Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 229 (2009) 21–41
www.elsevier.com/locate/entcs

Synthesis of Reo Circuits from Scenario-based Specifications

Farhad Arbab[1] Sun Meng[2]

CWI, Kruislaan 413, Amsterdam, The Netherlands

Christel Baier[3]

Institute for Theoretical Computer Science, Technische Universität Dresden, Dresden, Germany

Abstract

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" that encode the 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.

Keywords: Connector, Reo circuits, Scenario-based specification, UML, Synthesis

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 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 is verified

[1] Email: Farhad.Arbab@cwi.nl
[2] Corresponding author. Email: M.Sun@cwi.nl
[3] Email: baier@tcs.inf.tu-dresden.de

doi:10.1016/j.entcs.2009.06.027

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 the 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 showing 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) inside 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. Recently, scenario based languages such as UML Sequence Diagrams (SDs) [26], message sequence charts (MSCs) [16,17], and Live Sequence Charts (LSCs) [13], are being widely used to capture 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 [23].

The idea of using scenario descriptions, such as UML SDs, to generate operational models and/or executable code is not new [14,15,21,27]. However, most of the existing work take an endogenous approach for coordination, in which even small changes to a protocol can propagate through and affect large spans of software components, invalidating their previously verified properties. Such invasive modifications are not only undesirable, in many cases they are impractical or even impossible, e.g., when legacy code or third party providers are involved.

In [4] the problem of synthesizing Reo circuits from given automata specifications is discussed. In[6] 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 redundant synchrony in the synthesis process and result in quite 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 simpler and more efficient. Furthermore, the proposed translation from UML SDs to Reo circuits in this paper is compositional and therefore preserves the nature of the interaction inherent in a UML SD, while the translation from constraint automata to Reo circuits cannot recover parallelism.

The semantics of UML 2.0 sequence diagrams has been studied in [24], wherein we use coalgebras for defining the semantics of basic sequence diagrams and formally

define the operators for combining sequence diagrams. On the other hand, the coalgebraic semantics of Reo is also investigated in [7]. Therefore, the correctness of the mapping from UML to Reo in our synthesis approach can, in principle, be judged by comparing their semantics. Due to the lack of space, we will not discuss the formal correctness problem in this paper and leave it as future work.

The rest of this paper is organized as follows: Section 2 contains a brief summary of the main features 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 present related work and compare it with our approach. Finally, Section 6 concludes the paper.

2 Reo

Reo [3] 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 [3,7,9].

Complex connectors in Reo are organized in 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 they interconnect. 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. Figure 1 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 Figure 1). 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*

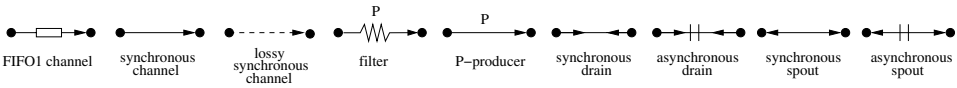


Fig. 1. Some basic channels in Reo

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 accepted by the channel are lost. An asynchronous drain accepts data items through its source ends and loses them, but never simultaneously. *(A)synchronous Spouts* are duals to the drain channels, as they have two sink ends.

Complex connectors are constructed by composing simpler ones via the *join* and *hiding* operations. Channels are joined together in nodes. A node consists of a set of channel ends. The set of channel ends coincident on a node A is disjointly partitioned into the sets $\mathsf{Src}(A)$ and $\mathsf{Snk}(A)$, denoting the sets of source and sink channel ends that coincide on A, respectively. Nodes are categorized into *source*, *sink* and *mixed nodes*, depending on whether all channel ends that coincide on a node are source ends, sink ends or a combination of the two. The hiding operation is used to hide the internal topology of a component connector. The hidden nodes can no longer be accessed or observed from outside. A complex connector has a graphical representation, called a *Reo circuit*, which is a finite graph where the *nodes* are labeled with pair-wise disjoint, non-empty sets of channel ends, and the *edges* represent the connecting channels. The behavior of a Reo circuit is formalized by means of the data-flow at its sink and source nodes. Intuitively, the source nodes of a circuit are analogous to the input ports, and the sink nodes to the output ports of a component, while mixed nodes are its hidden internal details. Components cannot connect to, read from, or write to mixed nodes. Instead, data-flow through mixed nodes is totally specified by the circuits they belong to.

A component can write data items to a source node that it is connected to. The write operation succeeds only if all (source) channel ends coincident on the node accept the data item, in which case the data item is transparently written to every source end coincident on the node. A source node, thus, acts as a replicator. A component can obtain data items, by an input operation, from a sink node that it is connected to. A take operation succeeds only if at least one of the (sink) channel ends coincident on the node offers a suitable data item; if more than one coincident channel end offers suitable data items, one is selected non-deterministically. A sink node, thus, acts as a non-deterministic merger. A mixed node non-deterministically selects and takes a suitable data item offered by one of its coincident sink channel ends and replicates it into all of its coincident source channel ends. Note that a component cannot connect to, take from, or write to mixed nodes. At most one component can be connected to a (source or sink) node at a time. The I/O operations are performed through interface nodes of components which are called ports.

Example 2.1 [Sequencer] Figure 2(a) shows an implementation of a sequencer by

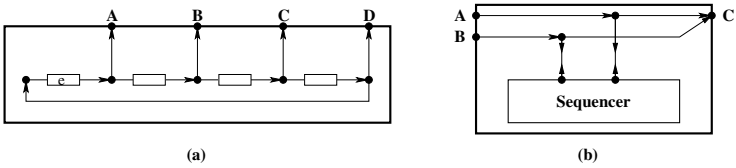


Fig. 2. Sequencer

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.

Figure 2(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 A, followed by the first data item written to B, followed by the second data item written to A, followed by the second data item written to B, 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 the time. Every component has a vertical dashed line called *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 Figure 3 as an example of sequence diagrams which describe the interactions in the login phase of an on-line banking scenario. A UML SD is represented as a rectangular frame labeled by the keyword **sd** followed by the interaction name. 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

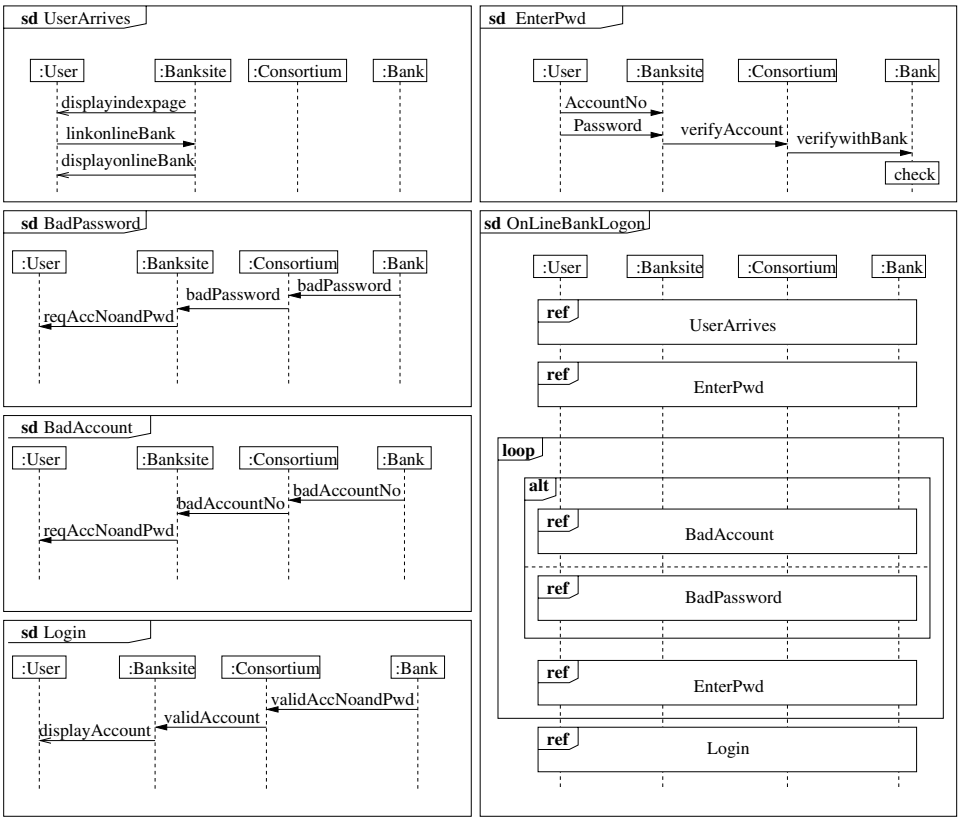


Fig. 3. Sequence diagrams for the on-line banking example

(represented by a filled arrow head). Two special kinds of messages are *lost* and *found* messages, which are represented by a small black circle at the arrow end (starting end, respectively) of 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. For example, the internal behavior or action within the lifelines in SDs (like the *check* action in Figure 3) will not be considered in the synthesis process. Consequently, the synthesized result in our approach reflects only the interaction aspect of a system and is not a global state machine that intertwines both the behavior of components and the interactions among them.

The internal behavior or action in SDs (such as *check* in Figure 3) is like an assumption that the component (*Bank* in Figure 3) has to fulfill. For the example, it means that after receiving the message *verifywithbank*, the bank has to check whether the account number and password fit together. This can be modeled by a constraint automaton for the bank. For the synthesis of the Reo circuit, this CA is irrelevant, but when verification of UML SD via Reo model checkers is considered, the automata for components (like the bank) may be important for proving certain properties using the assume/guarantee paradigm. Therefore, we ignore such internal behavior in this paper because we focus only on the construction of the Reo circuit. However, the automata for the components still can (and will) be used for the analysis of UML SDs.

UML SDs may contain sub-interactions called *interaction fragments* that can be structured and combined using *interaction operators*. There are several possible operators, such as **alt**, **seq**, **loop**, and so on. Depending on the operator used, an interaction fragment consists of one or more operands. For **loop**, the fragment has exactly one operand, while for most other operators there are several operands.

The semantics of an interaction fragment depends on the operator used in its definition, as informally described in the UML superstructure specification [26]. In the following, we give the meaning of some operators according to [26]:

- The operator **alt** designates that the combined fragment represents a choice of behavior alternatives, where at most one of the operand SDs will be chosen. The chosen operand must have an explicit or implicit guard expression that evaluates to true at this point in the interaction.
- The operator **opt** designates that the fragment represents a choice where either the (sole) operand happens or nothing happens.
- The operator **par** designates that the combined fragment represents a parallel merge between the behaviors of its operand SDs. The event occurrences of the different operands can be interleaved in any way as long as the ordering imposed by each operand as such is preserved.
- The operator **seq** designates that the fragment represents a weak sequencing between the behaviors of the operands, i.e., the ordering of event occurrences within each of the operands is maintained in the result, whereas event occurrences on different lifelines in different operands may come in any order. Event occurrences on the same lifeline in different operands are ordered such that an event occurrence of the first operand comes before that of the second operand.
- The operator **strict** designates that the combined SD represents a strict sequencing of the behaviors of the operands, i.e., a strict ordering of the operands on the first level within the combined fragment it defines.
- The **loop** operator specifies an iteration of an interaction. The loop operand will be repeated a number of times. The construct represents a recursive application of the **seq** operator where the loop operand is sequenced after the result of earlier iterations.

There are some more operators given in [26]. For example, the negative operator **neg** designates that the fragment represents traces that are invalid; the **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 **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 are useful, for example, for verifying system properties and test case construction. We can easily handle the cases for **ignore** and **critical**. For **neg**, the situation is a little more complex, since the behavior in the scenario is not permitted. However, it is still possible to deal with **neg** using constraint automata. We can first construct the constraint automaton

corresponding to a negative scenario according to the algorithm in [6], and then build its complement automaton, and finally generate its corresponding Reo circuit according to the approach in [4]. Due to the length limitation, we will not consider these operators in this paper.

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 $L = \{p_1, \cdots, p_n\}$. 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 $C_1, \cdots, C_n$ and sink nodes $D_1, \cdots, D_n$, 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 $\mathcal{N} = \{C_1, \cdots, C_n\} \cup \{D_1, \cdots, D_n\}$ 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_j's are sink nodes. Our goal is to construct a Reo circuit R with source nodes $C_1, \cdots, C_n$ and sink nodes $D_1, \cdots, D_n$, 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 k lifelines $p_1, \cdots, p_k$ 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 of the synthesis approach for basic sequence diagrams is to derive a sequencer for every lifeline p_i in a basic sequence diagram. If there are m events (sending and receiving messages) on the lifeline p_i, then the sequencer corresponding to p_i also has m nodes, the order of which corresponds to the order of events on p_i.

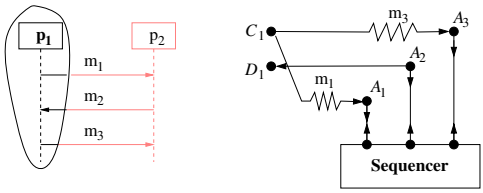


Fig. 4. Introducing sequencers for separate lifelines in a scenario (1)

Figure 4 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_1 side, there is no restriction on the channel A_2D, like the filters, 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 Figure 5. In other words, the type of a message is always guaranteed by the sender and verified by the receiver side.

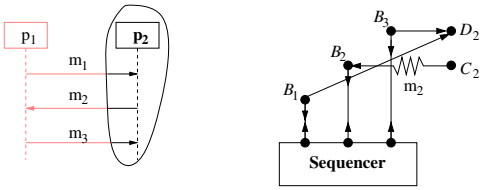


Fig. 5. Introducing sequencers for separate lifelines in a scenario (2)

4.2 Reo Circuits for Basic SDs

After we derive the sequencers for all participants from their respective life lines in isolation, we can connect the nodes $A_i, B_j, \cdots$ pairwise by synchronous or asynchronous Reo channels, according to the message kinds and order, as defined in a basic SD. For example, the Reo circuit on the right-hand-side in Figure 6 is the result of composing the Reo circuits in Figure 4 and 5, according to the basic SD on the left-hand-side of Figure 6, 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.

Note that in this example, there are two synchronous channels A_1B_1 and A_3B_3 for transmitting m_1 and m_3 respectively. For synchronous messages, one synchronous channel is enough for all message transmissions between the two participants, because the messages are ordered. Therefore, we can replace the connector on the right-hand-side of Figure 6 by the one in Figure 7, in which we have one merger and one replicator at nodes M and R, respectively. The filters RB_i ensure that m_1 is received before m_3 by p_2. This approach can be generalized to the case

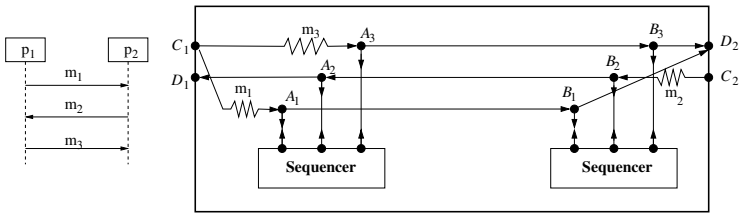


Fig. 6. Reo circuit for a scenario

where we have multiple synchronous messages between two participants in a distributed system: we can implement the connector between them in a distributed manner, and connect its two parts by just one synchronous channel, which provides an optimization of the previous Reo circuit in Figure 6.

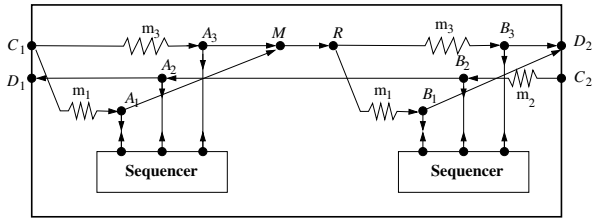


Fig. 7. An alternative connector

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 Figure 3. There are different possibilities for the ordering of events for asynchronous messages, as shown in Figure 8.

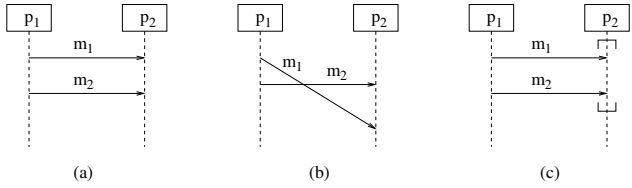


Fig. 8. Asynchronous messages

Since the order of asynchronous message passing may be different, it is not possible to use only one asynchronous channel for all asynchronous communications as for synchronous messages in Figure 7. In Figure 9, we give the Reo circuits for the scenarios in Figures 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 insignificant. Figure 8(c) shows an example of a coregion. In this case, the corresponding Reo circuit is as shown in Figure 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 Figure 10(b).

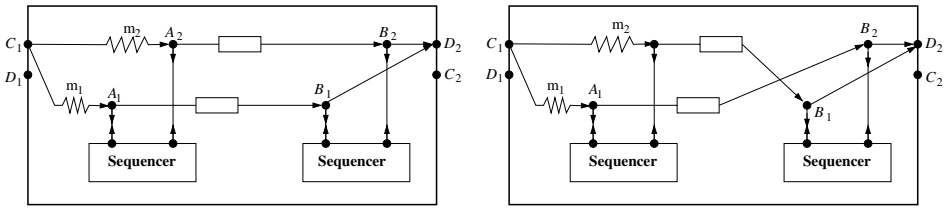


Fig. 9. Connectors for different orders of asynchronous messages

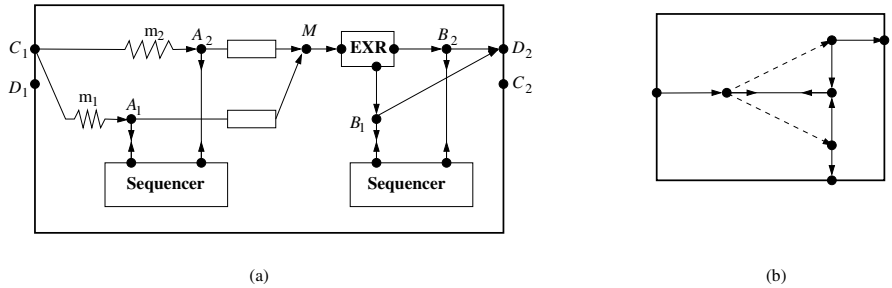


Fig. 10. Connectors for asynchronous messages in coregion

One participant in a scenario can send the same message multiple times. Figure 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 by the sequencer.

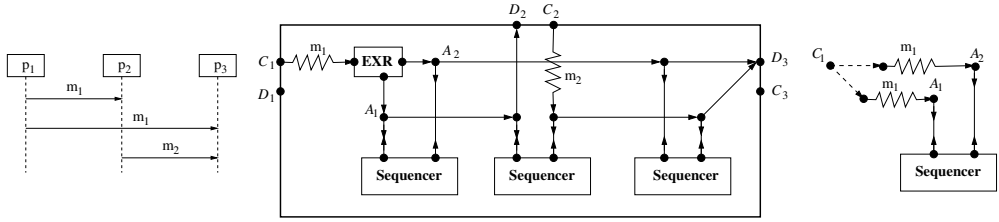


Fig. 11. Same message to different lifelines

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 Figure 12, where the source node B_{lost} can take a message from outside, and lose it in the synchronous drain. Such a component can be integrated in the synthesized connector by connecting the node B_{lost} to the node A_i synchronized with the sequencer for the sender of the lost message.

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

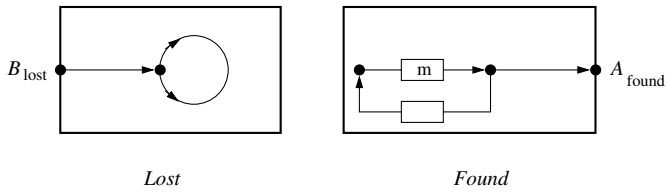


Fig. 12. Reo circuit for lost and found messages

such messages by the *Found* connector in Figure 12 whose found message is m. The sink node A_{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 come to the compositional construction of Reo circuits by combining the Reo circuits of basic SDs. To structure the connectors according to the operators in UML SDs, we use a general structure for the Reo circuits as shown in Figure 13: R_{sd} is the Reo circuit for a basic SD sd, which is obtained as in the previous section. In this construction, two more nodes $\widetilde{A}_{sd}$ and $\widetilde{B}_{sd}$ are added to R_{sd}, which are synchronized with the nodes of the sequencers inside R_{sd}, that correspond to the first send event and the last receive 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 $\overline{A}_{sd}$. As soon as $\overline{A}_{sd}$ 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 $\overline{B}_{sd}$, which puts it into the buffer between $\overline{B}_{sd}$ and B_{sd}. Thus, data-flow at the sink node $\overline{B}_{sd}$ can be viewed as a signal that R_{sd} has terminated. As an example, the generalized Reo circuit for the connector in Figure 7 is shown in Figure 14.

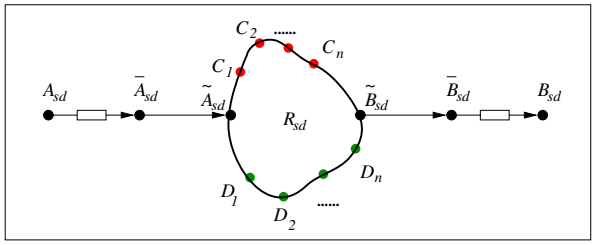


Fig. 13. Reo circuit for basic SDs

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.

For $SD = \mathbf{alt}(g_1 : sd1, g_2 : sd2)$, the Reo circuit R_{SD} is obtained by combining R_{sd1} and R_{sd2} with a replicator connected by two filters where the patterns on the two filters correspond to the guard conditions g_1 and g_2, respectively. The data item d to be transmitted via the channel $A_{SD}\overline{A}_{SD}$ is related to the guard condition that may be obtained from other nodes (i.e., g_1 and g_2 in Figure 15). In this case,

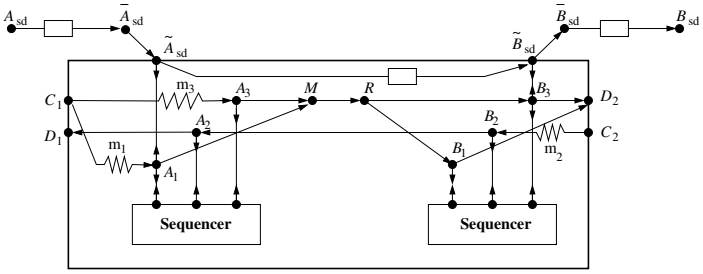


Fig. 14. General structure of the Reo circuit in Figure 7

there is another Reo circuit R connected to A_{SD}, in which a FIFO channel $\widetilde{A}_{sd}\widetilde{B}_{sd}$ is used for the control flow (see Figure 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 transimitted through node A_i in R, then we can move the source channel end of the FIFO1 channel $\widetilde{A}_{sd}\widetilde{B}_{sd}$ in R from node $\widetilde{A}_{sd}$ to node A_i to get the data item d related to the guard conditions, so that it can be used in the alternative choice.

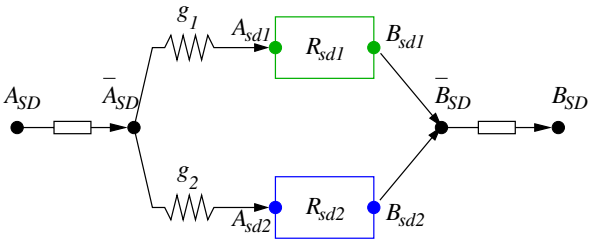


Fig. 15. Alternatives

For $SD = \mathbf{opt}(sd)$, 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.

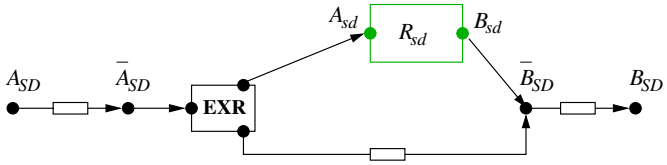


Fig. 16. Option

For $SD = \mathbf{par}(sd1, sd2)$, the Reo circuit is obtained by combining R_{sd1} and R_{sd2} with a replicator, which represents a parallel activation of both operands, where the internal FIFO channels in R_{sd1} and R_{sd2} (those connected to A_{sdi} and B_{sdi} 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 $sd1$ and $sd2$, then R_{sd1} and R_{sd2} 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 Figure 18(a)). For the source node C_i, if there is some message m sent by p_i in both $sd1$ and $sd2$ (as shown in Figure 18(c)), then in the resulting Reo circuit R_{SD}, the filters

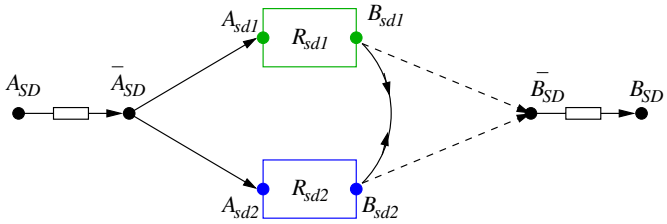
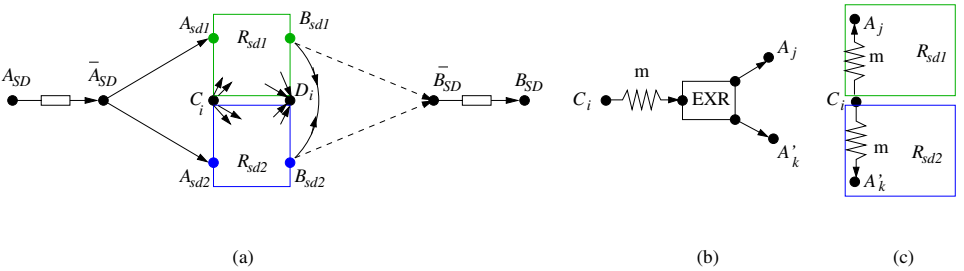


Fig. 17. Parallel composition

will be replaced by one filter and one exclusive router, as shown in Figure 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}.



(a) (b) (c)

Fig. 18. Parallel composition with same participant p_i

For $SD = \mathbf{strict}(sd1, sd2)$, the Reo circuit is obtained by combining R_{sd1} and R_{sd2} as in Figure 19, where the FIFO channels in R_{sd1} and R_{sd2} ensure that in the combined connector none of the events in $sd2$ can happen before the communication in $sd1$ has finished.

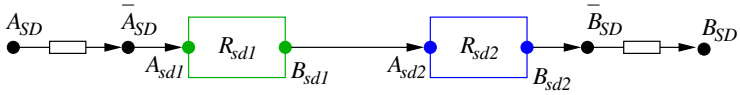


Fig. 19. Strict sequencing

The case for weak sequencing $\mathbf{seq}(sd1, sd2)$ is more complex, because the definition of the **seq** operator depends on whether the operands share some lifelines. If a common lifeline p_i exists in both $sd1$ and $sd2$, then all the event occurrences on p_i in $sd1$ should happen before those on p_i in $sd2$. However, for the event occurrences on different lifelines in the two operands, they may come in any order. If the operands involve disjoint sets of participants, the weak sequencing reduces to a parallel merge, as shown in Figure 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 Figure 20.

Note that the resulting Reo circuit in Figure 20 for weak sequencing can be optimized by replacing the two sequencers in R_{sd1} and R_{sd2} corresponding to the

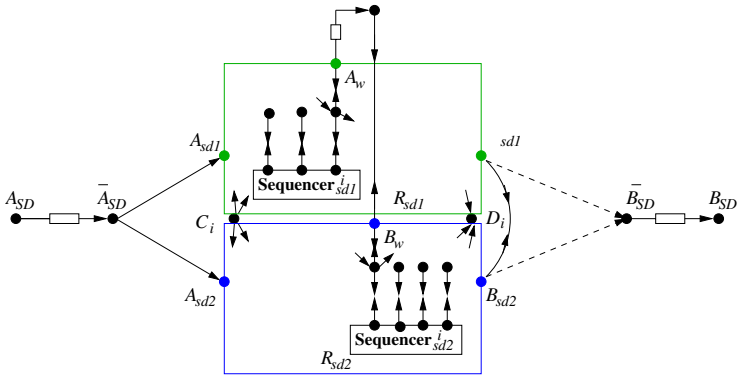


Fig. 20. Weak sequencing

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 $Sequencer^i_{sd1}$ and $Sequencer^i_{sd2}$ together with the synchronous drains connecting them can then be removed in the resulting Reo circuit $R_S D$ since the ordering information is now kept by the new sequencer, as shown in Figure 21.

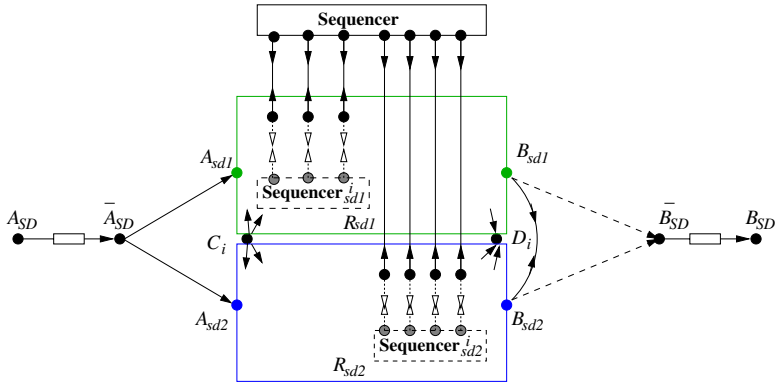


Fig. 21. Weak sequencing optimization

For $SD = \mathbf{loop}(sd)$, the Reo circuit is obtained from R_{sd} as in Figure 22. The connector $gEXR$ is a variant of the exclusive router in Figure 10(b), where we replace the lossy synchronous channels by two filters with pattern g and $\neg g$ respectively, where g is the guard condition of the loop. If g is satisfied, then the loop will iterate, otherwise, it will stop.

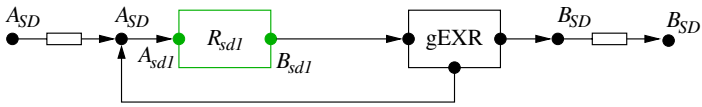


Fig. 22. Loop

5 Related Work

One closely related work is the synthesis of adapters in component based systems. The authors of [32] 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 on the signature level. The work reported in [8,28] 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 heterogenous descriptions of component interfaces. An adaptor can be automatically generated from an adaptor specification, which establishes a correspondence between messages in different components. However, the adaptor 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 synthesis of state-based models from scenario descriptions have been developed. For example, the authors of [20] present a statechart 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 [30,31] propose an approach to synthesize LTS models from MSC specifications, where the mechanism for communication among components is synchronous. The authors of [21] use MSCs for service specifications and propose an algorithm for synthesizing component automata from specifications. In [14,15], 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 [27] combine the LSC notation with Z, and propose a synthesis approach for generating distributed finite state designs from the combined specifications.

The authors of [22] propose an interactive algorithm that can be used to generate flat state-charts from UML sequence diagrams. In [18], the authors also provide an

interactive algorithm to generate state-charts from multiple scenarios expressed as UML collaborations. In [29], 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 [33] to synthesize state-charts of components from sequence diagrams, but it takes only the operators **alt**, **seq** and **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 the 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 only captures a possible system behavior and it is possible to add more scenarios during the development, the specification can be taken as complete up to some point and block what are not given in the specification.

In [4], we have shown how to synthesize Reo circuits from constraint automata specifications. On the other hand, an algebraic approach of generating constraint automata from scenario specifications has been proposed in [6]. 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, to generate 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 among services in service oriented application development.

6 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 [6] 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 directly from UML scenario specifications.

Our construction can be easily extended to treat timing constraints in UML SDs. For example, a message $m\{0..t\}$ states that the message m is constrained to last between 0 and t time units. Note that the semantics of UML on such messages is ambiguous and can have different meanings. For example, 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. Here we assume that the message 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 [4]. 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 [2]. As an example, we replace the message m_1 in Figure 8(a) by $m_1\{0..10\}$, and show the resulting Reo circuit in Figure 23. The timing constraints over the events for different messages can be captured similarly by adding synchronous drains and timer channels between their corresponding nodes.

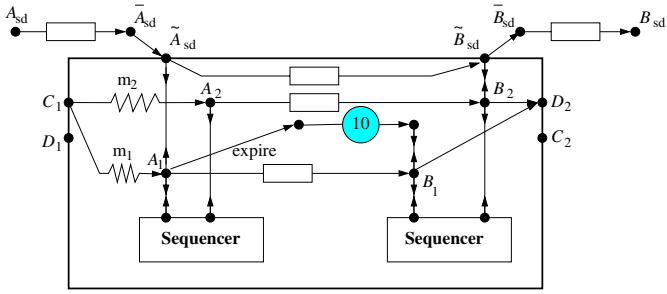


Fig. 23. Timed Reo circuit

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, and model checkers [1,11,19]. 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

[4] See [2] for P-producer and timer channels.

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 [19], 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) [25], 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 [6]. 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 [6] for the same UML SD. As both approaches are compositional, 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 [2] is in the scope of our future work as well.

Acknowledgement

The authors are indebted to the members of SEN3 for helpful discussions. 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

[1] Eclipse Coordination Tools. http://homepages.cwi.nl/~koehler/ect/.

[2] F. Arbab, C. Baier, F. de Boer, and J. Rutten. Models and Temporal Logics for Timed Component Connectors. In Jorge R. Cuellar and Zhiming Liu, editors, *SEFM2004, 2nd International Conference on Software Engineering and Formal Methods*, pages 198–207. IEEE Computer Society, 2004.

[3] Farhad Arbab. Reo: A Channel-based Coordination Model for Component Composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.

[4] Farhad Arbab, Christel Baier, Frank de Boer, Jan Rutten, and Marjan Sirjani. Synthesis of Reo Circuits for Implementation of Component-Connector Automata Specifications. In J.-M. Jacquet and G. P. Picco, editor, *COORDINATION 2005*, volume 3454 of *LNCS*, pages 236–251. Springer, 2005.

[5] Farhad Arbab, Tom Chothia, Sun Meng, and Young-Joo Moon. Component Connectors with QoS Guarantees. In A. L. Murphy and J. Vitek, editors, *Proceedings of 9th International Conference on Coordination Models and Languages, Coordination'07*, volume 4467 of *LNCS*, pages 286–304. Springer, 2007.

[6] Farhad Arbab and Sun Meng. Synthesis of Connectors from Scenario-based Interaction Specifications. submitted.

[7] Farhad Arbab and Jan Rutten. A coinductive calculus of component connectors. In M. Wirsing, D. Pattinson, and R. Hennicker, editors, *Recent Trends in Algebraic Development Techniques: 16th International Workshop, WADT 2002, Frauenchiemsee, Germany, September 24-27, 2002, Revised Selected Papers*, volume 2755 of *LNCS*, pages 34–55. Springer-Verlag, 2003.

[8] Marco Autili, Michele Flammini, Paola Inverardi, Alfredo Navarra, and Massimo Tivoli. Synthesis of Concurrent and Distributed Adaptors for Component-Based Systems. In V. Gruhn and F. Oquendo, editor, *EWSA 2006*, volume 4344 of *LNCS*, pages 17–32. Springer, 2006.

[9] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan Rutten. Modeling component connectors in Reo by constraint automata. *Science of Computer Programming*, 61:75–113, 2006.

[10] S. Balemi, G. J. Hoffmann, P. Gyugyi, H. Wong-Toi, and G. F. Franklin. Supervisory Control of a Rapid Thermal Multiprocessor. *IEEE Transactions on Automatic Control*, 38(7):1040–1059, 1993.

[11] Tobias Blechmann and Christel Baier. Checking Equivalence for Reo Networks. In *Proceedings of 4th International Workshop on Formal Aspects of Component Software, FACS'07*, 2007.

[12] Antonio Brogi, Carlos Canal, and Ernesto Pimentel. Behavioural Types and Component Adaptation. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Algebraic Methodology And Software Technology, 10th International Conference, AMAST'04, Proceedings*, volume 3116 of *LNCS*, pages 42–56. Springer, 2004.

[13] Werner Damm and David Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(0), 2001.

[14] D. Harel and H. Kugler. Synthesizing State-Based Object Systems from LSC Specifications. *Foundations of Computer Science*, 13:5–51, 2002.

[15] D. Harel, H. Kugler, and A. Pnueli. Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements. In *Proc. Formal Methods in Software and Systems Modeling*, pages 309–324, 2005.

[16] ITU-TS. Recommendation Z.120 : Message Sequence Chart (MSC), 1996. Geneva.

[17] ITU-TS. Recommendation Z.120(11/99) : MSC 2000, 1999. Geneva.

[18] Ismaïl Khriss, Mohammed Elkoutbi, and Rudolf K. Keller. Automating the synthesis of uml statechart diagrams from multiple collaboration diagrams. In *The Unified Modeling Language, UML'98: Beyond the Notation, First International Workshop*, volume 1618 of *LNCS*, pages 132–147. Springer, 1998.

[19] Sascha Klüppelholz and Christel Baier. Symbolic Model Checking for Channel-based Component Connectors. In C. Canal and M. Viroli, editors, *Proceedings of 5th Int. Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA'06)*, pages 19–36.

[20] Ingolf Krüger, Radu Grosu, Peter Scholz, and Manfred Broy. From mscs to statecharts. In *Distributed and Parallel Embedded Systems*, pages 61–72. Kluwer, 1999.

[21] Ingolf H. Krüger and Reena Mathew. Component Synthesis from Service Specifications. In S. Leue and T. J. Systä, editor, *Scenarios*, volume 3466 of *LNCS*, pages 255–277. Springer, 2005.

[22] Erkki Mäkinen and Tarja Systä. Mas - an interactive synthesizer to support behavioral modeling in uml. In *Proceedings of the 23rd International Conference on Software Engineering, ICSE 2001*, pages 15–24. IEEE Computer Society, 2001.

[23] S. Mauw and M.A. Reniers. High-Level Message Sequence Charts. In A. Cavalli and A. Sarma, editors, *SDL'97: Time for Testing - SDL, MSC and Trends, Proceedings of the Eighth SDL Forum, Evry, France*. Elsevier Science Publishers, 1997.

[24] Sun Meng and Luís Soares Barbosa. A Coalgebraic Semantic Framework for Reasoning about UML Sequence Diagrams. In *Proceedings of 8th International Conference on Quality Software, QSIC'08*. IEEE Computer Society, 2008.

[25] Object Management Group. *UML Profile for Schedulability, Performance, and Time, Version 1.1*. http://www.omg.org/cgi-bin/doc?formal/2005-01-02.

[26] Object Management Group. *Unified Modeling Language: Superstructure - version 2.1.1*, 2007. http://www.uml.org/.

[27] Jun Sun and Jin Song Dong. Design Synthesis from Interaction and State-Based Specifications. *IEEE Transactions on Software Engineering*, 32:349–364, 2006.

[28] Massimo Tivoli and Marco Autili. SYNTHESIS, a Tool for Synthesizing Correct and Protocol-Enhanced Adaptors. *RSTI L'object*, 12:77–103, 2006.

[29] Sebastian Uchitel, Greg Brunet, and Marsha Chechik. Behaviour model synthesis from properties and scenarios. In *29th International Conference on Software Engineering (ICSE 2007)*, pages 34–43. IEEE Computer Society, 2007.

[30] Sebastian Uchitel and Jeff Kramer. A Workbench for Synthesising Behaviour Models from Scenarios. In *Proceedings of International Conference on Software Engineering (ICSE'01)*, pages 188–197. IEEE Computer Society, 2001.

[31] Sebastian Uchitel, Jeff Kramer, and Jeff Magee. Detecting implied scenarios in message sequence chart specifications. In *Proceedings of the 9th European Software Engineering Conference and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE'01)*, pages 74–82. ACM, 2001.

[32] Daniel M. Yellin and Robert E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.

[33] Tewfik Ziadi, Loïc Hélouët, and Jean-Marc Jézéquel. Revisiting Statechart Synthesis with an Algebraic Approach. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*. IEEE Computer Society, 2004.