Formal Design and Verification of Long-Running Transactions with Extensible Coordination Tools

Natallia Kokash and Farhad Arbab, *Member*, *IEEE Computer Society*

Abstract—Ensuring transactional behavior of business processes and web service compositions is an essential issue in the area of service-oriented computing. Transactions in this context may require long periods of time to complete and must be managed using nonblocking techniques. Data integrity in long-running transactions (LRTs) is preserved using compensations, that is, activities explicitly programmed to eliminate the effects of a process terminated by a user or that failed to complete due to another reason. In this paper, we present a framework for behavioral modeling of business processes, focusing on their transactional properties. Our solution is based on the channel-based coordination language Reo, which is an expressive, compositional, and semantically precise design language admitting formal reasoning. The operational semantics of Reo is given by constraint automata (CA). We illustrate how Reo can be used for modeling termination and compensation handling in a number of commonly used workflow patterns, including sequential and parallel compositions, nested transactions, discriminator choice and concurrent flows with link dependences. Furthermore, we show how essential properties of LRTs can be expressed in LTL and CTL-like logics and verified using model checking technology. Our framework is supported by a number of Eclipse plug-ins that provides facilities for modeling, animation, and verification of LRTs to generate executable code for them.

Index Terms—Reo coordination language, long-running transactions, business process modeling, verifiable design

1 Introduction

Service-oriented computing advocates the idea of composing large business systems using loosely coupled self-contained services. A system constructed from independently designed and technology-agnostic services, nevertheless, has to be predictable, reliable, and consistent with application logic. Real-world business processes involve dozens of activities supplied by multiple partners. Their execution requires careful coordination, accounting for fault-tolerance, correct process termination and cancelation, without undesirable consequences at any stage of the execution. Therefore, realization of transactional behavior in service-oriented architectures (SOAs) is an indispensable task.

The term *transaction* is used to denote a compound unit of work performed completely or not at all. In traditional database systems, if something goes wrong during the execution of a transaction, a rollback activity is performed, which reestablishes the state of the system exactly as it was before the beginning of the transaction. Locks are acquired on the necessary resources at the beginning of a transaction and are released only at its end (in both cases of completion and rollback). The use of locks, which forbids others to access the resources, is justified by the short duration of the transaction. However, modern businesses created the need for new transactional processes in which remote entities interact by performing complex activities, both automated and manual, which require substantially longer times to complete, sometimes reaching days, weeks, months, or even years. Such prolonged times no longer allow the use of locks on resources, and, hence, makes transaction rollback impossible. In such transactions, called long-running transactions (LRTs), the alternative to rollback activities is the use of *compensations*, which are logical activities able to remove the effects of the performed actions.

In this paper, we present an approach to formal design of LRTs that extends our previous work [1] with more details on LRT semantics, modeling of timed LRT, description of our supporting software toolset, and examples of logic properties that can be verified with available model checking tools. Our approach relies on the channel-based coordination language Reo, which assumes that coordinated entities have no prior knowledge about each other. Reo has been successfully applied to service/component coordination [2], [3], business process modeling [4], and, in our view, is suitable for representing the logics of LRTs. A graphical notation along with several formal semantic models have been defined for Reo. This makes it applicable both for graphical design and automated verification using model checking tools. The corresponding tool support is provided. We consider Reo coordination in SOA as a bridge between domain-level design languages such as business process modeling notation (BPMN), and executable languages used for process implementation, e.g., WS-BPEL or Java. In this way, we assume an a-priori transactional behavior analysis that takes place before the system has been actually implemented. Only a process designer can decide whether a set of activities should be executed

• *The authors are with the Centre for Mathematics and Computer Science (CWI), Science Park 123, 1098 XG Amsterdam, The Netherlands. E-mail: {Natallia.Kokash, Farhad.Arbab}@cwi.nl.*

Manuscript received 2 June 2009; revised 6 July 2010; accepted 12 June 2011; published online 28 July 2011.
For information on obtaining reprints of this article, please send e-mail to: tsc@computer.org, and reference IEEECS Log Number TSC-2009-06-0153.
Digital Object Identifier no. 10.1109/TSC.2011.46.

transactionally and what should happen if their execution fails or the effects of the completed transaction should be canceled. Our approach offers a framework for a systematic design of LRTs where sets of compensatable service operations are grouped to logical workflows with the associated cancelation and/or compensation policies. Once the modeling of a (transactional) business process is completed, the designer can abstract from its internal details and use it in larger scale models with nested transactions. Due to its compositionality, Reo is suitable for modeling multiparty transactional processes where non of the parties has the view of the whole communication pattern: The "glue code" generated from the Reo model of a global LRT can be split into several parts and automatically deployed on various machines.

This paper is organized as follows: Section 2 contains an overview of related work. In Section 3, we discuss a domain-level LRT modeling. Section 4 is a brief introduction to Reo and illustrates its application to business process modeling. In Section 5, we discuss service coordination in sequential transactions. In Section 6, we focus on transactions with parallel flows and service coordination in some complex workflow patterns. Section 7 is dedicated to transactional property specification and model checking of Reo LRT models. Section 8 provides an insight into time-aware business transaction design. In Section 10, we describe Reo coordination tools from the perspective of their application to LRT modeling. Finally, Section 10 concludes the paper with an outline of our future work.

2 Related Work

A theoretical basis for LRTs is well established. A number of attempts have been made to formally specify exception and compensation handling in various workflow systems. For instance, Bocchi et al. [5] study the notion of LRTs incorporated into Microsoft BizTalk modeling environment. In this work, an extension of the asynchronous π-calculus is proposed to deal with LRTs, including the semantics of arbitrarily nested transactions [6]. However, this approach does not relate compensations with the control flow of the original process. For example, if one of the activities in a sequential flow fails, the compensations for all previously executed activities start simultaneously, while another (e.g., reverse) order may be required.

Butler and Ferreira [7] present an operational semantics for the Structural Activity Compensation (StAC) language. StAC is a business process modeling language inspired by the communicating sequential processes (CSP) with operators for compensation and exception handling. In [8], another CSP-based language for compensation orchestration, called *compensating CSP* or *cCSP* is proposed. Among the shortcomings of these languages are their complex semantics and noncompositional reasoning about the intended effects of a transaction. *Sagas Calculi* [9] have a more compact syntax, distinguish compensation and exception handling, and relate the behavior of the whole process with the success or failure of its atomic activities. For parallel processes, two versions of Sagas are proposed, *Naive* and *Revised*. A comparison of cCSP and Sagas [10] reveals that these two approaches account for different compensation policies when handling concurrent processes.

Gaaloul et al. [11] propose an event-driven approach to validate the transactional behavior of web service compositions. In this work, service compositions are specified using transactional patterns [12], which then are described in an event calculus to enable formal reasoning about their behavior. Transactional web service patterns can be seen as a convergence concept between workflow systems and transactional models. However, only very simple patterns such as a single parallel fork or a single parallel merge are considered in this work, and even for these constructs, specifying their transactional consistency as a set of logical formulas is rather cumbersome.

Several formalizations of failure, compensation and termination (FCT) handling in WS-BPEL have been proposed. Lucchi and Mazzara [13] introduce an orchestration language, called webπ, which is based on the idea of event notification as the unique error handling mechanism. Webπ is obtained by extending the π-calculus with a transactional construct composed of two processes. The authors show how WS-BPEL compensation handling can be reduced to event handling in the web π. However, this approach relies on statically specified compensation handlers and does not represent the default compensation in WS-BPEL. Laneve and Zavattaro [14] focus on the encoding of the WS-BPEL scope construct into the web π-calculus, but this work suffers from the same problems as the above approach. In [15], the theoretical foundation of scope-based flow languages is established. The authors propose a language, called *BPEL0*, that formalizes a subset of WS-BPEL. Eisentraut and Spieler [16] extend this work by providing support for repeating compensations, called *all-or-nothing semantics*, which allows for the compensation of failed compensations. Several works propose Petri net semantics for WS-BPEL. The most complete of them is given by Lohmann [17]. This approach formalizes control and data flows in WS-BPEL by means of open workflow nets (WFNs), a class of Petri nets extended with the interface for asynchronous message passing. Takemura [18] aims at formalizing the semantics of BPMN transactions using Petri nets. The mapping is not compositional and the resulting Petri nets for relatively simple case studies look complicated and difficult to understand. Moreover, such issues as cleaning of tokens in Petri net models of transactions with hazards are not considered.

In our approach, we do not adhere to any specific service composition language or workflow system. Our work, rather, aims at establishing a modeling framework able to unambiguously express any required compensation strategy. Therefore, we consider the most representative scenarios from the above papers and show how designers can benefit from using Reo in these cases. At a first glance, Reo is somewhat reminiscent of Petri nets. However, Petri nets normally offer synchronization at each transition of a net, whereas in Reo synchronization is defined by the types of channels connected together. This enables more concise representation of complex workflow patterns, including ones with exception handling and compensation mechanisms. Synchronous drain channels in Reo are convenient for modeling processes where token cleaning is required, while Petri nets are usually extended with inhibitor and reset arcs

for this purpose, which significantly reduces the number of software tools able to analyze such models [19].

Lanotte et al. [20] suggested the use of communicating hierarchical timed automata for modeling LRTs. This automaton-theoretic approach allows the time-aware verification of properties by model checking, but leaves the problem of appropriate fault handling in LRTs for further investigation. We believe that our modeling framework is more generic in the sense that it does not introduce special graphical constructs or automata for dealing with LRTs, but nonetheless, enables the verification of process transactional properties.

3 Domain-Level LRT Modeling

According to the BPMN [21], a widely used graphical language for domain-level process analysis, a business process can be represented in terms of *activities* carried out by humans or software applications, important *events* occurring in the process and a *control flow* on the involved activities. Additionally, BPMN supplies a number of modeling concepts more typical for implementation-level languages, such as subprocesses with exception handling, compensation associations, and transactions. For example, it assumes that an arbitrary process placed into a double-border rectangle is a *transaction*. The *compensation association* primitive is used to represent an activity with an associated compensation operation which should be executed to cancel its effects. A BPMN transaction is a group of such activities that must be all either successfully executed or canceled otherwise. There are three basic outcomes of a subprocess that represents a transaction:

- *Successful completion* when an execution token leaves the subprocess using the normal sequence flow.
- *Failed completion* when a transaction is successfully canceled, i.e., all its performed activities are compensated for and the token leaves the subprocess using a cancel intermediate event.
- *Exception* or *hazard completion* which means that neither successful nor failed completion is possible and the token leaves the subprocess using the exception flow originating from an error intermediate event attached to the boundary of the transaction.

BPMN has been designed for prompt sketching of business processes by domain experts and lacks precise semantics for unambiguous representation of process behavior, including a compensation handling mechanism for the specified transactions. Instead, WS-BPEL [22], a defacto standard for web service composition, defines primitives to describe a process flow at the execution-level, including its FCT handling. When a transaction fails, the effects of all its executed activities are negated by executing their respective compensations. By default, compensations in WS-BPEL are executed in the reverse order relative to the normal flow.

Observe that the notion of LRT in WS-BPEL is limited to a single business process instance, i.e., there is no distributed coordination among multiple-participant services. Technically, such coordination in SOA can be achieved by implementing protocols from WS-Transaction

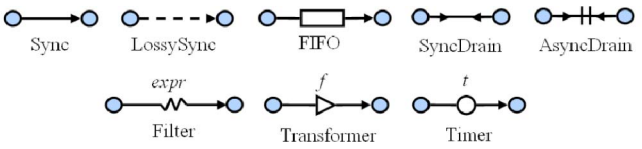


Fig. 1. Examples of Reo channels.

[23] specification, which identifies *atomic transactions* triggered using the classical ACID paradigm [24], and *business activity transactions*, which are managed by transaction coordinators. There are two protocols for managing LRTs. The first one is called *business agreement with participant completion*. In this protocol, a participant registers with the coordinator, so that the coordinator can manage it. A participant knows when it has completed all work for a business activity, and informs the coordinator when this is the case. In the *business agreement with coordinator completion* protocol a participant also registers with the coordinator. However, the end of a transaction is indicated by the coordinator. The outcome of a business activity can be atomic in nature or have a mixed outcome when some participants may commit results while others have to undo/compensate activities. Using the above solutions typically involves layering WS-Transaction protocols on top of WS-BPEL processes.

4 Reo Coordination Language: Semantics and Application

Reo is a coordination language in which components and services are coordinated exogenously by channel-based connectors [25]. Connectors are essentially graphs where the edges are user-defined communication channels and the nodes implement a fixed routing policy. Channels in Reo are entities that have exactly two ends, also referred to as ports, which can be either source or sink ends. Source ends accept data into, and sink ends dispense data out of their channels. Although channels can be defined by users, a set of basic Reo channels with predefined behavior suffices to implement rather complex coordination protocols. Fig. 1 shows such a set of Reo channels. Among these channels are

1. the Sync channel, which is a directed channel that accepts a data item through its source end if it can instantly dispense it through its sink end;
2. the LossySync channel, which always accepts a data item through its source end, tries to instantly dispense it through its sink end, and if this is not possible, loses the data item;
3. the SyncDrain channel, which is a channel with two source ends that accept data simultaneously and loses them subsequently;
4. the AsyncDrain channel, which accepts data items only through one of its two source channel ends at a time and loses it; and
5. the FIFO channel, which is an asynchronous channel with a buffer of capacity one.

Additionally, there are channels for data manipulation. For instance, the Filter channel always accepts a data item at its source end and synchronously passes or loses it depending on whether or not the data item matches a certain

predefined pattern or data constraint. Similarly, filter conditions can be added to the SyncDrain and AsyncDrain channels. Such channels appear useful for business process modeling when conditional synchronization of two flows is required. Finally, the Transform channel applies a user-defined function to the data item received at its source end and synchronously yields the result at its sink end.

Channels can be joined together using nodes. A node can be a source, a sink or a mixed node, depending on whether all of its coinciding channel ends are source ends, sink ends or a combination of both. Source and sink nodes together form the boundary nodes of a connector, allowing interaction with its environment. Source nodes act as synchronous replicators, and sink nodes as nondeterministic mergers. A mixed node combines these two behaviors by atomically consuming a data item from one of its sink ends at a time and atomically replicating it to all of its source ends.

The basic set of Reo channels can be extended to enable modeling of specific features of service communication. For example, timed Reo [26] channels were introduced to specify time-dependent interaction protocols, while probabilistic Reo channels [27] are used to build communication networks with unreliable links. Apart from functional aspects, channels can differ at the level of their nonfunctional characteristics. In quantitative Reo [28], channels are characterized by a set of associated QoS parameters such as data transfer delays or cost.

Complex connectors are constructed by composing simpler ones via the *join* and *hiding* operations. Join plugs two channel-ends together creating a node at their point of connection. To this node one can connect more channels via further join operation. If more than one accepting channel end is connected to a node every incoming message is simultaneously written to all outgoing channels whenever all outgoing channels in the node are ready to accept data. Whenever more than one channel-end offers data at a node a nondeterministic choice decides which data item is taken and written to all outgoing channels. The hiding operation hides away one node which means that the data-flow occurring at this node cannot be observed from outside and no new channel-end can be connected to this node. A complex connector has a graphical representation, called a *Reo circuit*, which is a finite graph where the *nodes* are labeled with pairwise disjoint, nonempty sets of channel ends, and the *edges* represent their connecting channels. The behavior of a Reo circuit is formalized by means of the data-flow at its sink and source nodes.

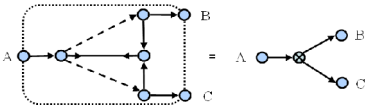


Fig. 2. Examples of Reo connectors: exclusive router.

Fig. 2 shows an implementation of an exclusive router using basic Reo channels. The connector provides three nodes A, B and C for other entities (connectors or component instances) to write to or take from. A data item arriving at the input port A flows through to only one of the output ports B or C, depending on which one is ready to consume it. The input data is never replicated to more than one of the output ports. If both output ports are ready to consume a data item, then the circuit selects one non-deterministically. To avoid drawing the circuit for an exclusive router every time it is used, we introduce a graphical shorthand notation similar to a node to represent this connector. We will also use XOR-nodes with $n > 2$ outputs. Such a connector can be defined by combining $n - 1$ exclusive routers with two outputs. Additionally, it is useful to define a priority on the outputs of an exclusive router in such a way that the data item will always flow into the prioritized output if more than one output is possible. When such a behavior of an exclusive router is assumed, we use a small exclamation mark to show its prioritized output in the corresponding Reo circuit.

Arbab et al. [4] define Reo connectors that simulate the behavior of basic BPMN modeling objects. By composing these connectors, one can model arbitrarily complex process workflows. For example, Fig. 3 shows an annotated Reo model for a fragment of the Purchase-to-Pay scenario within a procurement application. A corresponding BPMN diagram for this example can be found in Sadiq et al. [29]. In this scenario, two entities, Purchaser and Supplier, perform a number of activities within their workflows and exchange messages to coordinate their work. Each atomic activity is represented by a FIFO channel, which intuitively means that such an activity is started by accepting a flow token (data item) and completes by asynchronously disposing this token (data item). Observe

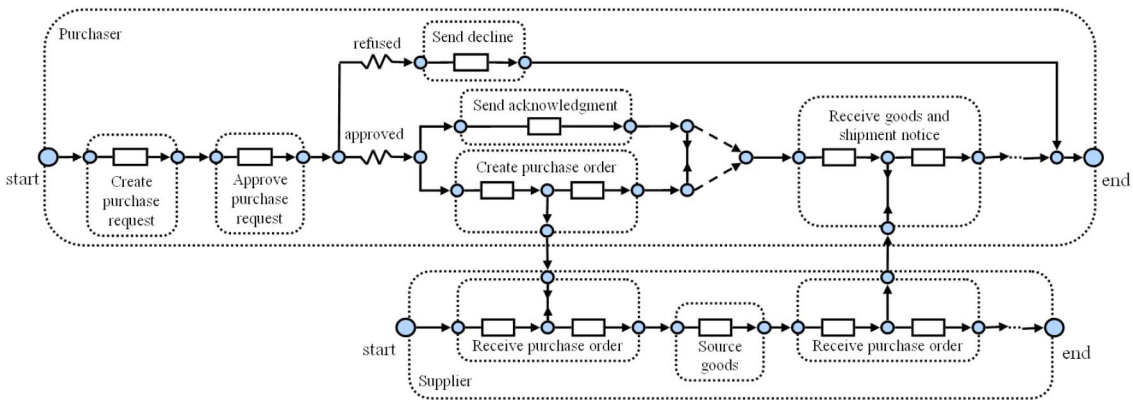


Fig. 3. Reo model for a fragment of the Purchase-to-Pay scenario.

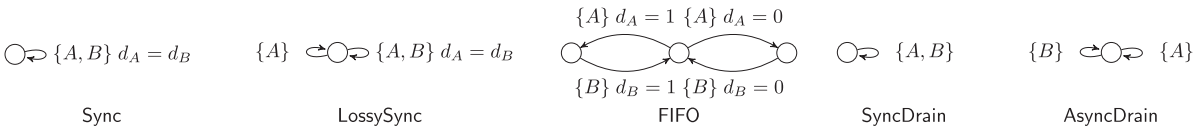


Fig. 4. Semantics of basic Reo channels and nodes.

that annotations on Reo circuits merely provide clues to help human understanding; they in no way affect the semantics of the circuits. The models with FIFO buffers that represent basic process activities (analogous to places in Petri nets) are mostly used for process simulation, validation and generation of state-transition systems for their formal verification. Once the designer is assured that the basic control/data flow logic is correct, components, which are black boxes with associated behavioral interfaces, can be attached to the model. These components can further be associated with WSDL files or other similar service interface specifications (e.g., semantic Web service standards). The resulting model can be converted to executable coordination code realized, e.g., in WS-BPEL, C++, or Java, with the invocation of existing web services.

The most basic model for expressing the operational semantics for Reo relies on constraint automata (CA) [30], which are essentially labeled transition systems with associated data constraints. Assuming that $\mathcal{N}$ is a finite set of node names and $Data$ is a fixed, nonempty set of data that can be sent and received via Reo channels, a function $\delta : N \to Data$ defines a data assignment for a subset of nodes $N \subseteq \mathcal{N}$. CA use a symbolic representation of data assignments by data constraints which are propositional formulas built from the atoms $d_A = d_B$, $d_A = d$ and standard Boolean connectors, where $A, B \in \mathcal{N}$, d_X is a symbol for the observed data item at the node X and $d \in Data$. We write $DA(N)$ to refer to the set of all data assignments for the node-set N, $DC(N)$ to denote the set of data constraints that at most refer to the observed data items d_A at node $A \in N$, and DC for $DC(\mathcal{N})$.

Definition 1 (Constraint Automaton (CA) [30]). *A constraint automaton $\mathcal{A} = (S, \mathcal{N}, \to, s_0)$ consists of a set of states S, a set of node names $\mathcal{N}$, a transition relation $\to \subseteq S \times 2^{\mathcal{N}} \times DC \times S$, where DC is the set of data constraints over a finite data domain* Data, *and an initial state $s_0 \in S$.*

We write $q \overset{N,g}{\to} p$ instead of $(q, N, g, p) \in \to$. Fig. 4 shows the CA for the basic Reo channels. Note that the constraint automaton shown for the FIFO is with respect to the data domain $Data = \{0, 1\}$.

The behavior of any Reo circuit composed of basic channels can be obtained by computing the product of the constraint automata of its constituent parts.

Definition 2 (Product of CA [30]). *The product of two CA $\mathcal{A}_1 = (S_1, \mathcal{N}_1, \to_1, s_0^1)$ and $\mathcal{A}_2 = (S_2, \mathcal{N}_2, \to_2, s_0^2)$ is defined as the constraint automaton $\mathcal{A}_1 \bowtie \mathcal{A}_2 = (S_1 \times S_2, \mathcal{N}_1 \cup \mathcal{N}_2, \to, \langle s_0^1, s_0^2 \rangle)$ where the transition relation $\to$ is determined by the following rules:*

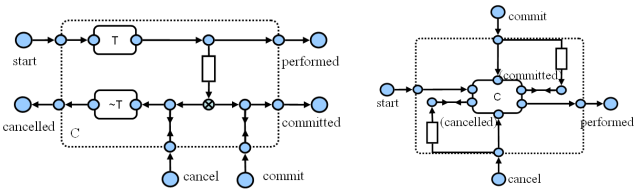
$$\frac{s_1 \overset{N_1,g_1}{\to_1} t_1 \quad N_1 \cap \mathcal{N}_2 = \emptyset}{\langle s_1, s_2 \rangle \overset{N_1,g_1}{\to} \langle t_1, s_2 \rangle} \qquad \frac{s_2 \overset{N_2,g_2}{\to_1} t_2 \quad N_2 \cap \mathcal{N}_1 = \emptyset}{\langle s_1, s_2 \rangle \overset{N_2,g_2}{\to} \langle s_1, t_2 \rangle}, \quad (1)$$

and

$$\frac{s_1 \overset{N_1,g_1}{\to_1} t_1 \quad s_2 \overset{N_2,g_2}{\to_1} t_2 \quad N_1 \cap \mathcal{N}_2 = N_2 \cap \mathcal{N}_1 = \emptyset}{\langle s_1, s_2 \rangle \overset{N_1 \cup N_2, g_1 \wedge g_2}{\to} \langle t_1, t_2 \rangle}. \quad (2)$$

CA are not expressive enough to describe precisely all details of the intuitive behavior of Reo. In particular, they cannot express the context-dependent behavior of the LossySync channel that loses a data item only if the environment or subsequent channels are not ready to consume it. Several models have been proposed to overcome this specific problem. The simplest of them is the connector coloring [31] which describes the behavior of Reo in a compositional fashion by coloring the parts of the circuit with and without dataflow using different colors that match on common ports/nodes. When three colors are used, the model captures context-dependent behavior by propagating negative information about the exclusion of dataflow through the connector. This model currently is used as a theoretical basis for Reo circuit animation and simulation tools. To provide a compositional semantics for Reo with communication delays, another automata-based model has been recently proposed [32]. This model generalizes CA in the sense that it distinguishes several actions observable on channel ports. This allows us to represent states in which a circuit is busy transferring data and, thus, cannot accept incoming requests, which provides a formal model for evaluating end-to-end delays in a connector.

In our recent work [33], [34], we expressed the behavior of Reo in the specification language mCRL2 [35]. mCRL2 is expressive enough to represent the behavior of all aforementioned three automata models. The basic notion in mCRL2 is the action. Actions represent atomic events and can be parameterized with data. Actions in mCRL2 can be synchronized using the synchronization operator |. Synchronized actions are called multiactions. Processes are defined by process expressions, which are compositions of actions and multiactions using a number of operators. Moreover, the mCRL2 language provides a number of built-in datatypes (e.g., Boolean, natural, integer) with predefined standard arithmetic operations and a datatype definition mechanism to declare custom types (also called sorts). We employed the mCRL2 toolset to generate state spaces for graphical Reo circuits and subsequently model check them. mCRL2 models for Reo circuits are generated in the following way [33]: observable actions (i.e., dataflow on the channel ends in the basic CA model) are represented as atomic actions, while data items observed at these ports are modeled as parameters of these actions. Analogously, we introduce a process for every node and actions for all channel ends meeting at the node. A custom sort $Data$ and the mCRL2 summation operator are used to model the input data domain and iterate over it while specifying data



(a) Internal view (b) Input/output hiding

Fig. 5. A task with an associated compensation activity.

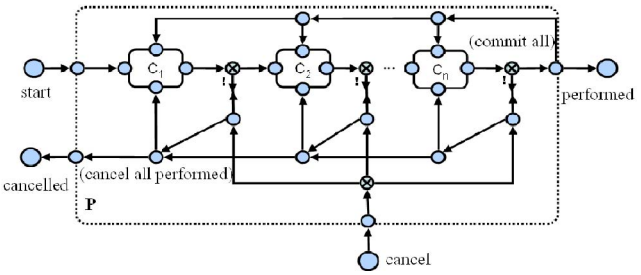


Fig. 6. A transaction consisting of n sequential tasks.

constraints imposed by channels. This work enriches Reo with a functional language for specifying data constraints and enables tool supported verification of data-aware service-based process models specified in Reo.

5 Sequential Flows

In this section, we apply Reo to model compensation strategies in transactional processes composed of a set of sequentially executed activities.

As mentioned in the introduction, BPMN introduces a special notation to identify tasks with associated compensation activities. According to this notation, only one activity can be marked as a target compensation activity, i.e., a sequence of compensation activities is not allowed. If several actions are required for the compensation, they must be combined into a single subprocess. Naturally, only activities that have been executed can be compensated for. Taking into account this description, an atomic task T with an associated compensation activity $\sim T$, written as $T \div \sim T$, can be represented as shown in Fig. 5a. In this Reo circuit, after the task T has executed, a token flows into the internal FIFO buffer which leads to the change of the connector state. An external user is notified that the task has been completed by the message observed on the port "performed." The performed task can be canceled or committed, where the effects of a committed task cannot be undone or canceled anymore. The token that resides in the FIFO buffer enables the connector to accept cancel or commit messages. If a cancel message arrives, the compensation activity $\sim T$ is executed and the task T is considered to be canceled. If a commit message arrives, the status of the task changes to "committed." The commands to commit or cancel the task effects are received from a transaction manager which generates them according to some global event such as output or failure of another service, timeout, or upon receiving a client's request to cancel the process. Such events are part of the application logic and can be modeled using Reo as discussed in [4].

Fig. 6 shows a Reo model for a transactional process $P \equiv C_1; C_2; \ldots; C_n$ consisting of a set of sequentially executed compensatable activities. Here, each connector $C_i \equiv T_i \div \sim T_i, 1 \leq i \leq n$, stands for an aforementioned compensation pair that includes an atomic task T_i whose effect is compensated for by another atomic task $\sim T_i$. In this scenario, unused outputs representing the "canceled" and the "committed" states of the connector for each compensation pair can be hidden using two FIFO and two SyncDrain channels as shown in Fig. 5b. For an external observer such a connector, after hiding, will have four I/O ports: three inputs for accepting "execute," "commit," and "cancel" messages, and one output representing the "performed" state of the source activity. The transactional process has several possible outputs. At the end of the successful transaction execution, that is, if no cancel message has been received, a token is back-propagated to commit all performed activities. If, instead, a cancel message has been received, it is picked up at a place where the execution token currently resides and back-propagated to cancel all performed activities. Since in this model we assume that an atomic compensation task cannot fail, the cancel message can be simultaneously forwarded to the output of the transactional process to signal the successful cancelation of the whole transaction.

Taking into account the design of the compensation pair connectors, namely, that after their source tasks have been executed, each connector is ready to accept the cancel message, we propagate the cancel message simultaneously to all performed activities. However, such a behavior can be easily changed by substituting synchronous channels going to the "cancel" port of each compensation pair connector $C_i, 1 \leq i \leq n$, with FIFO channels. In this case, each compensation activity will be activated independently.

In both variants of the circuit discussed above, the compensation activities are performed concurrently. However, a process may require an ordered execution of compensation activities. For example, Fig. 7 shows a transaction in which the effects of the tasks in a normal flow are compensated for in the reverse order with respect to the normal flow order. In this circuit, we use connectors representing compensation pairs with only one hidden port (corresponding to the "committed" state). A cancel message is sent to the "cancel" port of the circuit C_{i-1} only if the circuit C_i produces an output signalling that its task has been compensated for.

Now imagine that in some circumstances compensation activities may fail. Fig. 8a models a compensation pair that admits a failure of its compensation activity. In this model,

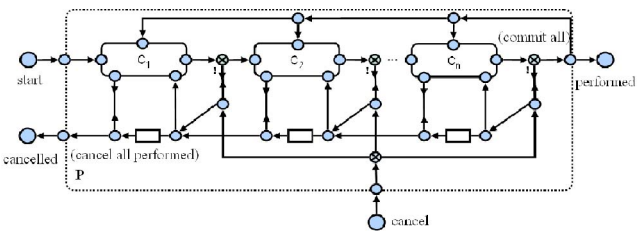


Fig. 7. A transaction consisting of n sequential tasks whose cancelation occurs in the reverse order.

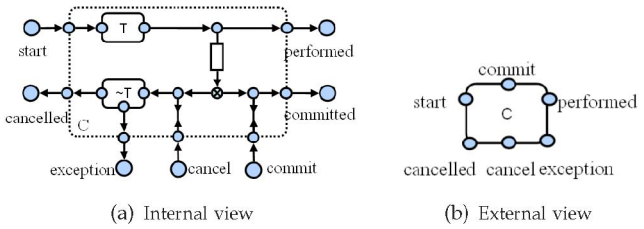


Fig. 8. A task with an associated compensation activity with possible failure.

the task $\sim T$ can have two outcomes, namely, successful completion, which means that the effects of the task T have been completely canceled, and exception, which signals that something went wrong while canceling the effects of the source task. After hiding the "committed" state of this connector we obtain a connector shown in Fig. 8b. Using such connectors, we can model transactions with exceptions, called *hazards* in BPMN. Fig. 9 shows a transaction process consisting of a set of sequentially executed activities with a possible hazard output. In contrast to the previous models, a cancel message cannot be simply propagated to the cancel output port. Instead, we need to ensure that all completed activities have been successfully canceled. This involves a structure similar to the one for executing and canceling parallel activities [4]. First, all compensation pair connectors receive cancel messages analogously to the sequential process in Fig. 6. Second, messages confirming the successful execution of all compensation activities must be received. Only in this case the transaction is considered successfully canceled. If some of the compensation activities fail, we can immediately signal the hazard event. However, in this case, a problem arises regarding the clean up of tokens returned by each of the invoked compensation activities. To resolve this problem, we use the same idea as for canceling a process consisting of a number of sequential activities: The exclusive router Y redirects the exception token to one of the places $y_i, 1 \leq i \leq n$, where the cancelation token currently resides, and both are disposed of in the corresponding synchronous drain (y_i, z_i). Additionally, tokens flow from this point into all available FIFO channels and wait until all compensation activities have disposed their tokens, either through the cancel output or through the exception output.

Note that designers are not supposed to directly construct complex circuits such as the one in Fig. 9. Instead, they can use higher level design languages such as BPMN, while Reo circuits can be regarded as the semantics of such higher level specifications. Moreover, it can be observed that all circuits for process modeling we introduced in this section are composed of relatively simple repeatable patterns, each easily understandable. After a correct circuit is constructed to reflect the semantics of a common modeling pattern such as, e.g., the sequential transaction with hazards, such a circuit can be converted to a component and used in higher level models.

6 Parallel Flows

Arbab et al. [4] examined how Reo can be used to coordinate parallel activities with exception handling. A Reo circuit for a parallel process $P \equiv C_1|C_2|\ldots|C_n$ is essentially composed of a parallel fork and a parallel join

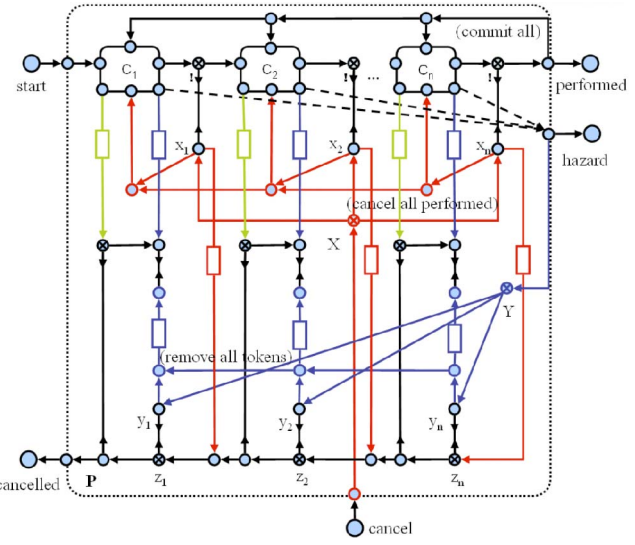


Fig. 9. A transaction consisting of n sequential tasks with possible failures in compensation activities.

gateways with n outgoing and n incoming branches, respectively. When an activity $T_i, 1 \leq i \leq n$, has completed, its corresponding token waits until other activities complete as well. After that, the token flows to the circuit output. For interrupting the process, a cancel message, either coming from an external source, or spawned by a failed activity in one of the parallel flows, is asynchronously directed to each of the remaining branches. A similar Reo connector can be used to cancel parallel activities within an LRT. Additionally to the aforementioned pattern, we must commit each activity after all branches have completed successfully.

Below, we consider LRTs for more complex scenarios involving parallel activities. For example, one of the interesting patterns is the so called *discriminator choice* which allows alternatives to be explored in parallel. Once one branch finishes successfully, all remaining alternatives are stopped and compensated for. A Reo circuit modeling such a behavior is shown in Fig. 10. The first completed branch initiates the compensation for all other branches. The compensation is performed asynchronously when the connector for its corresponding compensation pair is ready to accept the cancel message.

Some languages, e.g., WS-BPEL, provide a mechanism for adding control dependences to concurrent flows. This is done by means of *links*. A link is a directed connection between a source activity and a target activity. After a source activity is executed, the link is set to true, allowing the target activity to start.

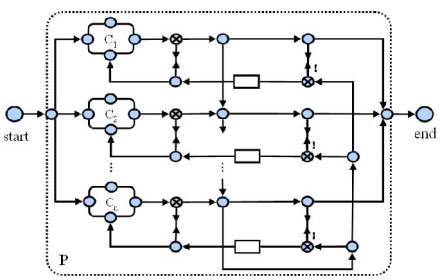


Fig. 10. A transaction consisting of the discriminator pattern with n parallel activities.

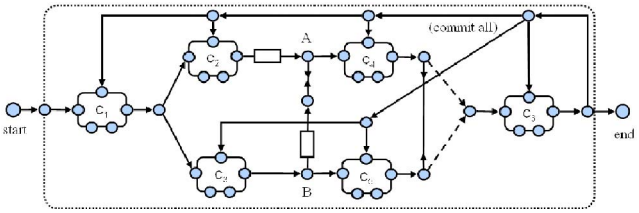


Fig. 11. Coordination of activities in parallel flows using links.

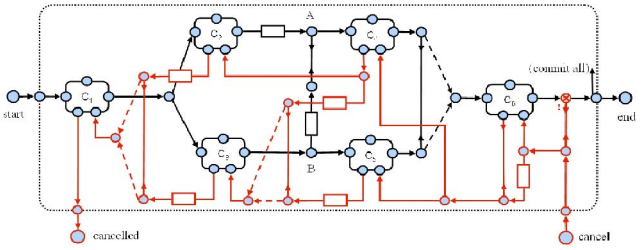


Fig. 12. Compensation of parallel flows with control links.

Consider a process $P \equiv C_1; (C_2; C_4 | C_3; C_5); C_6$ consisting of six compensation pairs $C_i \equiv T_i \div \sim T_i, 1 \leq i \leq 6$ (adopted from Bruni et al. [9]). In the normal flow, two pairs of tasks $(T_2; T_4)$ and $(T_3; T_5)$ are initiated after the task T_1 has completed, and execute concurrently. Now, assume that there is an additional constraint, written as $link(T_3, T_4)$, which states that the task T_4 must be executed after the task T_3 has completed. This constraint can be easily modeled with Reo using a FIFO1 and a synchronous drain channels connecting nodes A and B as shown in Fig. 11. One more FIFO1 channel is needed to keep the execution token returned by the connector C_2 while waiting for the completion of the task T_3 within the C_3 connector.

While control links are considered to be a useful mechanism for synchronizing concurrent flows, they obscure the desired compensation behavior in case of a process failure. We assume that such behavior can vary in different scenarios and must be dealt with by more refined modeling. For example, Fig. 12 shows a Reo circuit for the process compensation after executing the activity T_6. In this circuit, all activities are compensated for in the reverse order relative to the normal flow. In particular, the compensation activity for the task T_3 is activated after the compensations for the tasks T_4 and T_5 have completed, while the compensation for the task T_2 can be activated independently from that of the task T_5, but after the task T_4 has been compensated for.

Observe that by hiding internal nodes of Reo circuits, any transactional process can be presented in the form of a connector shown in Fig. 8b with or without exception/hazard output ports. Nested transactions, thus, can be handled by propagating messages in/out of their corresponding Reo connectors.

7 Process Verification

Finite-state verification is a powerful means to detect errors in concurrent systems that otherwise may be difficult to find and reproduce. With the help of Reo flash animation and simulation tools, designers can visually validate the behavior of a certain process model. However, since Reo connectors coordinating services in LRTs can be rather intricate, it is better to encode such properties as logic formulas and verify them automatically using model checking technology.

Several model checking tools are available for analyzing Reo. One of them is the symbolic model checker from the mCRL2 toolset.[1] This tool relies on the parameterized Boolean equation system (PBES) solver to encode model checking problems such as verifying first-order modal μ-calculus formulae on linear process specifications. Its application to the analysis of timed data-aware workflows modeled in Reo has been discussed in [33] and [36].

Alternatively, verification of Reo circuits can be accomplished with the help of the *Vereofy* model checker [37]. Vereofy uses two input languages, namely, the Reo Scripting Language (RSL), and a guarded command language called Constraint Automata Reactive Module Language (CARML), which are textual versions of Reo and CA, respectively. Scripts in these languages are automatically generated from graphical Reo/CA models. Nevertheless, they can be written manually as well, which can be useful, e.g., for the specification of large connectors with repeating patterns such as we observe in LRTs involving multiple services.

Regarding the analysis of component/service-based process model, the mCRL2 toolset has several advantages over Vereofy as it supports timed specifications with abstract data types and user-defined functions. However, both tools are suitable for control flow analysis, and since the mCRL2 property specification format based on μ-calculus is more difficult to use, we choose Vereofy to check essential temporal properties of LRT. For model checking with Vereofy, a constraint automaton needs to be associated with an arbitrary finite data domain Data, which collects all possible data items exchanged through the corresponding Reo circuit or stored within the local variables of the components. Data is a global data type, which in the current implementation of the Vereofy can be Bool, int, or enum, depending on the user settings. The default data domain is int(0,1) and we use it for control flow analysis.

Let $\mathcal{N}$ be the finite set of names representing the channel ends and Reo nodes and all interface ports of components. Let $A, B \in \mathcal{N}$ and $d \in Data$ a value from the data domain. I/O-constraints then have the following syntax $c := c \mid c_0 \wedge c$, where c denotes a constraint and

$$c_0 := A \mid \neg A \mid d_A = d \mid d_A \neq d \mid d_A = d_B \mid d_A \neq d_B,$$

is a constraint atom. An I/O-constraint stands for the access to a set of synchronized write and read operations at channel ends, ports, and Reo nodes. They are given by configurations of atoms of the form A (*"there is data flow at port A"*), $\neg A$ (*"there is no data flow at A"*), or conditions on the data item d_A exchanged through A. Conditions on the data flow may relate to either a constant value $d \in Data$ or the data value observed at some other port $B \in \mathcal{N}$.

Vereofy supports linear and branching-time model checking. Properties of Reo circuits can be specified either in the linear temporal logic (LTL) or the alternating-time stream logic (ASL), a variant of computation tree logic

1. http://www.mcrl2.org.

(CTL) extended with the ability to express conditions on data flow in channel nodes using regular expressions. LTL allows designers to encode formulae about the future of execution paths such as that some condition will eventually be true or will be true until another condition becomes true. CTL is a branching-time logic which models time as a tree-like structure and allows designers to encode formulae about the future of possible execution paths. For example, an ASL formula $AG[EX[true]]$ which literally means *"for all paths, it is globally true that there exists a next state"* can be used for deadlock detection.

Among the useful properties that we can verify are the following:

$$\forall i, 1 \leq i < n, \\ \mathbf{G}\ (C_i.start \rightarrow \mathbf{F}\ (C_i.committed \vee C_i.cancelled)).$$

This property says that either the committed state or the canceled state of each started compensation pair C_i will eventually be reached. We use quantifies to iterate over a set of n compensatable activities used in an LRT, meaning that the aforementioned temporal constraint should hold for every one of these activities. The next property

$$\mathbf{G}\ (P.commit \rightarrow \mathbf{F}\ \wedge_{i=1}^{n} C_i.commit),$$

states that the process commit implies the commitment of all of its involved activities.

The successful cancelation of an LRT occurs when the arrival of a cancelation message implies that all performed activities will be eventually compensated for (cancelled). This property can be checked using the following formula for parallel transactions:

$$\mathbf{G}\ (M \rightarrow \mathbf{F}\ \wedge_{i=1}^{n} C_i.cancelled).$$

A cancelation message can be received at any moment of the process execution. In our models of sequential LRTs, we do not wait until the process is completed, but initiate compensation after the completion of the atomic activity being performed at the moment of the arrival of the cancelation message. Therefore, the successful cancelation of the LRT can be formalized as follows:

$$\forall i, 1 \leq i < n, \\ \mathbf{G}\ ((C_i.start \rightarrow \mathbf{X}\ (\neg C_{i+1}.start \wedge M)) \\ \rightarrow \mathbf{F}\ \wedge_{j=1}^{i} C_j.cancelled).$$

In the case of the simultaneous invocation of compensation activities, this condition can be rewritten as

$$\forall i, 1 \leq i < n, \\ \mathbf{G}\ ((C_i.start \rightarrow \mathbf{X}\ (\neg C_{i+1}.start \wedge M)) \rightarrow \wedge_{j=1}^{i} C_j.cancel).$$

To guarantee the reverse order cancelation, it is enough to show that for each activity the compensation of its preceding activity starts at the next step after the successful cancelation of the current activity

$$\forall i, 1 \leq i < n, \\ \mathbf{G}\ ((C_i.start \rightarrow \mathbf{X}\ (\neg C_{i+1}.start \wedge M)) \rightarrow \\ \mathbf{F}\ C_1.cancelled \wedge_{j=2}^{i} (C_j.cancelled \rightarrow \mathbf{X}\ C_{j-1}.cancel)).$$

In the model shown in Figs. 6 and 7, we assumed that if all n activities have been successfully completed, the process is committed. However, it is easy to modify this process in a way that it will be cancelable even after the completion of all of its activities. For example, according to the *business agreement with coordinator completion protocol* suggested by WS-Transaction, the commitment is initiated by the external coordinator. In this case, conditions formalizing the successful cancelation of LRT may be slightly different.

Generally, LTL- and CTL-like logics supported by Vereofy allow designers to express various properties of business processes. Examples of useful properties can be found in Dwyer et al. [38]. This work provides property specification templates in LTL, CTL and regular expressions which allow designers to easily describe such facts as the absence or existence (including bounded existence) of certain events or states or cause-effect relationships between pairs or sequences of events/states.

8 Time-Aware LRTs

A timed transaction is as an activity or a subprocess that must be interrupted and compensated for if it does not complete before a specified time-out. In our approach, time-aware design of LRTs can be accomplished with the help of Reo timer channels.

The operational semantics of time-aware Reo circuits is given by *timed constraint automata (TCA)* [26], which are defined as follows: Let $\mathcal{C}$ be a finite set of clocks. A clock assignment is a function $v : \mathcal{C} \rightarrow \mathbb{R}_{\geq 0}$. If $t \in \mathbb{R}_{\geq 0}$ then $v + t$ denotes the clock assignment that assigns the value $v(x) + t$ to every clock $x \in C$. If $C \in \mathcal{C}$ then $v[C := 0]$ stands for the clock assignment that returns the value 0 for every clock $x \in C$ and the value $v(x)$ for every clock $x \in \mathcal{C} \setminus C$. A clock constraint (denoted cc) for C is a conjunction of atoms of the form $x \bowtie n$ where $x \in \mathcal{C}, \bowtie \in \{<, \leq, >, \geq, =\}$ and $n \in \mathbb{N}$. $CA(\mathcal{C})$ (or CA) denotes the set of all clock assignments and $CC(\mathcal{C})$ (or CC) the set of all clock constraints.

Definition 3 (Timed Constraint Automaton (TCA) [26]). *A TCA is an extended CA $\mathcal{A} = (S, \mathcal{N}, \rightarrow, s_0, ic)$ with the transition relation $\rightarrow \subseteq S \times 2^{\mathcal{N}} \times DC \times CC \times 2^{C} \times S$ such that $dc \in DC(N)$, C is a finite set of clocks and $ic : S \rightarrow CC$ is a function that assigns an invariance condition $ic(s)$ to any location s.*

This definition is similar to the definition of standard timed automata [39]. However, in contrast to simple timed automata, TCA contains three transition labels: 1) synchronization constraints that represent a set of ports where data flow is observed simultaneously, 2) data constraints that enable these transitions and, finally, 3) clock constraints. Fig. 13 illustrates how timed LRTs can be managed using Reo timer channels, namely a *t-timer with off and reset options* channel with source end A and sink node B. The source end of the channel accepts any input value $d \in Data$ and produces through its sink end a timeout signal after a delay of t time units. The *off-option* allows the timer to be stopped before the expiration of its delay when a special "off" value is consumed through its source end, while the

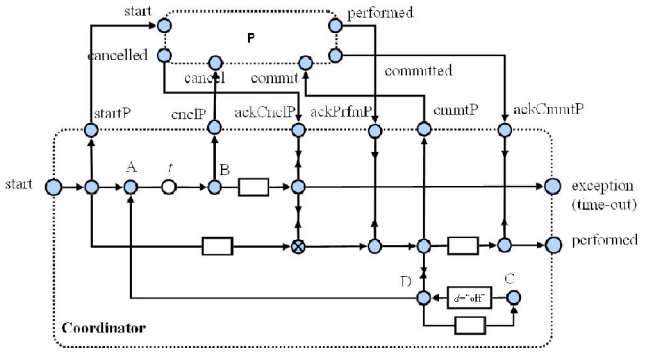


Fig. 13. Modeling timed LRTs.

reset-option allows us to reset the clock of the channel to 0 without switching it off. The TCA describing this behavior is shown in Fig. 14.

Thus, in our scenario, the *Coordinator* component keeps track of time, and either interrupts the transactional process P if it does not complete within the predefined timeout, or otherwise stops the timer and commits the process. Here, we use a full FIFO channel with the source end C and the sink end D to keep the special "off" value which is sent to the timer.

The timed properties of LRT models can be verified with the mCRL2 toolset. For the mapping of timer channels to mCRL2, we need to capture off, reset and timeout signals. Thus, along with the basic global data type Data the circuit should be able to deal with data items representing these options. Timer channels also behave differently when switched on or switched off. The details of the encoding of timed Reo in mCRL2 can be found in [36]. In a nutshell, the timer with off and reset options can be represented as the following parameterized mCRL2 process:

$$\begin{aligned}
&Timer(isOFF{:}Bool, x{:}Real, t{:}Real)\\
&\quad = isOFF \rightarrow (\Sigma_{d:DataTimer}\ isOther(d)\\
&\qquad \rightarrow A(d).Timer(false, 0, t)) \diamond\\
&\qquad ((x < t) \rightarrow (\Sigma_{d:DataTimer}\\
&\qquad\quad isReset(d) \rightarrow A(d).Timer(false, 0, t)\\
&\qquad\quad + isOff(d) \rightarrow A(d).Timer(true, x, t)\\
&\qquad\quad + tick@x.Timer(false, x+1, t)) \diamond\\
&\qquad\quad B(timeout).Timer(true, x, t));
\end{aligned}$$

Here, the operator $p + q$ means *alternative composition*; $p \cdot q$ *sequential composition*; the construct $c \rightarrow p \diamond q$, where c is a Boolean expression corresponds to the *conditional* operator or *if-then-else*; $\Sigma_{d:D}\ p$ is a *summation* operator used to quantify over a data domain D; and the *at* operator $a@t$ indicates that an action a happens at time t. The labeled transition system obtained with the help of mCRL2 tools is equivalent to the (T)CA-based semantics for Reo. The mCRL2 specifications for any Reo circuit is generated automatically and then converted to a labeled transition system. However, for reducing the state space, it is useful to generate mCRL2 specifications for frequently used behavioral patterns that are converted to components directly. When abstracted from the details of the data flow in internal nodes, the behavior of a connector observable at its

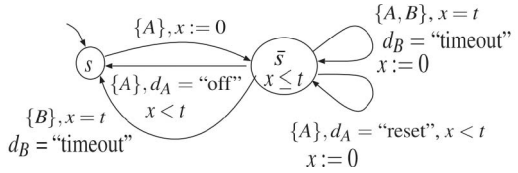


Fig. 14. TCA for the *t-timer with off-option.*

boundary nodes can be encoded in mCRL2 in a much more concise way and processed faster. In [36], for instance, we applied this approach to a connector that models a variable. For analyzing LRT models with mCRL2 toolset, we convert compensation pairs to their corresponding mCRL2 specifications. For example, the following mCRL2 process:

$$\begin{aligned}
&CompensationPair = \Sigma_{d:Data} \cdot\\
&\quad (start(d) \cdot performed(d) \cdot (cancel(d) \cdot cancelled(d)\\
&\quad + commit(d) | committed(d))) \cdot CompensationPair;
\end{aligned}$$

describes the observable behavior of the Reo model for the compensation pair in Fig. 5a. The properties of timed LRT can be specified in timed μ-calculus.[2] For example, we can verify that some action x happens within some time period after an action y. In the context of the given example, such a property can be used to check whether the transactional process P completes within t units of time after it has been initiated

$$\begin{aligned}
&[true^*]\forall \tau : R \cdot [start@\tau]\langle true\rangle\\
&\quad \exists u : R \cdot (u \leq \tau + t\ \wedge\ performed@u).
\end{aligned}$$

Alternatively, TCA can be analyzed with the SAT-based bounded model checker developed by Kemper [40]. In this work, the behavior of a TCA is represented by formulae in propositional logic with linear arithmetic to be analyzed by various SAT solvers. Since TCA provide operational semantics for timed Reo, this approach can be used for model checking time properties of LRTs. However, at the moment there is no tool for generating TCA from graphical Reo connectors. The development of such a tool for data-aware Reo will require the integration of TCA with some functional language for specifying constraints and functions used in filter and transformer channels. By using the mCRL2 toolset to obtain automata-based semantics of timed process models, we avoid this problem.

9 Modeling LRTs with Extensible Coordination Tools

In this section, we overview Reo tools for supporting business process modeling and LRT behavior analysis.

Reo coordination tools[3] consist of a set of plug-ins on top of the Eclipse platform.[4] Additionally, multiple other plug-ins, including ones for business process design and execution, have been developed for Eclipse. Our approach adheres to the principle of model-driven development and establishes

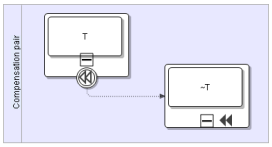
2. http://www.mcrl2.org/mcrl2/wiki/index.php/Language_reference/mu-calculus_syntax.
3. http://reo.project.cwi.nl/.
4. http://www.eclipse.org.

the connection of popular BPMNs and service composition languages such as BPMN, UML, or WS-BPEL with our proposed formal tools. The overall set of tools we apply to the design of transactional processes looks as follows:

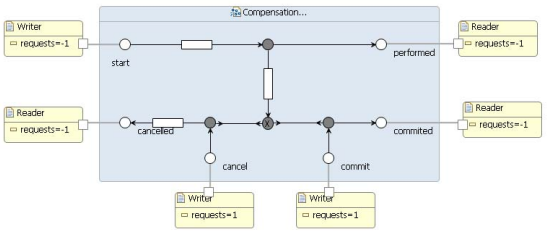
- *BPMN modeler*[5] is a graphical editor for creating BPMN diagrams. It is based on the graphical modeling framework (GMF) and uses an eclipse modeling framework (EMF) object model. The object model persists as XMI. Other modeling tools, e.g., Eclipse UML2,[6] can be used to design transactional business processes as well.
- *BPMN2Reo converter* is a plug-in for mapping BPMN diagrams into Reo models. We assume that BPMN diagrams are automatically converted to Reo models using a set of predefined ATL rules [41] and further refined to remove any ambiguity or semantic errors in the desired process behavior. A set of rules for mapping the aforementioned patterns have been specified in the prototype of the conversion tool [42].
- *Reo graphical editor* is a plug-in for the development of Reo connectors composed of the basic communication channel types. The editor supports hierarchical design by allowing previously defined Reo connectors to be converted to components and incorporated into new coarser-grained models.
- *Reo animation engine* is a plug-in that generates flash animated simulations of Reo connectors. Two animation modes are supported: a *plain* mode, which demonstrates the whole process, including all possible execution alternatives, and a *guided* or *stepwise* mode, which shows each execution step separately, including all possible alternatives for a current step.
- *Reo verification plug-ins* are tools for verifying Reo process models against formally specified properties. This set includes the aforementioned 1) Vereofy model checker, 2) the `mCRL2` toolset capable of verifying timed and data-aware process models [33], [36], and 3) a prototype of a SAT-based bounded model checker used for verifying timed (data agnostic) Reo networks [40].
- *Java code generation engine* is a plug-in that implements Reo circuits as a set of Java classes. This engine can be used to generate distributed processes from Reo models annotated with deployment information. A distributed version of the code generation engine is also available.

Fig. 15 demonstrate the application of ECT tools for LRT modeling and implementation. First, we assume that the designer models a transactional business process using a suitable modeling tool, e.g., the aforementioned Eclipse BPMN graphical editor. For example, Fig. 15a shows a BPMN modeling primitive for the compensation pair. Then, using the BPMN2Reo converter, a Reo model of the compensation pair shown in Fig. 15b can be obtained. This model gives operational semantics to the corresponding BPMN diagram. Fig. 15c shows a CA with four states and

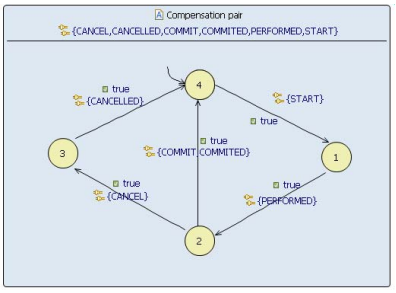
5. http://www.eclipse.org/stp/bpmn/.
6. http://www.eclipse.org/modeling/mdt/?project=uml2.



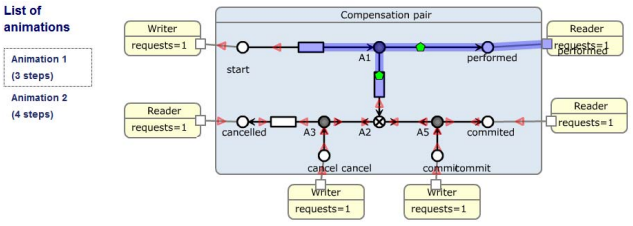
(a) BPMN compensation pair



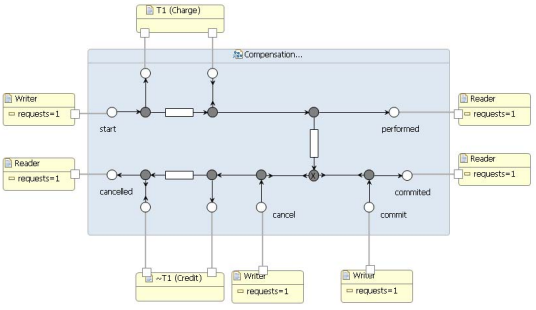
(b) Reo model of the compensation pair



(c) CA for the compensation pair



(d) Animation of the compensation pair



(e) Compensatable web services coordinated by the Reo circuit

Fig. 15. BPMN compensation pair modeled with ECT.

five transitions for the compensation pair generated from the above Reo process model. Here transition labels represent names of external (visible) ports, namely, "*start*," "*performed*," "*cancel*," "*canceled*," "*commit*," and "*committed*," where data flow can be observed by external user. Such a CA can be given as input to the Vereofy model checker together with the properties to be verified. For example, using an LTL formula

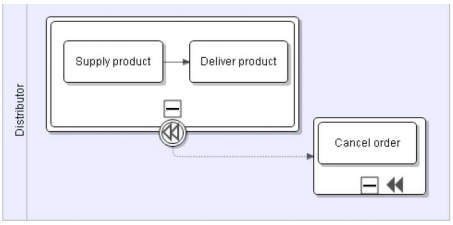


Fig. 16. A transactional product distribution process consisting of two services.

$$\mathbf{F}\ performed \rightarrow !(committed \vee cancelled)\ \mathbf{U}\ performed,$$

we can ensure that the results of the task cannot be canceled or committed until it has been performed. Similarly, using an LTL formula

$$\mathbf{G}\ (performed \rightarrow \mathbf{G}(cancel \rightarrow \mathbf{F}\ cancelled)),$$

we can check that after the task has been performed, it can be canceled in response to the cancel message.

For verifying timed dataflow of Reo process models with mCRL2, the user needs to select a graphical model and specify a property. The corresponding mCRL2 code will be generated automatically. The generation of mCRL2 code can be customized using various options. For instance, the option *with components* incorporates process definitions for the components attached at the boundary of a connector. The option *with data* enables data-aware encoding. Datatypes of components and services coordinated by Reo, as well as data constraints for data dependent channels such as Filter or Transform channels, can be defined using the same interface. They are saved as annotations in the Reo model and are propagated to the final mCRL2 specification. This way Reo circuits can be compiled automatically into mCRL2 without any manual editing. The tool further includes integrated space visualization tools. In particular, we use the `mcrl22lps` utility for the generation of the linear process representation of mCRL2 code, `lps2lts`, and `lpsconvert` for generating and minimizing labeled transitions systems, `lps2pbes` for symbolic model checking of modal μ-calculus formulas, and finally `ltsgraph` for visualization of state spaces.

The conversion tools create Reo counterparts for generic business process models. Due to the ambiguities of such models, automatically obtained Reo networks may require manual refinement. This is especially relevant for transactional processes as the high-level BPMN models usually lack the necessary technical information on how transactional properties of the process should be achieved. At this stage, the Reo animation engine is particularly useful. It allows designers to see simulations of the process execution in the form shown in Fig. 15d and correct the process if necessary. By wiring Reo circuits with component primitives as shown in Fig. 15e, we can incorporate service invocation in the process model and thus implement a service composition to realize the corresponding business process.

Reo connectors for process fragments can be further reused in more complex process models. For example, Fig. 16 shows a product distribution process that consists of a transaction with two basic activities, *supply* and *deliver*, and a subprocess *cancel order* which is executed if a customer decides to cancel his/her order.

Fig. 17 shows a snapshot of a Reo model created in the ECT environment for the sequential composition of two compensatable services. Using the above models, we can generate the Java code skeleton for

- coordinating two complementary activities within each of the compensation pairs, and
- coordinating two independent compensatable services within a sequential transaction.

To execute the generated code, the developer should implement the activities associated with the FIFO channels in the model. In our example, the FIFO channel is used in the compensation pair to model a state where the activity has been executed, but can still be canceled or committed. In practice, the status of each transaction is stored in some database, so at this point, the coordinator should check and update this status.

Note that these Reo connectors normally will be developed and executed by different parties, namely, providers of compensatable web services and an organization that composes such services to implement its own composite service with transactional behavior. We integrate all components of the LRT in a single model for verification purposes only. Although ECT currently does not support automated WSDL and WS-BPEL code generation, the LRT model can be easily realized using the standard web service stack. The model suggests that two WS-BPEL processes should be introduced to provide compensatable services *supply* and *deliver* with corresponding WSDL specifications. The boundary ports of Reo connectors correspond to operations in these specifications. Then, a WS-BPEL process can be defined to realizes the product distribution process through the invocation of each individual service and its compensation activities from the given WSDL specifications.

The performance of the model verification depends on the type of the analysis, choice of the semantic model and the underlying model checking tool. The translation of BPMN and WS-BPEL specifications to Reo requires polynomial time in the number of basic activities in the model. Vereofy model checker relies on the state-of-the-art techniques for model checking optimization and can deal with considerably large state spaces. However, the generation of the CA for graphical Reo circuits is rather time consuming. The mCRL2 toolset has been proven suitable for the state space generation and the analysis of the large industrial systems with tens of millions of states and transitions. For the verification of the properties that formalize the correctness of the control flow in the LRTs, basic CA models can be employed. These models are very concise, thus, the analysis of a middle sized process model (50 atomic activities corresponding to 2^{50} states) completes within several seconds with both Vereofy and mCRL2. For example, as was shown in [36], the generation of the state space for a circuit with 30 buffers takes less than 5 seconds. The analysis of the models with context-dependent LossySync channels is rather efficient as well, since such models have the same number of states as the CA models, and a slightly larger number of transitions. Regarding the timed analysis, the generation of an explicit state space for the timer channel with the delay in 1,000,000 time units requires around 11 minutes on a standard machine with 4 cores and

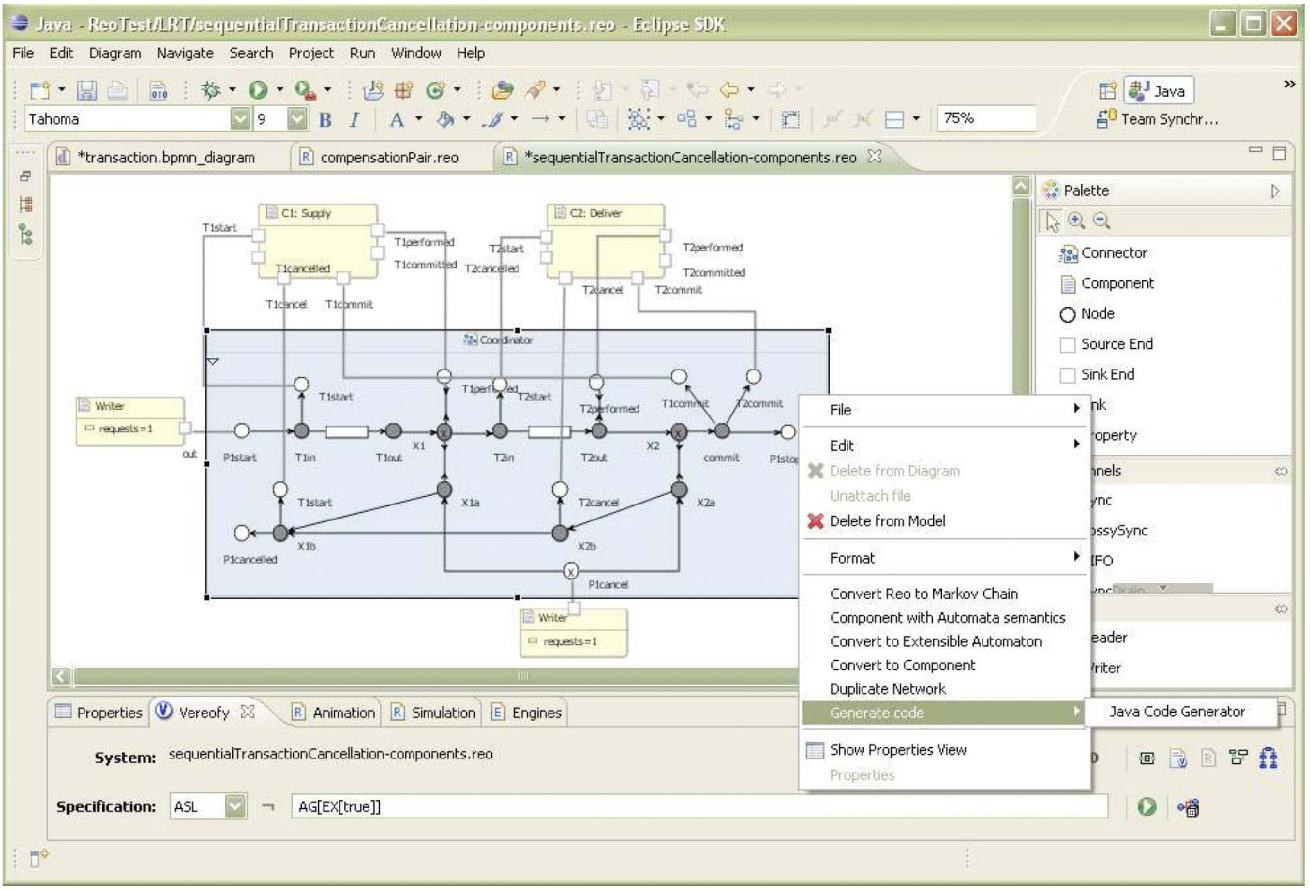


Fig. 17. A sequential transaction consisting of two compensation pairs modeled with ECT.

8 GB of memory, running Linux 2.6.27 and the January 2010 release version of mCRL2 (revision 201001). Note that the generation of the explicit state space is not needed if the mCRL2 symbolic model checking utility based on the PBES solver is used.

Dataflow model checking depends on the complexity of the input domain and filter constraints which affect the total number of states in the final state space. The analysis of Reo models with internal coordination and data transfer delays [32], which is useful, in particular, for estimating end-to-end communication within a single transaction, is much less efficient because of the larger number of states involved. However, for computing the execution time of a single transaction we do not need the entire state space, since this property can be computed on a small partial model. By abstracting away the irrelevant details of a model, e.g., via hiding of internal nodes or conversion of connectors to components, the applicability of the presented tools can be scaled up to tackle larger models.

10 Conclusions and Future Work

In this paper, we have shown how the Reo coordination language and its supporting software tools can be used to model business processes with transactional properties. Our approach makes it possible to formally verify the process transactional behavior using a number of validation and verification tools. We have illustrated how most typical workflow patterns with termination and compensation handling can be built using Reo channels. Automatic generation of Java code from verifiable LRT designs simplifies the task of LRT development and leads to more reliable LRT realizations.

Our approach has several advantages over existing formal tools for LRT modeling. Most of the workflow modeling languages and dedicated process-algebra-based approaches to LRT specification need special extensions to deal with tricky transactional patterns such as, e.g., discriminator choice, while Reo is able to cope with this task in a unified manner. Due to its combination of synchrony and asynchrony, Reo is more suitable for specifying exception and compensation handling in LRTs than classical Petri-nets. On the other hand, Reo is easier than process algebras, which makes it a promising technique for practical applications for designers without strong formal background. Furthermore, by introducing new channels (e.g., data transformers) in Reo and appropriate constraints in CA, we can deal with formal verification of various process properties including data-related constraints and nonfunctional requirements. Our ongoing work on Reo and CA with priorities will eventually allow designer to model highly flexible transactional processes able to favor certain services and interactions over others.

References

[1] N. Kokash and F. Arbab, "Applying Reo to Service Coordination in Long-Running Business Transactions," *Proc. ACM Symp. Applied Computing (SAC '09)*, pp. 318-319, 2009.

[2] A. Lazovik and F. Arbab, "Using Reo for Service Coordination," *Proc. Fifth Int'l Conf. Service-Oriented Computing (ICSOC '07)*, vol. 4749, pp. 398-403, 2007.

[3] F. Arbab, "A Behavioral Model for Composition of Software Components," *Proc. L'Objet: Selected Papers of the Int'l Workshop Coordination and Adaptation of Software Entities (WCAT '04)*, vol. 12, no. 1, pp. 33-76, 2006.

[4] F. Arbab, N. Kokash, and M. Sun, "Towards Using Reo for Compliance-Aware Business Process Modelling," *Proc. Leveraging Applications of Formal Methods Conf. (ISoLA '08)*, vol. 17, 2008.

[5] L. Bocchi, C. Laneve, and G. Zavattaro, "A Calculus for Long-Running Transactions," *Formal Methods for Open Object-Based Distributed Systems*, vol. 2884, pp. 124-138, 2003.

[6] L. Bocchi, "Compositional Nested Long-Running Transactions," *Proc. Fundamental Approaches to Software Eng. Conf. (FASE '04)*, vol. 2984, pp. 194-208, 2004.

[7] M. Butler and C. Ferreira, "An Operational Semantics for StAC, A Language for Modelling Long-Running Business Transactions," *Coordination Models and Languages*, vol. 2949, pp. 87-104, 2004.

[8] M. Butler, T. Hoare, and C. Ferreira, "A Trace Semantics for Long-Running Transactions," *Proc. Communicating Sequential Processes*, vol. 3525, pp. 133-150, 2005.

[9] R. Bruni, H. Melgratti, and U. Montanari, "Theoretical Foundations for Compensations in Flow Composition Languages," *Proc. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, vol. 40, no. 1, pp. 209-220, 2005.

[10] R. Bruni, M.J. Butler, C. Ferreira, C.A.R. Hoare, H.C. Melgratti, and U. Montanari, "Comparing Two Approaches to Compensable Flow Composition," *Proc. Int'l Conf. Concurrency Theory (CONCUR '05)*, vol. 3653, pp. 383-397, 2005.

[11] W. Gaaloul, M. Rouached, C. Godart, and M. Hauswirth, "Verifying Composite Service Transactional Behavior Using Event Calculus," *Proc. OTM Confederated Int'l Conf. the Move to Meaningful Internet Systems: CoopIS, DOA, ODBASE, GADA, and IS*, vol. 4803, pp. 353-370, 2007.

[12] O.P.S. Bhiri and C. Godart, "Transactional Patterns for Reliable Web Services Compositions," *Proc. Int'l Conf. Web Eng. (ICWE '06)*, pp. 137-144, 2006.

[13] R. Lucchi and M. Mazzara, "A Pi-Calculus Based Semantics for WS-BPEL," *J. Logic and Algebraic Programming*, vol. 70, no. 1, pp. 96-118, 2007.

[14] C. Laneve and G. Zavattaro, "Foundations of Web Transactions," *Proc. Eighth Int'l Conf. Foundations of Software Science and Computation Structures (FOSSACS '05)*, vol. 3441, pp. 282-298, 2005.

[15] G. Pu, H. Zhu, Z. Qiu, S. Wang, X. Zhao, and J. He, "Theoretical Foundations of Scope-Based Compensable Flow Language for Web Service," *Proc. Eighth IFIP WG 6.1 Int'l Conf. Formal Methods for Open Object-Based Distributed Systems (FMOODS '06)*, vol. 4037, pp. 251-266, 2006.

[16] C. Eisentraut and D. Spieler, "Failure, Compensation and Termination in BPEL 2.0 A Comparative Analysis," *Proc. Int'l Workshop Web Services and Formal Methods*, 2008.

[17] N. Lohmann, "A Feature-Complete Petri Net Semantics for WS-BPEL 2.0," *Proc. Int'l Workshop Web Services and Formal Methods*, vol. 4937, pp. 77-91, 2008.

[18] T. Takemura, "Formal Semantics and Verification of BPMN Transaction and Compensation," *Proc. IEEE Asia-Pacific Services Computing Conf.*, pp. 284-290, 2008.

[19] I. Raedts, M. Petković, Y.S. Usenko, J.M. van der Werf, J.F. Groote, and L. Somers, "Transformation of BPMN Models for Behaviour Analysis," *Proc. Int'l Workshop Modelling, Simulation, Verification and Validation of Enterprise Information Systems (MSVVEIS '07)*, pp. 126-137, 2007.

[20] R. Lanotte, A. Maggiolo-Schettini, P. Milazzo, and A. Troina, "Design and Verification of Long-Running Transactions in a Timed Framework," *Science of Computer Programming*, vol. 73, nos. 2/3, pp. 76-94, 2008.

[21] OMG, "Business Process Modeling Notation (BPMN) Specification," http://www.omg.org/bpmn/Documents/OMG_Final_Adopted_BPMN_1-0_Spec_06-02-01.pdf, Final Adopted Specification, 2006.

[22] A. Alves et al., "Web Services Business Process Execution Language (WS-BPEL) Version 2.0," OASIS, http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.pdf, Public Rev. Draft, 2006.

[23] W. Cox, F. Cabrera, G. Copeland, T. Freund, J. Klein, T. Storey, and S. Thatte, "Web Services Transaction (WS-Transaction)," technical report, BEA, IBM and Microsoft, http://dev2dev.bea.com/pub/a/2004/01/ws-transaction.html, 2004.

[24] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1993.

[25] F. Arbab, "Reo: A Channel-Based Coordination Model for Component Composition," *Math. Structures in Computer Science*, vol. 14, no. 3, pp. 329-366, 2004.

[26] F. Arbab, C. Baier, F.S. de Boer, and J.J.M.M. Rutten, "Models and Temporal Logics for Timed Component Connectors," *Int'l J. Software and Systems Modeling*, vol. 6, no. 1, pp. 59-82, 2007.

[27] C. Baier, "Probabilistic Models for Reo Connector Circuits," *J. Universal Computer Science*, vol. 11, no. 10, pp. 1718-1748, 2005.

[28] F. Arbab, T. Chothia, R. van der Mei, S. Meng, Y.-J. Moon, and C. Verhoef, "From Coordination to Stochastic Models of QoS," *Proc. 11th Int'l Conf. Coordination Models and Languages (COORDINATION '09)*, vol. 5521, pp. 268-287, 2009.

[29] S. Sadiq, G. Governatori, and K. Naimiri, "Modeling Control Objectives for Business Process Compliance," *Proc. Fifth Int'l Conf. Business Process Management (BPM '07)*, vol. 4714, pp. 149-164, 2007.

[30] C. Baier, M. Sirjani, F. Arbab, and J. Rutten, "Modeling Component Connectors in Reo by Constraint Automata," *Science of Computer Programming*, vol. 61, pp. 75-113, 2006.

[31] D. Clarke, D. Costa, and F. Arbab, "Connector Coloring I: Synchronization and Context Dependency," *Science of Computer Programming*, vol. 66, pp. 205-225, 2007.

[32] N. Kokash, B. Changizi, and F. Arbab, "A Semantic Model for Service Composition with Coordination Time Delays," *Proc. 12th Int'l Conf. Formal Eng. Methods and Software Eng. (ICFEM '10)*, pp. 106-121, 2010.

[33] N. Kokash, C. Krause, and E. de Vink, "Data-Aware Design and Verification of Service Composition with Reo and mCRL2," *Proc. ACM Symp. Applied Computing (SAC '10)*, pp. 2406-2413, 2010.

[34] N. Kokash, C. Krause, and E. de Vink, "Verification of Context-Dependent Channel-Based Service Models," *Proc. Int'l Conf. Formal Methods for Components and Objects (FMCO '10)*, vol. 6286, pp. 21-40, 2010.

[35] J. Groote, A. Mathijssen, M. Reniers, Y. Usenko, and M. van Weerdenburg, "The Formal Specification Language mCRL2," *Proc. Methods for Modelling Software Systems. Dagstuhl Seminar*, 2007.

[36] N. Kokash, C. Krause, and E. de Vink, "Time and Data Aware Analysis of Graphical Service Models in Reo," *Proc. IEEE Eighth Int'l Conf. Software Eng. and Formal Methods (SEFM '10)*, pp. 125-134, 2010.

[37] C. Baier, T. Blechmann, J. Klein, and S. Klüppelholz, "A Uniform Framework for Modeling and Verifying Components and Connectors," *Proc. 11th Int'l Conf. Coordination Models and Languages (COORDINATION '09)*, vol. 5521, pp. 268-287, 2009.

[38] J.C.M. Dwyer and G. Avrunin, "Property Specification Patterns for Finite-State Verification," *Proc. Int'l Workshop Formal Methods on Software Practice*, pp. 7-15, 1998.

[39] R. Alur and D. Dill, "A Theory of Timed Automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183-235, 1994.

[40] S. Kemper, "SAT-Based Verification for Timed Component Connectors," *Electronic Notes in Theoretical Computer Science*, vol. 255, pp. 103-118, 2009.

[41] F. Jouault, F. Allilaire, J. Tezivin, I. Kurtev, and P. Valduriez, "ATL: A QVT-Like Transformation Language," *Proc. 21st ACM SIGPLAN Symp. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '06)*, pp. 719-720, 2006.

[42] B. Changizi, N. Kokash, and F. Arbab, "A Unified Toolset for Business Process Model Formalization," *Proc. Int'l Workshop Formal Eng. Approaches to Software Components and Architectures (FESCA '10)*, pp. 147-156, 2010.

Natallia Kokash received the MSc degree in applied mathematics and computer science from Belarusian State University in 2004 and the PhD degree in information technology and communication from the University of Trento in 2008. She is a postdoctoral researcher in the Foundations of Software Engineering Group at the Dutch National Research Center for Mathematics and Computer Science in Amsterdam. Her current research interests include service-oriented computing, business process modeling, and formal methods in software engineering. She has several years of professional experience in banking software development and worked on EU FP6/7 projects PLASTIC, NEXOF-RA, and COMPAS.

Farhad Arbab received the PhD degree in computer science from the University of California, Los Angeles, in 1982. He is a senior researcher in the Foundations of Software Engineering Group at the Dutch National Research Center for Mathematics and Computer Science in Amsterdam and holds the chair of Software Composition as a professor of computer science at Leiden University, the Netherlands. His fields of interest include software composition, coordination models and languages, service-oriented computing, component-based systems, and concurrency. He leads the work on the development, implementation, and applications of Reo. He is a member of the IEEE Computer Society.