_1$ and $sd_2$, then all the event occurrences on $p_i$ in $sd_1$ must happen before those on $p_i$ in $sd_2$. However, for the event occurrences on different lifelines in the two operands, they may occur in any order. If the operands involve disjoint sets of participants, the weak sequencing reduces to
a parallel merge, as shown in Fig. 17. Otherwise, suppose they share a lifeline \( p_i \), and the sequencers in the circuits for \( sd1 \) and \( sd2 \) for \( p_i \) have \( n_1 \) and \( n_2 \) nodes, respectively. Then we can add two more nodes \( A_w \) and \( B_w \) to the two Reo circuits \( R_{sd1} \) and \( R_{sd2} \), respectively, and synchronize them with the nodes of the sequencers inside the two Reo circuits such that they correspond to the last event on lifeline \( p_i \) in \( R_{sd1} \) and the first event on \( p_i \) in \( R_{sd2} \). The two nodes \( A_w \) and \( B_w \) are ordered by adding a FIFO1 channel and a SynchDrain between them, as shown in Fig. 20.

Note that the resulting Reo circuit in Fig. 20 for weak sequencing can be optimized by replacing the two sequencers in \( R_{sd1} \) and \( R_{sd2} \) corresponding to the same lifeline \( p_i \) with one sequencer with \( n_1 + n_2 \) nodes, where the first \( n_1 \) nodes are synchronized with the \( n_1 \) nodes of the sequencer for \( p_i \) in \( R_{sd1} \) using synchronous drains, and the last \( n_2 \) nodes are similarly synchronized with the \( n_2 \) nodes of the sequencer for \( p_i \) in \( R_{sd2} \). The two sequencers \( \text{Sequencer}_{sd1} \) and \( \text{Sequencer}_{sd2} \) together with the synchronous drains connecting them can then be removed in the resulting Reo circuit \( R_{SD} \) since the ordering information is now kept by the new sequencer, as shown in Fig. 21.

For \( SD = \textbf{loop} (sd) \), the Reo circuit is obtained from \( R_{sd} \) as in Fig. 22. The connector \( gEXR \) is a variant of the exclusive router in Fig. 10(b), where we replace the lossy synchronous channels by two filters with the patterns \( g \) and \( \neg g \) respectively, where \( g \) is the guard condition of the loop. If \( g \) is satisfied, then the loop iterates, otherwise, it stops.
Example 2. To illustrate our approach, we consider the sequence diagram for the on-line banking example in Fig. 4. The generated Reo circuit is given in Fig. 23. Note that for simplicity, we do not give the Reo circuit for the whole scenario. Instead, we show the structure of the connector for the whole scenario in Fig. 23(a) and give one subcircuit \( R_{\text{Login}} \) in Fig. 23(c), which corresponds to the basic sequence diagram Login. Fig. 23(b) shows the internal structure of the guarded exclusive router \( g\text{EXR} \) used in Fig. 23(a). Details of the other building blocks (subcircuits) in Fig. 23(a) are similar to Fig. 23(c), and can be easily obtained by our synthesis approach.

4.4. One step further

Our construction can be easily extended to treat timing constraints in UML SDs. Since the semantics of UML on such messages is ambiguous and can have different meanings, we shall not be exhaustive here, but rather suggest a possible step in this direction, and investigate the approach in our future work. As an example, we consider a message \( m[0..t] \) which states that the message \( m \) is constrained to last between 0 and \( t \) time units. If the receiver of the message is not ready to accept it in \( t \) time units, the message can either be lost or be stored in some queue, waiting for the receiver to process it. We assume that the message with a time constraint \( m[0..t] \) can always be successfully transmitted to the receiver side and waits to be processed. For such an interpretation, we just need to connect the nodes \( A_i \) and \( B_j \) which are internal nodes of the synthesized circuit under construction, to the nodes for the sender and the receiver of the message, respectively via a \( P \)-producer (where \( P \) is the singleton data set \{expire\}), a synchronous drain and a timer channel with early expiration.\(^5\)

\(^5\) See \cite{bib:3} for more details of timer channels.
Such a timer channel allows the timer to produce its timeout signal through its sink end and reset itself when it consumes a special “expire” value through its source [3]. As an example, we replace the message \(m_1\) in Fig. 8(a) by \(m_1\{0..10\}\), and show the resulting Reo circuit in Fig. 24.

5. Correctness

To provide a soundness proof of our approach, we use the framework of coalgebraic bisimulations. For this purpose, we provide a coalgebraic interpretation for the Reo circuit of a sequence diagram and then establish a homomorphism from the coalgebra semantics of a sequence diagram to the coalgebra assigned to its Reo circuit. The bisimulation relation between the coalgebras for sequence diagrams composed by interaction operators and the corresponding Reo circuits can be obtained from the bisimulations between the component coalgebras. Recall that the denotational semantics of a sequence diagram can be viewed as a coalgebra \((C, (\epsilon, \overline{\cdot}))\) where \(C\) is the set of possible configurations.

Let \(sd = (I, \text{Loc}^*, \text{Mes}, \text{Loc}_{\text{in}}, \text{loc}, E, \preceq)\) be a sequence diagram that contains no interaction operators. \(R_{sd}\) is the Reo circuit synthesized from \(sd\). The semantics of \(R_{sd}\), denoted by \([\llbracket R_{sd} \rrbracket]\), can also be given as a coalgebra, where the state space of \([\llbracket R_{sd} \rrbracket]\) is decided by the buffers of the FIFO channels. The FIFO channels in the synthesized Reo circuits can be separated into two categories: channels for the control structure and channels for the communication behavior. Taking Fig. 24 as an example, the channels \(A_{sd}B_{sd}, B_{sd}B_{sd}, A_{sd}B_{sd}\) and the FIFO channels in the sequencers are for the control structure, while the channels \(A_1B_1\) and \(A_2B_2\) are for the communication behavior. While defining the coalgebra \([\llbracket R_{sd} \rrbracket]\), we consider the actual values of data elements only in the case of channels that contribute to the communication behavior, i.e., the messages. Since the actual data values in the buffers of the channels that comprise its control structure cannot be observed through the ports of the synthesized connector circuit, we abstract away from these data values and consider only the empty or non-empty status of such FIFO channels as the indicator of a state. Then the behavior preservation for the synthesized circuit can be witnessed by the bisimulation relation between the two coalgebras [31]. Note that here the coalgebraic interpretation for Reo circuits is different than the coalgebraic semantics defined in [6], where the semantics is expressed in terms of relations on infinite Timed Data Streams, i.e., relations on final coalgebras. Our approach uses one coalgebra, instead of a relation on multiple coalgebras, for the semantics of a Reo circuit, and this coalgebra is an abstraction of the constraint automaton that captures the operational semantics of the Reo circuit [9].

In the sequence diagram \(sd\), for every lifeline \(p_i\) corresponding to the instance \(i \in I\), suppose \(\text{card}(\text{loc}(i)) = n\), then according to the construction in Section 4.1, the sequencer on the \(p_i\) side also has \(n\) external nodes and \(n\) FIFO channels inside. Let \(A_1^i, \ldots, A_n^i\) be the nodes linked to the external nodes of the sequencer by synchronous drains, and denote the buffers of the FIFO channels in the sequencer by \(buf_1^i, \ldots, buf_n^i\), respectively. Suppose \(\text{loc}(i) = \{l_1^i, \ldots, l_n^i\}\), and \(\text{next}(l_j^i) = l_{j+1}^i\) for \(j = 1, \ldots, n - 1\). According to the construction, if there is a synchronous message \(msg\) from \(l_j^i\) to \(l_m^i\), then the nodes \(A_j^i\) and \(A_m^i\) are linked by a synchronous channel, \(A_j^i\) is linked to \(C_i\) by a filter over \(\text{msg}\), and \(A_m^i\) to \(D_i\) by a synchronous channel. When \(buf_j^i\) and \(buf_m^i\) are filled with the tokens, \(\langle l_j^i \cdot l_m^i, \text{msg}\rangle\) and \(\langle l_m^i \cdot l_j^i, \text{msg}\rangle\) become active and the communication can happen. On the other hand, if there is an asynchronous message \(msg\) from \(l_j^i\) to \(l_m^i\), then the nodes \(A_j^i\) and \(A_m^i\) are linked by a FIFO channel, with the buffer denoted by \(buf_{j,m}^{i,k}\). The buffer is filled with \(msg\) after the message is sent out via \(A_j^i\) and becomes empty again after the message is consumed via \(A_m^i\).

Let \(Buf\) be the set of all the buffers inside \(R_{sd}\), then

\[
\llbracket R_{sd} \rrbracket = (U, (\rho, \beta), u_0)
\]

where \(U = 2^{buf}, \rho : U \to \wp(\Sigma_{in}), \beta : U \times \Sigma \to U, u_0 = \{buf_1^i | i \in I\}\). For a state \(u \in U\) and a buffer \(buf \in u, \text{elem}(buf)\) returns the element in \(buf\). Thus the function \(\rho\) is defined as

\[
\rho(u) = \{ \langle l_j^i \leftarrow l_m^i, \text{msg}\rangle | bufk_{j,m}^{i,k} \in u \land \text{elem}(buf_{j,m}^{i,k}) = \text{msg} \} \cup \bigcup_{buf_j^i \in u} \{ e | \pi(e) = l_j^i \land \text{type}(e) \neq \cdot \}
\]
In an arbitrary state \( u \), if \( \text{buf}_{j}^{i}, \text{buf}_{j \rightarrow m}^{k} \in u \), and \( A_{j}' \) and \( A_{m}' \) are linked by a synchronous channel (assume it is from \( A_{j}' \) to \( A_{m}' \)), then the circuit is ready to both accept a synchronous message \( \text{msg} \) from \( C_{i} \) (the source node linked to \( A_{j}' \)) and consume it via \( D_{c} \), and the tokens in the buffers \( \text{buf}_{j}^{i}, \text{buf}_{j \rightarrow m}^{k} \) will move to the next ones. So,

\[
\beta(u, (\langle \ell_{j}^{i} \rightarrow \ell_{m}^{k}, \text{msg} \rangle, \langle \ell_{m}^{k} \rightarrow \ell_{j}^{i}, \text{msg} \rangle)) = u \setminus \{ \text{buf}_{j}^{i}, \text{buf}_{j \rightarrow m}^{k} \} \cup \{ \text{buf}_{j+1}^{i}, \text{buf}_{j \rightarrow m+1}^{k} \}
\]

Similarly, if \( A_{j}' \) and \( A_{m}' \) are linked by an asynchronous channel (assume it is a FIFO channel from \( A_{j}' \) to \( A_{m}' \)), then when the buffer \( \text{buf}_{j \rightarrow m}^{i,k} \) is empty, and \( \text{buf}_{j}^{i} \in u \), the circuit in state \( u \) is ready to accept an asynchronous message \( \text{msg} \) from \( C_{i} \) and put it into the buffer \( \text{buf}_{j \rightarrow m}^{i,k} \). So,

\[
\beta(u, (\ell_{j}^{i} \rightarrow \ell_{m}^{k}, \text{msg})) = u \setminus \{ \text{buf}_{j}^{i} \} \cup \{ \text{buf}_{j+1}^{i}, \text{buf}_{j \rightarrow m}^{i,k} \mid \text{elem}(\text{buf}_{j \rightarrow m}^{i,k}) = \text{msg} \}
\]

When \( \text{elem}(\text{buf}_{j \rightarrow m}^{i,k}) = \text{msg} \), and the buffer \( \text{buf}_{j \rightarrow m}^{i,k} \in u \), the circuit in state \( u \) is ready to consume \( \text{msg} \) via \( D_{c} \).

\[
\beta(u, (\ell_{m}^{k} \leftarrow \ell_{j}^{i}, \text{msg})) = u \setminus \{ \text{buf}_{j}^{i}, \text{buf}_{j \rightarrow m}^{i,k} \} \cup \{ \text{buf}_{m}^{k} \}
\]

For a lost message \( (\ell_{j}^{i} \rightarrow \bullet, \text{msg}) \), the circuit is ready to consume the message when \( \text{buf}_{j}^{i} \in u \), and the token in \( \text{buf}_{j}^{i} \) is moved to \( \text{buf}_{j+1}^{i} \).

\[
\beta(u, (\ell_{j}^{i} \rightarrow \bullet, \text{msg})) = u \setminus \{ \text{buf}_{j}^{i} \} \cup \{ \text{buf}_{j+1}^{i} \}
\]

For a found message \( (\ell_{j}^{i} \leftarrow \bullet, \text{msg}) \), let \( \text{buf}_{f} \) and \( \text{buf}_{f} \) be the two buffers in the circuit \( \text{Found} \) as in Fig. 12. Initially, let \( \text{buf}_{f} \in u \). The circuit is ready to take the message when \( \text{buf}_{f} \in u \). After the event occurs, the token in \( \text{buf}_{f} \) is moved to \( \text{buf}_{f+1}^{i} \), and the message in \( \text{buf}_{f} \) is moved to \( \text{buf}_{f}^{i} \) and then enters \( \text{buf}_{f} \) again. Since the flow of message in \( \text{buf}_{f} \) and \( \text{buf}_{f} \) is in the black-box, we can assume that the buffer \( \text{buf}_{f} \) is in \( u \) after \( (\ell_{j}^{i} \leftarrow \bullet, \text{msg}) \).

\[
\beta(u, (\ell_{m}^{k} \leftarrow \ell_{j}^{i}, \text{msg})) = u \setminus \{ \text{buf}_{j}^{i} \} \cup \{ \text{buf}_{j+1}^{i} \}
\]

**Proposition 1.** \( [[sd]] \sim [[R_{sd}]] \).

**Proof.** We prove bisimulation equivalence of the co-algebras of \( sd \) and its Reo circuit \( R_{sd} \) by establishing a homomorphism \( h : C \rightarrow U \). For a configuration \( c = \bigcap_{i \in I, 0 \leq j < \text{loc}(i)} (\ell_{j}^{i}) \), let

\[
h(c) = \{ \text{buf}_{j}^{i} \mid (\ell_{j}^{i} \in c) \} \cup \{ \text{buf}_{j \rightarrow m}^{i,k} \mid (\exists e = (\ell_{m}^{k} \leftarrow \ell_{j}^{i}, \text{elem}(\text{buf}_{j \rightarrow m}^{i,k})) \in \epsilon(c) ) \}
\]

All we have to prove is that \( h \) is a co-algebra homomorphism, i.e., that

\[
h \cdot \alpha(c, e) = \beta(h(c), e)
\]

\[
\epsilon(c) = \rho(h(c))
\]

We first give the proof for Eq. (4). Let \( c = \bigcap_{i \in I, 0 \leq j < \text{loc}(i)} (\ell_{j}^{i}) \), for \( e = (\ell_{j}^{i} \rightarrow \ell_{m}^{k}, \text{msg}), (\ell_{m}^{k} \rightarrow \ell_{j}^{i}, \text{msg}) \),

\[
h \cdot \alpha(c, e) = h(c, (\ell_{j}^{i+1/j}, \ell_{m+1}^{k}), (\ell_{m+1}^{k}))
\]

\[
= \{ \text{buf}_{j+1}^{i} \mid (\ell_{j}^{i+1/j} \in c) \} \cup \{ \text{buf}_{m+1}^{k} \mid (\exists e = (\ell_{m}^{k} \leftarrow \ell_{j}^{i+1/j}, \text{elem}(\text{buf}_{m+1}^{k})) \in \epsilon(c) ) \}
\]

\[
= \{ \text{buf}_{j+1}^{i} \mid (\ell_{j}^{i+1/j} \in c) \} \cup \{ \text{buf}_{m+1}^{k} \mid (\exists e = (\ell_{m}^{k} \leftarrow \ell_{j}^{i+1/j}, \text{elem}(\text{buf}_{m+1}^{k})) \in \epsilon(c) ) \}
\]

\[
= h(c) \cdot (\ell_{j}^{i+1/j}, \ell_{m+1}^{k})
\]

for \( e = (\ell_{j}^{i} \rightarrow \ell_{m}^{k}, \text{msg}) \).

\[
h \cdot \alpha(c, e) = \{ \text{buf}_{j+1}^{i} \mid (\ell_{j}^{i+1/j} \in c) \} \cup \{ \text{buf}_{j \rightarrow m+1}^{i,k} \mid (\exists e = (\ell_{m}^{k} \leftarrow \ell_{j}^{i+1/j}, \text{elem}(\text{buf}_{j \rightarrow m+1}^{i,k})) \in \epsilon(c) ) \}
\]

\[
= \{ \text{buf}_{j+1}^{i} \mid (\ell_{j}^{i+1/j} \in c) \} \cup \{ \text{buf}_{j+1}^{i} \cup \{ \text{buf}_{j \rightarrow m+1}^{i,k} \mid \text{elem}(\text{buf}_{j \rightarrow m+1}^{i,k}) = \text{msg} \} \cup \{ \text{buf}_{m+1}^{k} \mid (\exists e = (\ell_{m}^{k} \leftarrow \ell_{j}^{i+1/j}, \text{elem}(\text{buf}_{m+1}^{k})) \in \epsilon(c) ) \}
\]

\[
= h(c) \cup \{ \text{buf}_{j+1}^{i} \cup \{ \text{buf}_{j \rightarrow m+1}^{i,k} \mid \text{elem}(\text{buf}_{j \rightarrow m+1}^{i,k}) = \text{msg} \}
\]

\[
= \beta(h(c), e)
\]
for \( e = (l_j ^f \leftarrow l_m ^k, \text{msg}) \),

\[
\begin{align*}
    h \cdot \alpha(c, e) &= \{ \text{buf}^f_j \mid l_j ^f \in c \cap (l_{j+1} ^f / l_j ^f) \} \cup \{ \text{buf}^k_{m,j} \mid \exists (l_j ^f \leftarrow l_m ^k, \text{elem}(\text{buf}_{m,j} ^k)) \in e(\alpha(c, e)) \} \\
    &= \{ \text{buf}^f_j \mid l_j ^f \in c \} \cup \{ \text{buf}^f_j \mid \exists (l_j ^f \leftarrow l_m ^k, \text{elem}(\text{buf}_{m,j} ^k)) \in e(c) \} \\
    &= \{ \text{buf}^f_j \mid l_j ^f \in c \} \cup \{ \text{buf}^f_j \mid \exists (l_j ^f \leftarrow l_m ^k, \text{elem}(\text{buf}_{m,j} ^k)) \in e(c) \} \setminus \{ \text{buf}^k_{m,j} \} \\
    &= h(c) \setminus \{ \text{buf}^f_j \} \cup \{ \text{buf}^k_{m,j} \} \\
    &= \beta(h(c), e)
\end{align*}
\]

The corresponding results for lost and found messages and co-region location can be similarly obtained.

We now provide the proof for Eq. (5) using the induction principle. As the basis of induction, we regard the initial configuration \(c_0\). Since \( u = h(c_0)\) does not contain an element of the form \(\text{buf}^k_{m,j}\), we have:

\[
\rho(h(c_0)) = \bigcup_{\text{buf}^f_j \in h(c_0)} \{ e \mid \pi(e) = l_j ^f \land \text{type}(e) \neq \rightarrow \}
\]

\[
= \bigcup_{l_j ^f \in c_0} \{ e \mid \pi(e) = l_j ^f \land \text{type}(e) \neq \rightarrow \}
\]

\[
= \{ e \mid \pi(e) \in c_0 \land \text{type}(e) \neq \rightarrow \}
\]

\[
= \{ e \mid \pi(e) \in \text{Loc}_m \land \text{type}(e) \neq \rightarrow \}
\]

\[
= \epsilon(c_0)
\]

As the induction step, we assume that \(\rho(h(c)) = \epsilon(c)\) and show that for any \(e\), \(\rho(h \cdot \alpha(c, e)) = \epsilon(\alpha(c, e))\).

We first consider \(e = (l_j ^f \rightarrow l_m ^k, \text{msg}), (l_m ^k \leftarrow l_j ^f, \text{msg})\), and therefore

\[
\rho(h \cdot \alpha(c, e)) = \{ (l_j ^f \leftarrow l_m ^k, \text{msg}) \mid \text{buf}^k_{m,j} \in h \cdot \alpha(c, e) \land \text{elem}(\text{buf}^k_{m,j}) = \text{msg} \} \cup
\]

\[
\bigcup_{\text{buf}^f_j \in h \cdot \alpha(c, e)} \{ e' \mid \pi(e') = l_j ^f \land \text{type}(e') \neq \rightarrow \}
\]

\[
= \{ (l_j ^f \leftarrow l_m ^k, \text{msg}) \mid \text{buf}^k_{m,j} = \beta(h(c), e) \land \text{elem}(\text{buf}^k_{m,j}) = \text{msg} \} \cup
\]

\[
\bigcup_{\text{buf}^f_j \in \beta(h(c), e)} \{ e' \mid \pi(e') = l_j ^f \land \text{type}(e') \neq \rightarrow \}
\]

\[
= \{ (l_j ^f \leftarrow l_m ^k, \text{msg}) \mid \text{buf}^k_{m,j} \in h(c) \land \text{elem}(\text{buf}^k_{m,j}) = \text{msg} \} \cup
\]

\[
\left( \bigcup_{\text{buf}^f_j \in h(c)} \{ e' \mid \pi(e') = l_j ^f \land \text{type}(e') \neq \rightarrow \} \right) \setminus \{ e' \mid \pi(e') = l_j ^f \lor \pi(e') = l_m ^k \land \text{type}(e') \neq \rightarrow \}
\]

\[
= \rho(h(c)) \setminus \{ e' \mid \pi(e') = l_j ^f \lor \pi(e') = l_m ^k \land \text{type}(e') \neq \rightarrow \} \cup
\]

\[
\{ e' \mid \pi(e') = l_j ^f \lor \pi(e') = l_m ^k \land \text{type}(e') \neq \rightarrow \} \cup
\]

\[
\{ e' \mid \pi(e') = l_j ^f \land \text{type}(e') \neq \rightarrow \}
\]

\[
= \epsilon(c) \setminus \{ (l_j ^f \rightarrow l_m ^k, \text{msg}), (l_m ^k \leftarrow l_j ^f, \text{msg}) \} \cup
\]

\[
\{ e' \mid \pi(e') = l_j ^f \lor \pi(e') = l_m ^k \land \text{type}(e') \neq \rightarrow \} \cup
\]

\[
= \epsilon(\alpha(c, e))
\]
The proof for lost and found messages and co-region can be similarly obtained. □
We obtain the equivalence result for arbitrary sequence diagrams by an induction on the structure of the sequence diagrams as in the following theorem. There is a slight change in the proof style used for Theorem 1 and Proposition 1. For Proposition 1, the bisimulation between each sequence diagram and its translation is shown by means of a homomorphism between their corresponding coalgebras. In the proof of Theorem 1, however, we construct the concrete bisimulation relation directly instead of using the graph of a homomorphism.

Note that the composed \(R_{sd}\) has the general structure as in Fig. 13, which has three FIFO channels besides the ones in the sequencers and for the message passing between the external ports. Let \(\text{buf}_{\text{sd}, \text{sd}}, \text{buf}_{\text{sd}, \text{ab}}, \text{buf}_{\text{ab}}\) be the corresponding buffers in these channels. Then the semantics of \(R_{sd}\) can be defined as

\[
\llbracket R_{sd} \rrbracket = (U, (\rho', \beta'), u_0)
\]

where \(U = 2^{\text{buf}_{\text{sd}, \text{sd}}, \text{buf}_{\text{sd}, \text{ab}}, \text{buf}_{\text{ab}}}\), \(\rho' : U \rightarrow \mathcal{P}(\Sigma_m)^n, \beta' : U \times \Sigma \rightarrow U, u_0 = \{\text{buf}_{i} | i \in I\} \cup \{\text{buf}_{\text{sd}}\}\). In any state \(u \in U\), there is at most one element from \(\{\text{buf}_{\text{sd}}, \text{buf}_{\text{sd}, \text{ab}}, \text{buf}_{\text{ab}}\}\) in \(u\). State \(u_F = \{\text{buf}_{i} | i \in I\} \cup \{\text{buf}_{\text{ab}}\}\) is called the final state. Thus the functions \(\rho'\) and \(\beta'\) are defined as

\[
\rho'(u) = \begin{cases} 
\text{buf}_{\text{sd}} \in u & \Rightarrow \rho(u \setminus \{\text{buf}_{\text{sd}}\}) \\
\text{buf}_{\text{sd}} \in u & \Rightarrow \rho(u \setminus \{\text{buf}_{\text{sd}}\}) \\
\text{buf}_{\text{sd}} \in u & \Rightarrow \rho(u \setminus \{\text{buf}_{\text{sd}}\}) \\
\end{cases}
\]

\[
\beta'(u, e) = \begin{cases} 
u = u_0 & \Rightarrow \beta(u_0 \setminus \{\text{buf}_{\text{sd}}\}, e) \cup \{\text{buf}_{\text{ab}}\} \\
\text{buf}_{\text{ab}} \in u \land \rho'(u \setminus \{\text{buf}_{\text{ab}}\}, e) \neq \emptyset & \Rightarrow \beta(u \setminus \{\text{buf}_{\text{ab}}\}, e) \cup \{\text{buf}_{\text{ab}}\} \\
\text{buf}_{\text{ab}} \in u \land \rho'(u \setminus \{\text{buf}_{\text{ab}}\}, e) = \emptyset & \Rightarrow \beta(u \setminus \{\text{buf}_{\text{ab}}\}, e) \cup \{\text{buf}_{\text{ab}}\} \\
\end{cases}
\]

Furthermore, the function \(h' : C \rightarrow U\) is defined as

\[
h'(c) = \begin{cases} 
c = c_0 & \Rightarrow h(c) \cup \{\text{buf}_{\text{sd}}\} \\
c = c_F & \Rightarrow h(c) \cup \{\text{buf}_{\text{ab}}\} \\
\text{otherwise} & \Rightarrow h(c) \cup \{\text{buf}_{\text{ab}}\} \\
\end{cases}
\]

**Theorem 1.** For sequence diagrams \(sd_1\) and \(sd_2\), if \(\llbracket sd_1 \rrbracket \sim \llbracket sd_2 \rrbracket\), then

\[
\begin{align*}
\llbracket \text{strict}(sd_1, sd_2) \rrbracket &\sim \llbracket \text{strict}(sd_1, sd_2) \rrbracket \\
\llbracket \text{par}(sd_1, sd_2) \rrbracket &\sim \llbracket \text{par}(sd_1, sd_2) \rrbracket \\
\llbracket \text{opt}(sd_1) \rrbracket &\sim \llbracket \text{opt}(sd_1) \rrbracket \\
\llbracket \text{alt}(g_1 : sd_1, g_2 : sd_2) \rrbracket &\sim \llbracket \text{alt}(g_1 : sd_1, g_2 : sd_2) \rrbracket \\
\llbracket \text{seq}(sd_1, sd_2) \rrbracket &\sim \llbracket \text{seq}(sd_1, sd_2) \rrbracket \\
\llbracket \text{loop}(g : sd_1) \rrbracket &\sim \llbracket \text{loop}(g : sd_1) \rrbracket \\
\end{align*}
\]

**Proof.** Assume that \(\llbracket R_{sd} \rrbracket = (U, (\rho', \beta'), u_0)\), and \(\llbracket sd \rrbracket = (C_i, (\epsilon_i, \alpha_i), e_i^0)\) for \(i = 1, 2\). Note that for weak sequential composition we consider only the case for \(I_1 \cap I_2 \neq \emptyset\), the case for \(I_1 \cap I_2 = \emptyset\) can be obtained by the proof for parallel composition.

**strict:** \(\text{strict}(sd_1, sd_2)\) is given by the coalgebra \((U, (\tilde{\rho}, \tilde{\beta}), u_0)\), where

\[
\begin{align*}
U &= \{u \cup \{\text{buf}_{i}^1 | i_2 \in I_2\} | u \in U_1 \} \cup \{u \cup \{\text{buf}_{i}^2 | i_1 \in I_1\} | u \in U_2\} \cup \{u_0, u_F\} \\
u_0 &= \{\text{buf}_{i}^1 | i_1 \in I_1\} \cup \{\text{buf}_{i}^2 | i_2 \in I_2\} \cup \{\text{buf}_{\text{sd}, \text{sd}}\} \\
u_F &= \{\text{buf}_{i}^1 | i_1 \in I_1\} \cup \{\text{buf}_{i}^2 | i_2 \in I_2\} \cup \{\text{buf}_{\text{sd}, \text{ab}}\} \\
\tilde{\rho}(u) &= \begin{cases} 
u = u_0 \lor u = u_F & \Rightarrow \emptyset \\
u = u_1 \cup \{\text{buf}_{i}^2 | i_2 \in I_2\} \land u_1 \in U_1 & \Rightarrow \rho_1'(u_1) \\
u = u_2 \cup \{\text{buf}_{i}^1 | i_1 \in I_1\} \land u_2 \in U_2 & \Rightarrow \rho_2'(u_2) \\
\end{cases} \\
\tilde{\beta}(u, e) &= \begin{cases} 
u = u_1 \cup \{\text{buf}_{i}^2 | i_2 \in I_2\} \land u_1 \in U_1 \land \beta_1'(u_1, e) \neq \emptyset & \Rightarrow \beta_1'(u_1, e) \cup \{\text{buf}_{i}^2 | i_2 \in I_2\} \\
u = u_1 \cup \{\text{buf}_{i}^2 | i_2 \in I_2\} \land u_1 \in U_1 \land \beta_1'(u_1, e) = \emptyset & \Rightarrow \{\text{buf}_{i}^1 | i_1 \in I_1\} \cup \{\text{buf}_{i}^2 | i_2 \in I_2\} \cup \{\text{buf}_{\text{sd}, \text{ab}}\} \\
u = u_2 \cup \{\text{buf}_{i}^1 | i_1 \in I_1\} \land u_2 \in U_2 & \Rightarrow \beta_2'(u_2, e) \cup \{\text{buf}_{i}^1 | i_1 \in I_1\} \\
\end{cases}
\end{align*}
\]
Because of the structure of $R_{\text{strict}}(sd_1, sd_2)$, when the circuit is in state $u_0$, since the sink end of $A_{\text{SD}}$ is connected to the source end of $A_{\text{sd}}$, by a synchronous channel, and $\tilde{\rho}(u_0) = \emptyset$, the circuit can take an internal transition by transmitting the token in the buffer $buf_{\text{sd}}$ to the buffer $buf_{\text{sa}}$, which makes the state change to $buf_{\text{sa}} | i_1 \in I_1 \cup \{buf_{\text{sd}} | i_2 \in I_2\}$. Similarly, when the circuit is in state $buf_{\text{sa}} | i_1 \in I_1 \cup \{buf_{\text{sd}} | i_2 \in I_2\} \cup \{buf_{\text{sa}} | i_1 \in I_1 \cup \{buf_{\text{sd}} | i_2 \in I_2\} \cup \{buf_{\text{sa}} | i_1 \in I_1 \cup \{buf_{\text{sd}} | i_2 \in I_2\}$. And when there is a token in $buf_{\text{sa}}$ it moves to $buf_{\text{sa}}$ and the state changes to $u_F$.

According to the construction, we have the coalgebra homomorphisms $h_i : [sd_i] \rightarrow [R_{\text{sd}}]$. From the definition of $\text{strict}(sd_1, sd_2)$, we know that the initial configuration of $sd_1$ is merged with the initial configuration of $sd_2$ in the strict sequential composition. Let $B_1 = \{buf_{\text{sd}} | i_1 \in I_1\}$ and $B_2 = \{buf_{\text{sd}} | i_2 \in I_2\}$, and the relation $\approx \subseteq C \times U$ be defined as follows:

$$\approx = \{(c_0, B_1 \cup B_2 \cup \text{buf}_{\text{sd}}), (c_0, B_1 \cup B_2 \cup \text{buf}_{\text{sa}}), (c_1, B_1 \cup B_2 \cup \text{buf}_{\text{sd}}), (c_1, B_1 \cup B_2 \cup \text{buf}_{\text{sa}})\} \cup$$

$$\{(c, h_1(c) \cup \text{buf}_{\text{sa}}, B_2) \mid c \in C_1 \setminus \{c_0, c_1\}\} \cup$$

$$\{(c_0, B_1 \cup B_2 \cup \text{buf}_{\text{sa}}, B_2), (c_0, B_1 \cup B_2 \cup \text{buf}_{\text{sa}}), (c_1, B_1 \cup B_2 \cup \text{buf}_{\text{sa}})\} \cup$$

$$\{(c, h_2(c) \cup \text{buf}_{\text{sa}}, B_1) \mid c \in C_2 \setminus \{c_0, c_1\}\} \cup$$

$$\{(c_0, B_1 \cup B_2 \cup \text{buf}_{\text{sa}}, B_1), (c_0, B_1 \cup B_2 \cup \text{buf}_{\text{sa}}), (c_1, B_1 \cup B_2 \cup \text{buf}_{\text{sa}})\} \cup$$

From the above discussion, we know that there is an internal transition from $B_1 \cup B_2 \cup \text{buf}_{\text{sd}}$ to $B_1 \cup B_2 \cup \text{buf}_{\text{sa}}$. For $u = B_1 \cup B_2 \cup \text{buf}_{\text{sa}}$,

$$\tilde{\rho}(u) = \rho'(B_1 \cup \text{buf}_{\text{sa}})$$

and

$$\tilde{\rho}(u) = \rho'(B_1 \cup \text{buf}_{\text{sa}})$$

According to the definition of $[\text{strict}(sd_1, sd_2)]$, we have

$$(\text{strict}(\alpha_1, \alpha_2)(c_0, e), \tilde{\rho}(u, e)) = (\alpha_1(c_0, e), h_1(\alpha_1(c_0, e)) \cup \{buf_{\text{sa}} \cup \text{buf}_{\text{sa}}\}) \cup B_2$$

$$\{\text{let } c = \alpha_1(c_0, e) \in \approx\}$$

Let $u = h_1(c) \cup \{buf_{\text{sa}} \cup \text{buf}_{\text{sa}}\} \cup B_2$ for $c \in C_1 \setminus \{c_0, c_1\}$, and $e \in \tilde{\rho}(c)$, if $\rho'(\beta'(h_1(c) \cup \{buf_{\text{sa}} \cup \text{buf}_{\text{sa}}\}, e)) \neq \emptyset$,

$$\tilde{\rho}(u, e) = \beta'(h_1(c) \cup \{buf_{\text{sa}} \cup \text{buf}_{\text{sa}}\}, e) \cup B_2$$

and

$$\tilde{\rho}(u, e) = \beta'(h_1(c) \cup \{buf_{\text{sa}} \cup \text{buf}_{\text{sa}}\}, e) \cup B_2$$

and

$$\tilde{\rho}(u) = \rho'(h_1(c) \cup \{buf_{\text{sa}} \cup \text{buf}_{\text{sa}}\})$$

$$\rho_1(h_1(c))$$

$$\epsilon_1(c)$$

$$\text{strict}(\epsilon_1, \epsilon_2)(c)$$
if \( \rho'(\beta'((h_1(c) \cup \{buf_{\bar{A}_{sd}, \bar{B}_{sd}}\}, e)) = \emptyset, \ \hat{\beta}(u, e) = B_1 \cup B_2 \cup \{buf_{\bar{B}_{sd}, \bar{B}_{sd}}\}, \) according to the definition of \( \rho' \) and \( \beta' \), we have
\( \rho_1(\beta_1(h_1(c), e)) = \emptyset. \) Therefore, \( \epsilon_1(\alpha_1(c), e)) = \emptyset \) and strict\((\alpha_1, \alpha_2)(c, e) = c_0^1. \) In both cases, we have
\[
(\text{strict}(\alpha_1, \alpha_2)(c, e), \hat{\beta}(u, e)) \in \approx
\]
Similarly, we can prove that for any \( (c, u) \in \approx, (\text{strict}(\alpha_1, \alpha_2)(c, e), \hat{\beta}(u, e)) \in \approx \) and \( \hat{\rho}(u) = \text{strict}(\epsilon_1, \epsilon_2)(c). \) So \( \approx \) is a bisimulation relation and \( (c_0, u_0) \in \approx. \) Therefore,
\[
\llbracket \text{strict}(sd_1, sd_2) \rrbracket \sim \ llbracket \text{strict}(sd_1, sd_2) \rrbracket
\]
par: \( \text{par}(sd_1, sd_2) \)
The semantics of \( R_{\text{par}(sd_1, sd_2)} \) is given by the coalgebra \( (U, \langle \hat{\beta}, \hat{\rho} \rangle, u_0) \), where
\[
U = \{u_1 \cup u_2 | u_1 \in U_1 \cup u_2 \in U_2 \} \cup \{u_0, u_f\}
\]
\[
u_0 = \{buf_{1}^1 | i_1 \in I_1\} \cup \{buf_{2}^2 | i_2 \in I_2\} \cup \{buf_{\bar{A}_{sd}, \bar{B}_{sd}}\}
\]
\[
u_f = \{buf_{1}^1 | i_1 \in I_1\} \cup \{buf_{2}^2 | i_2 \in I_2\} \cup \{buf_{\bar{B}_{sd}, \bar{B}_{sd}}\}
\]
\[
\hat{\rho}(u) = \begin{cases} u = u_0 \lor u = u_f \Rightarrow \emptyset \\ u = u_1 \cup u_2 \land u_1 \in U_1 \land u_2 \in U_2 \Rightarrow \rho_1(u_1) \cup \rho_2(u_2) \\ u = u_1 \cup u_2 \land u_2 \in U_2 \land e \in \rho_1(u_1) \Rightarrow \beta_1(u_1, e) \cup u_2 \\ u = u_1 \cup u_2 \land u_1 \in U_1 \land u_2 \in U_2 \land e \in \rho_2(u_2) \Rightarrow \beta_2(u_2, e) \cup u_1 \\
\end{cases}
\]
\[
\hat{\beta}(u, e) = \begin{cases} u = u_1 \cup u_2 \land u_1 \in U_1 \land u_2 \in U_2 \land e \in \rho_1(u_1) \Rightarrow \beta_1(u_1, e) \cup u_2 \\ u = u_1 \cup u_2 \land u_2 \in U_2 \land e \in \rho_2(u_2) \Rightarrow \beta_2(u_2, e) \cup u_1 \\
\end{cases}
\]
There is an internal transition from \( u_0 \) to \( u_1^1 \cup u_2^2 \), in which the token in \( buf_{\bar{A}_{sd}, \bar{B}_{sd}} \) is copied by the replicator in node \( \bar{A}_{sd}. \) The tokens move to \( buf_{\bar{B}_{sd}, \bar{B}_{sd}} \) and \( buf_{\bar{A}_{sd}, \bar{B}_{sd}} \) respectively, and the state changes to \( u_1^0 \cup u_2^3. \) Similarly, in state \( u_1^3 \cup u_2^2, \) the two tokens in \( buf_{\bar{B}_{sd}, \bar{B}_{sd}} \) and \( buf_{\bar{A}_{sd}, \bar{B}_{sd}} \) are synchronously consumed in the syncdrain \( B_{sd}B_{sd} \). One of the tokens is non-deterministically chosen by the merger in node \( \bar{B}_{sd} \) and moved to the buffer \( buf_{\bar{B}_{sd}} \), while the other one is lost in the lossy channel. Let the relation \( \approx \subseteq C \times U \) be defined as follows:
\[
\approx = \{(c_0, u_0), (c_0, u_0^1 \cup u_0^2)\} \cup \{(c, h_1^1(c_1) \cup h_2^2(c_2))\} \cup \{(c_1, u_f), (c_1, B_1 \cup B_2 \cup \{buf_{\bar{B}_{sd}, \bar{B}_{sd}}\}) \cup \{buf_{\bar{A}_{sd}, \bar{B}_{sd}}\} \}
\]
We have
\[
\hat{\rho}(u_0^1 \cup u_0^2) = \rho_1^1(u_0^1) \cup \rho_2^2(u_0^2)
\]
\[
= \rho_1^1(buf_{1}^1 | i_1 \in I_1) \cup \rho_2^2(buf_{2}^2 | i_2 \in I_2)
\]
\[
= \rho_1(h_1^1(c_0^1) \cup \rho_2(h_2^2(c_0^2))
\]
\[
= \epsilon_1(c_0^1) \cup \epsilon_2(c_0^2)
\]
\[
= \text{par}(\epsilon_1, \epsilon_2)(c_0)
\]
and if \( e \in \rho_1^1(u_0^1) \),
\[
\hat{\beta}(u_0^1 \cup u_0^2, e) = \beta_1^1(u_0^1, e) \cup u_0^2
\]
\[
= \beta_1^1(buf_{1}^1 | i_1 \in I_1, e) \cup \{buf_{\bar{B}_{sd}, \bar{B}_{sd}}\} \cup u_0^2
\]
\[
= \beta_1^1(buf_{1}^1 | i_1 \in I_1, e) \cup \{buf_{\bar{B}_{sd}, \bar{B}_{sd}}\} \cup \{buf_{1}^2 | i_2 \in I_2\} \cup \{buf_{\bar{A}_{sd}, \bar{B}_{sd}}\}
\]
since \( \rho_1(u_0^1) = \rho_1(buf_{1}^1 | i_1 \in I_1) = \epsilon_1(c_0^1) \), we have
\[
\text{par}(\alpha_1, \alpha_2)(c_0, e) = (\alpha_1(\pi_1c_0, e), \pi_2c_0)
\]
Furthermore,
\[
h_1^1(\alpha_1(\pi_1c_0, e)) = \beta_1^1(buf_{1}^1 | i_1 \in I_1, e)
\]
\[
h_2^2(\pi_2c_0) = buf_{1}^2 \mid i_2 \in I_2
\]
So \( \text{par}(\alpha_1, \alpha_2)(c_0, e), \hat{\beta}(u_0^1 \cup u_0^2, e) \in \approx. \) For the case \( e \in \rho_2^2(u_0^2) \), the result can be similarly obtained. Analogously, we can obtain that for any \( (c, u) \in \approx, \text{par}(\alpha_1, \alpha_2)(c, e), \hat{\beta}(u, e) \in \approx \) and \( \hat{\rho}(u) = \text{par}(\epsilon_1, \epsilon_2)(c). \) So \( \approx \) is a bisimulation relation, and
\[
\llbracket \text{par}(sd_1, sd_2) \rrbracket \sim \ llbracket \text{par}(sd_1, sd_2) \rrbracket
\]
Option: \( \text{opt}(sd_1) \)
The semantics of \( \text{opt}(sd_1) \) is given by the coalgebra \((U, \widehat{\rho}, \widehat{\beta}), u_0)\), where

\[
\begin{align*}
U &= U_1 \cup \{u_0, u_F\} \\
u_0 &= \{buf_1^{i_1} \mid i_1 \in I_1\} \cup \{buf_{\text{in}_F}^{\text{sd}}\} \\
u_F &= \{buf_1^{i_1} \mid i_1 \in I_1\} \cup \{buf_{\text{sd}}\} \\
\widehat{\rho}(u) &= \begin{cases} u = u_0 \Rightarrow \{\text{skip}\} \\
u = u_f \Rightarrow \emptyset \\
u \in U_1 \Rightarrow \rho_i'(u) \end{cases} \\
\widehat{\beta}(u, e) &= \begin{cases} u \in U_1 \Rightarrow \beta_i'(u, e) \\
u = u_0 \land e = \text{skip} \Rightarrow u_F \end{cases}
\end{align*}
\]

When the circuit is in state \( u_0 \), it has a nondeterministic choice between the behavior of \( R_{\text{sd}_1} \) and an empty behavior (denoted by \( \text{skip} \)). In other words, there are two internal transitions with \( u_0 \) as source state, and the target states are \( u_F \) and \( \{buf_1^{i_1} \mid i_1 \in I_1\} \cup \{buf_{\text{in}_F}^{\text{sd}}\} \) respectively. Furthermore, when the circuit is in state \( \{buf_1^{i_1} \mid i_1 \in I_1\} \cup \{buf_{\text{sd}}\} \), there is no active event and it can move to state \( \{buf_1^{i_1} \mid i_1 \in I_1\} \cup \{buf_{\text{sd}}\} \) by an internal transition. Define the relation \( \approx \subseteq C \times U \) as:

\[
\begin{align*}
\approx &= \{(c_0, u_0), (c_0, u_0')\} \cup \{(c, h_1(c) \cup \{buf_{\text{in}_F}^{\text{sd}}\}) \mid c \in C \setminus \{c_0, c_F\}\} \cup \\
&\quad \{(c_F, \{buf_1^{i_1} \mid i_1 \in I_1\} \cup \{buf_{\text{sd}}\})\}, (c_F, u_F)\}
\end{align*}
\]

According to the semantics, we have

\[
\begin{align*}
\widehat{\rho}(u_0) &= \beta_1'(u_0) \\
&= \rho_1(\{buf_1^{i_1} \mid i_1 \in I_1\}) \\
&= \epsilon(c_0)
\end{align*}
\]

and

\[
\widehat{\rho}(u_0) = \{\text{skip}\}
\]

so,

\[
\widehat{\rho}(u_0) \cup \widehat{\beta}(u_0) = \text{opt}(\epsilon_1)(c_0)
\]

Additionally,

\[
\widehat{\beta}(u_0, \text{skip}) = \{buf_1^{i_1} \mid i_1 \in I_1\} \cup \{buf_{\text{sd}}\}
\]

and for \( e \in \Sigma_1 \), if \( \rho_1(\beta_1(\{buf_1^{i_1} \mid i_1 \in I_1\}, e)) \neq \emptyset \), then

\[
\begin{align*}
\widehat{\beta}(u_0, e) &= \beta_1'(u_0, e) \\
&= \beta_1(\{buf_1^{i_1} \mid i_1 \in I_1\}, e) \cup \{buf_{\text{sd}}\} \\
&= h_1(\alpha_1(c_0, e)) \cup \{buf_{\text{sd}}\}
\end{align*}
\]

So we have

\[
(\text{opt}(\alpha_1)(c_0, e), \widehat{\beta}(u_0, e)) \in \approx
\]

and

\[
(\text{opt}(\alpha_1)(c_0, \text{skip}), \widehat{\beta}(u_0, \text{skip})) \in \approx
\]

Similarly, we can obtain that for any \((c, u) \in \approx\),

\[
(\text{opt}(\alpha_1)(c, e), \widehat{\beta}(u, e)) \in \approx
\]

\[
\widehat{\rho}(u) = \text{opt}(\epsilon_1)(c)
\]
So $\approx$ is a bisimulation relation, and 

$$\llbracket \text{opt}(sd_1) \rrbracket \sim \llbracket \text{opt}(sd_1) \rrbracket$$

**Choice: $\text{alt}(g_1 : sd_1, g_2 : sd_2)$**

The semantics of $R_{\text{alt}(g_1 : sd_1, g_2 : sd_2)}$ is given by the coalgebra $(U, (\tilde{\rho}, \tilde{\beta}), u_0)$, where 

$$U = \{u \cup \{buf_1^{i_1} | i_2 \in I_2\} | u \in U_1\} \cup \{u \cup \{buf_1^{i_1} | i_1 \in I_1\} | u \in U_2\} \cup \{u_0, u_F\}$$

$$u_0 = \{buf_1^{i_1} | i_1 \in I_1\} \cup \{buf_1^{i_2} | i_2 \in I_2\} \cup \{buf_{sd_1}^{\text{alt}}\}$$

$$u_F = \{buf_1^{i_1} | i_1 \in I_1\} \cup \{buf_1^{i_2} | i_2 \in I_2\} \cup \{buf_{sd_2}^{\text{alt}}\}$$

$$\tilde{\rho}(u) = \begin{cases} 
\{ u = u_0 \} & \Rightarrow \{ \text{skip} \} \\
\{ u = u_F \} & \Rightarrow \emptyset \\
\{ u = u_1 \cup \{buf_1^{i_2} | i_2 \in I_2\} \land u_1 \in U_1 \} & \Rightarrow \rho_1'(u_1) \\
\{ u = u_2 \cup \{buf_1^{i_1} | i_1 \in I_1\} \land u_2 \in U_2 \} & \Rightarrow \rho_2'(u_2) 
\end{cases}$$

$$\tilde{\beta}(u, e) = \begin{cases} 
\{ u = u_0 \land e = \text{skip} \} & \Rightarrow u_F \\
\{ u = u_1 \cup \{buf_1^{i_2} | i_2 \in I_2\} \land u_1 \in U_1 \} & \Rightarrow \beta_1'(u_1, e) \cup \{buf_1^{i_2} | i_2 \in I_2\} \\
\{ u = u_2 \cup \{buf_1^{i_1} | i_1 \in I_1\} \land u_2 \in U_2 \} & \Rightarrow \beta_2'(u_2, e) \cup \{buf_1^{i_1} | i_1 \in I_1\} 
\end{cases}$$

When the circuit is in state $u_0$, and the token in $A_{sd_1}A_{sd_2}$ satisfies $g_1$, the token can be transmitted through the filter $A_{sd_1}A_{sd_2}$ by an internal transition, which changes the state from $u_0$ to $\{buf_1^{i_1} | i_1 \in I_1\} \cup \{buf_1^{i_2} | i_2 \in I_2\} \cup \{buf_{sd_1}^{\text{alt}}\}$. This activates the circuit $R_{sd_1}$. If $R_{sd_1}$ arrives at its final state, there is an internal transition in which the token in $buf_{sd_1}^{\text{alt}}$ moves to $buf_{sd_2}^{\text{alt}}$, and the state changes to $u_F$. Define the relation $\simeq \subseteq C \times U$ as follows:

$$\simeq = \{ (c_0, u_0) \} \cup 
\{ (c, h_1(c) \cup \{buf_1^{i_1} | i_2 \in I_2\}) | c \in C_1 \setminus \{c_0, c_1\} \} \cup 
\{ (c, h_2(c) \cup \{buf_1^{i_1} | i_1 \in I_1\}) | c \in C_2 \setminus \{c_0, c_2\} \} \cup 
\{ (c_f, \{buf_1^{i_1}, i_1 \in I_1\} \cup \{buf_1^{i_2}, i_2 \in I_2\} \cup \{buf_{sd_1}^{\text{alt}}\}) \} \cup 
\{ (c_r, \{buf_1^{i_1}, i_1 \in I_1\} \cup \{buf_1^{i_2}, i_2 \in I_2\} \cup \{buf_{sd_2}^{\text{alt}}\}) \} \cup 
\{ (c_f, u_F) \}$$

and the relation $\approx$ as 

$$\approx = \{ c_0 \models g_1 \land g_2 \Rightarrow \simeq \cup \{(c_0, u_1 \cup \{buf_1^{i_2} | i_2 \in I_2\}), (c_0, u_2 \cup \{buf_1^{i_1} | i_1 \in I_1\}) \} \\
| c_0 \models g_1 \land \neg g_2 \Rightarrow \simeq \cup \{(c_0, u_0 \cup \{buf_1^{i_2} | i_2 \in I_2\}) \} \\
| c_0 \models g_2 \land \neg g_1 \Rightarrow \simeq \cup \{(c_0, u_0 \cup \{buf_1^{i_1} | i_1 \in I_1\}) \}$$

If neither of the guard conditions is satisfied by the token in $buf_{sd_1}^{\text{alt}}$, then skip happens, which moves the token in $buf_{sd_1}^{\text{alt}}$ to $buf_{sd_2}^{\text{alt}}$, and the state changes from $u_0$ to $u_F$. In this case, we have

$$(\text{alt}(\alpha_1, \alpha_2)(c_0, \text{skip}), \tilde{\beta}(u_0, \text{skip})) = (c_r, u_F) \in \approx$$

If $c_0 \vdash g_1 \land \neg g_2$, the token in $buf_{sd_1}^{\text{alt}}$ moves to $buf_{sd_1}^{\text{alt}}$, the state of the circuit changes to $u = u_0 \cup \{buf_1^{i_2} | i_2 \in I_2\}$. According to the definition, we have

$$\tilde{\rho}(u) = \rho_1'(u_0)$$

$$= \rho_1(\{buf_1^{i_1} | i_1 \in I_1\})$$

$$= \Sigma_0$$

$$= \text{alt}(\epsilon_1, \epsilon_2)(c_0)$$

and if $e \in \Sigma_0^*$,

$$\tilde{\beta}(u_0 \cup \{buf_1^{i_2} | i_2 \in I_2\}, e) = \beta_1'(u_0, e) \cup \{buf_1^{i_1} | i_1 \in I_1\}$$

$$= \beta_1(\{buf_1^{i_1} | i_1 \in I_1\}, e) \cup \{buf_{sd_1}^{\text{alt}}\} \cup \{buf_1^{i_2} | i_2 \in I_2\}$$
There is an internal transition from \( c_0 \) to \( u_0 \), which is copied by the replicator in node \( \overline{A}_{sd} \). The tokens move to \( buf_{A_{sd}A_{sd}} \) and \( buf_{A_{sd}A_{sd}} \), respectively, and the state changes to \( u_0 \). Similarly, in state \( u_0 \), the two tokens in \( buf_{A_{sd}A_{sd}} \) and \( buf_{A_{sd}A_{sd}} \) are consumed synchronously in the syncrain \( B_{sd}B_{sd} \). One of the tokens is nondeterministically chosen by the merger in node \( B_{sd}B_{sd} \) and moved to the buffer \( buf_{B_{sd}B_{sd}} \) while the other one is lost in the lossy channel. Define the relation \( \approx \subseteq C \times U \) as follows:

\[
\approx = \{(c_0, u_0), (c_0, u_0' \cup u_0')\} \cup \\
\{(c, h_1'((\pi_1, \pi_2) c) \cup h_2'((\pi_1' c, \pi_3) c)) | \pi_1 c \in loc_1(i) \cup \{\pi_1 c\}\} \cup \\
\{(c, h_1'((\pi_1' c, \pi_2) c) \cup h_2'((\pi_1, \pi_3) c)) | \pi_1 c \in loc_2(i)\} \cup \\
\{(c, h_1'((\pi_1, \pi_2) c) \cup h_2'((\pi_1' c, \pi_3) c) \cup \{buf_{A_{sd}A_{sd}}\}) | \pi_1 c = \pi_1 c\} \cup \\
\{(c_f, u_f), (c_f, u_f' \cup u_f')\}
\]
We have

\[ \hat{\rho}(u_0^I \cup u_0^B) = \rho'_1(u_0^I) \cup (\rho'_2(u_0^B) \setminus \{ e | \pi(e) = \ell_1^I \}) \]
\[ = \rho_1(\{ buf^I_{i_1} | i_1 \in I_1 \}) \cup (\rho_2(\{ buf^B_{i_2} | i_2 \in I_2 \}) \setminus \{ e | \pi(e) = \ell_1^I \}) \]
\[ = \rho_1(h_1(c_0^I)) \cup (\rho_2(h_2(c_0^B)) \setminus \{ e | \pi(e) = \ell_1^I \}) \]
\[ = \epsilon_1(c_0^I) \cup (\epsilon_2(c_0^B) \setminus \{ e | \pi(e) = \ell_1^I \}) \]
\[ = \epsilon_1(c_0^I) \cup \epsilon_2(c_0^B) \setminus \{ e | \pi(e) \in loc(i) \land \pi(e) \neq \pi_1 c_0^I \} \]
\[ = \epsilon_1(c_0^I) \cup \epsilon_2(c_0^B) \setminus \{ e | \pi(e) \in loc(i) \land \pi(e) \neq \pi_1 c_0^I \} \]
\[ = \text{seq}(\epsilon_1, \epsilon_2)(c_0^I) \]

and if \( e \in \rho'_2(u_0^B) \),

\[ \hat{\beta}(u_0^I \cup u_0^B, e) = \beta'_1(u_0^I, e) \cup u_0^B \]
\[ = \beta_1(\{ buf^I_{i_1} | i_1 \in I_1 \}, e) \cup \{ buf_{A_{ad} \tilde{A}_{ad}} \} \cup u_0^B \]
\[ = h'_1(\alpha_1(c_0^I, e)) \cup u_0^B \]
\[ = h'_1(\alpha_1(c_0^I, e)) \cup h'_2(\pi_1, \pi_2 c_0^B) \]
\[ = h'_1(\alpha_1(c_0^I, e)) \cup h'_2(\pi_1 c_0^I, \pi_3 c_0^B) \]
\[ = h'_1(\pi_1, \pi_2) \text{seq} (\alpha_1, \alpha_2)(c_0, e) \cup h'_2(\pi_1 c_0^I, \pi_2 \text{seq} (\alpha_1, \alpha_2)(c_0, e)) \]
\[ = h'_1(\pi_1, \pi_2) \text{seq} (\alpha_1, \alpha_2)(c_0, e) \cup h'_2(\pi_1 c_0^I, \pi_2 \text{seq} (\alpha_1, \alpha_2)(c_0, e)) \]

So (\text{seq}(\alpha_1, \alpha_2)(c_0, e), \hat{\beta}(u_0^I \cup u_0^B, e)) \in \approx. For the case \( e \in \rho'_2(u_0^B) \), the result can be similarly derived. Analogously, we can obtain that for any \((c, u) \in \approx, (\text{seq}(\alpha_1, \alpha_2)(c, e), \hat{\beta}(u, e)) \in \approx \) and \( \hat{\rho}(u) = \text{seq}(\epsilon_1, \epsilon_2)(c) \). So \( \approx \) is a bisimulation relation, and

\[ \| \text{seq}(sd_1, sd_2) \| \sim \| R_{\text{seq}(sd_1, sd_2)} \| \]

**Loop:** \( \text{loop}(g : sd_1) \)
The semantics of \( R_{\text{loop}(g : sd_1)} \) is given by the coalgebra \((U, (\hat{\rho}, \hat{\beta}), u_0)\), where

\[ U = U_1 \cup \{ u_0, u_f \} \]
\[ u_0 = \{ buf^I_{i_1} | i_1 \in I_1 \} \cup \{ buf_{A_{ad} \tilde{A}_{ad}} \} \]
\[ u_f = \{ buf^I_{i_1} | i_1 \in I_1 \} \cup \{ buf_{A_{ad} \tilde{A}_{ad}} \} \]
\[ \hat{\rho}(u) = \begin{cases} 
  u = u_0 & \Rightarrow \{ \text{skip} \} \\
  u = u_f & \Rightarrow \emptyset \\
  u \in U_1 & \Rightarrow \rho'_1(u) \\
  \text{otherwise} & \Rightarrow \beta'_1(u, e) 
\end{cases} \]
\[ \hat{\beta}(u, e) = \begin{cases} 
  u = u_0 \land e = \text{skip} & \Rightarrow u_f \\
  \text{otherwise} & \Rightarrow \beta'_1(u, e) 
\end{cases} \]

When the circuit is in state \( u_0 \) and the token in \( A_{ad} \tilde{A}_{ad} \) satisfies \( g \), there is an internal transition from state \( u_0 \) to \( \{ buf^I_{i_1} | i_1 \in I_1 \} \cup \{ buf_{A_{ad} \tilde{A}_{ad}} \} \), in which the token in \( buf_{A_{ad} \tilde{A}_{ad}} \) is transmitted to \( buf_{A_{ad} \tilde{A}_{ad}} \), and the circuit moves to state \( u_1^I \). When the circuit is in state \( u_0 \) and the token in \( A_{ad} \tilde{A}_{ad} \) does not satisfy \( g \), the token moves to \( buf_{A_{ad} \tilde{A}_{ad}} \) by \text{skip}, and the state changes to \( u_f \). When the circuit is in \( u_f \) and the token in \( buf_{A_{ad} \tilde{A}_{ad}} \) satisfies \( g \), then it is transmitted via \text{gEXR} and the synchronous channels to \( buf_{A_{ad} \tilde{A}_{ad}} \) by an internal transition, and the circuit moves to state \( u_1^I \) again. Otherwise, the token moves to \( buf_{A_{ad} \tilde{A}_{ad}} \) by an internal transition, and the state changes to \( u_f \). Define the relation \( \approx \subseteq C \times U \) as follows:

\[ \approx = \{ (c_0, u_0), (c_0, u_0^I) \} \cup \\
(\{ c, h_1(c) \cup \{ buf_{A_{ad} \tilde{A}_{ad}} \} \} \setminus \{ c_0, c_1^I \}) \cup \\
\{ (c_1^I, buf_{i_1}^I | i_1 \in I_1 \} \cup \{ buf_{A_{ad} \tilde{A}_{ad}} \}) \cup \\
\{ (c_1^I, u_f) \}
\]
According to the definition, we have

\[
\hat{\rho}(u_0^1) \cup \hat{\rho}(u_0) = \rho_i(u_0^1) \cup \{\text{skip}\}
\]

\[
= \rho_i(\{\text{buf}^1_1 | i_1 \in I_1\}) \cup \{\text{skip}\}
\]

\[
= \epsilon_1(c_1^1) \cup \{\text{skip}\}
\]

\[
= \epsilon_1(c_0) \cup \{\text{skip}\}
\]

\[
= \text{loop}(\epsilon_1)(c_0)
\]

If \( e \in \epsilon_1(c_0), \) then \( \hat{\beta}(u_0, e) = \beta_1(\{\text{buf}^1_1 | i_1 \in I_1\}, e) \cup \{\text{buf}^1_{ad}, \text{buf}^1_{sd}\}. \) From \( \text{loop}(\epsilon_1)(c_0, e) = \alpha_1(c^1_0, e) \) (here we assume that \( \text{loop}(\epsilon_1)(c_0, e) \neq \emptyset \)) and \( \beta_1(\{\text{buf}^1_1 | i_1 \in I_1\}, e) = h_1(\alpha_1(c^1_0, e)), \) we have

\[
(\text{loop}(\alpha_1)(c_0, e), \hat{\beta}(u_0, e)) \in \approx
\]

Furthermore, \( (\text{loop}(\alpha_1)(c_0, \text{skip}), \hat{\beta}(u_0, \text{skip})) = (c_F, u_F), \) where \( c_F = c^1_F. \) So

\[
(\text{loop}(\alpha_1)(c_0, \text{skip}), \hat{\beta}(u_0, \text{skip})) \in \approx
\]

Similarly, we can derive that for any \( (c, u) \in \approx, \)

\[
(\text{loop}(\alpha_1)(c, e), \hat{\beta}(u, e)) \in \approx
\]

\[
\hat{\rho}(u) = \text{loop}(\epsilon_1)(c)
\]

So \( \approx \) is a bisimulation relation, and

\[
\llbracket \text{loop} (g : sd_1) \rrbracket \sim \llbracket R_{\text{loop}(g : sd_1)} \rrbracket \quad \square
\]

6. Related work

One closely related work is the synthesis of adapters in component based systems. The authors of [39] propose an approach to modify the interaction mechanisms that are used to glue components together by integrating the interaction protocol into components. However, this approach acts only at the signature level. The work reported in [8,34] goes beyond the signature level and supports protocol transformations in the synthesis process, but the initial coordinator being synthesized behaves only as the “no-op” coordinator, which requires the assembly of new components to enhance its protocol for communication.

Brogi et al. [12] set a formal foundation for the adaptation of heterogeneous components that may present mismatching interaction behavior. Session types are used to cope with heterogeneous descriptions of component interfaces. An adapter can be automatically generated from an adapter specification, which establishes a correspondence between messages in different components. However, the adapter specification in their approach requires a good deal of implementation details such as correspondences among methods (and their parameters) of different components.

In [10], an approach to scheduler synthesis for discrete event physical systems using supervisory control is proposed, where supervisors are defined as processes and the allowable executions of a system are specified as a set of traces. The supervisory controller interacts with the running system and makes it conform to the specification, which is given as a collection of languages that can be intersected to yield a global specification. A supervisor is synthesized to restrict the system’s behavior by synchronizing the events in the system. Our approach goes beyond behavioral restriction, and our synthesized circuits can interact with system components through different communicating mechanisms encoded in the channels.

A number of approaches for the synthesis of state-based models from scenario descriptions have been developed. For example, the authors of [23] present a state-chart synthesis algorithm, but their approach does not support High-Level Message Sequence Charts (HMSC), which provide a composition mechanism very close to UML2.0 SDs. The authors of [36, 37] propose an approach to synthesize LTS models from MSC specifications, where the mechanism for communication among components is synchronous. The authors of [25] use MSCs for service specifications and propose an algorithm for synthesizing component automata from specifications. In [15,16], the problem of synthesizing state machines from LSC models was tackled by defining the notion of consistency of an LSC model. A global system automaton can be constructed and then decomposed. However, this approach suffers from the state explosion problem due to the construction of the global system automaton, which is often huge in size because of the underlying weak partial ordering semantics of LSC. The authors of [32] combine the LSC notation with Z, and propose a synthesis approach for generating distributed finite state designs from the combined specifications.

The authors of [27] propose an interactive algorithm that can be used to generate flat state-charts from UML sequence diagrams. In [22], the authors also provide an interactive algorithm to generate state-charts from multiple scenarios expressed as UML collaborations. In [35], the existing LTS synthesis algorithms are extended to produce Modal Transition Systems from the combination of properties (expressed in temporal logic) and scenarios. An algebraic approach was adopted
in [40] to synthesize state-charts of components from sequence diagrams, but it takes only the operators \texttt{alt}, \texttt{seq} and \texttt{loop} into account, and does not consider any of the other UML2.0 operators on SDs.

Regardless of the scenario notations used (MSC, LSC or UML), all scenario-based synthesis approaches focus only on generating the state-based models for separate components, or a global state machine for the whole system. These approaches differ from ours as (1) we are concerned about the coordination aspects in distributed applications instead of the behavior models for separate individual components, and (2) our synthesized connectors also provide the actual protocols used for communication among components/services in the system, and our components do not need to contain any protocol information. Therefore, changes in the communication protocol caused by system evolution require us to change only the connector implementation, without changing any of the components that are not directly involved in the evolution. Furthermore, the framework of synthesizing Reo circuits from scenario specifications provides a certain flexibility in the synthesis process. When we modify the scenario specification (for example, adding, removing or changing a sequence diagram), part of the previous synthesized Reo circuits can be reused. Since every scenario described by UML SDs captures only a possible system behavior and it is possible to add more scenarios during the development, the specification can be taken as complete at some point, after which we can block whatever is not given in the specification.

7. Conclusion

In this paper we have presented a novel approach for constructing Reo circuits from scenario specifications represented as UML sequence diagrams. This work extends our previous work described in [4] that presents an algorithm for automatically synthesizing Reo circuits from constraint automata specifications, and the results in [7] that show the synthesis of constraint automata from UML sequence diagrams. The method described in this paper allows us to derive a Reo circuit as the implementation of the coordinator for a concurrent system directly from UML scenario specifications, which can have greater structural fidelity compared to the connectors generated by using CA as a bridge, and thus higher reusability.

In [4], we have shown how to synthesize Reo circuits from constraint automata specifications. On the other hand, an algebraic approach for generating constraint automata from scenario specifications has been proposed in [7]. However, like most program-generated code, the synthesis of a Reo circuit from a constraint automaton, as reported in [4], generally yields verbose circuits that do not “look natural” to the human eye. Therefore, generating a Reo circuit from a constraint automaton synthesized from UML2 SDs yields Reo coordinator circuits that may not easily correlate back to their original SD specifications. The merit of synthesizing Reo circuits directly from SDs lies in the greater structural fidelity between the resulting Reo circuits in this approach and their original SD specifications. The new contribution in this paper is that we go one step further and generate Reo circuits directly from scenario specifications represented by UML sequence diagrams. There is substantial benefit in this work which bridges the gap between requirements and implementation of coordination in the development of large-scale, distributed systems. From the presentation of the synthesis approach, we can easily conclude that the size of the generated Reo circuit, i.e., the number of channels in the circuit, is linear to the size of the corresponding sequence diagrams in terms of the number of lifelines and messages.

Among our next steps is the automation of the synthesis approach described in this paper. We already have a set of integrated, visual tools to support coordination of components/services, including graphical editors, animation and simulation tools, code generators, and model checkers [1,11,21,38]. We expect our tool to be useful in model-based development of service oriented applications. Our aim is to aid designers who are interested in complex coordination scenarios by enabling them to use UML SDs as the basis for generating implementations automatically using our synthesis approach. Once the Reo circuit is generated from a scenario specification, we can also apply the existing tools, for example, the Reo model checker [21], to check for containment and equivalence of connectors. It would also be interesting to consider extensions of UML, for example, the UML Profile for Schedulability, Performance and Time (UML-SPT) [29], which can be used to provide appropriate representation of QoS aspects in UML, and their connection with quantitative Reo circuits [5]. Another future direction is to establish a formal consistency result for our translation from UML SDs to Reo circuits and the synthesis algorithm of constraint automata from UML SDs suggested in [7]. The proof obligation will be to show that the constraint automaton of the Reo circuit constructed from a given UML SD is (bisimulation) equivalent to the constraint automaton obtained by the techniques of [7] for the same UML SD. As both approaches are based on structural induction, we may use inductive arguments to establish this consistency result. The investigation on the link between timing constraints in UML and the verification methods for timed constraint automata [3] is in the scope of our future work as well.

Acknowledgements

Comments of the anonymous referees are gratefully acknowledged. We are indebted to Luis Barbosa and Jan Rutten for the fruitful discussions regarding the coalgebraic semantics of UML sequence diagrams, and to all the members of SEN3 for helpful discussions. We are particularly grateful for Erik de Vink’s interest in this work and his help on improving the paper. Parts of this paper have been presented in the Proceedings of FOCLASA’08. We are thankful for the attention and discussion with the participants in FOCLASA’08, and to the anonymous reviewers for their valuable comments. The work reported in this paper is supported by a grant from the GLANCE funding program of the Dutch National Organization for Scientific Research (NWO), through project CooPer (600.643.000.05N12), and the DFG-NWO-project SYANCO.
References


