Applying Reo to Service Coordination in Long-Running Business Transactions

Natallia Kokash and Farhad Arbab
CWI, Kruislaan 413, Amsterdam, The Netherlands
firstName.lastName@cwi.nl

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 non-blocking techniques. Compensations are activities executed to preserve data integrity and eliminate the effects of a process terminated by a user or that failed to complete due to another reason. This paper presents an approach to formal modeling of long-running business transactions. Our solution is based on the channel-based exogenous coordination language Reo, which is an expressive, compositional and semantically precise design language that admits formal reasoning. We illustrate how Reo can be used for termination and compensation handling in a number of commonly-used workflow patterns, including sequential and parallel compositions, discriminator choice and concurrent flows with link dependencies.

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, analysis of transactional behavior of Service-Oriented Architectures (SOAs) is an indispensable task.

Each transaction in SOA can be kept open much longer than in traditional distributed database systems, reaching days and even months for certain processes. In this context, resource locks and isolation cannot be maintained for long periods of time. The concept of Long Running Transactions (LRT) has been introduced to cope with this problem. Chaudhari at al. [1] identify two complementary strategies for managing LRTs in SOA, namely,

– Compensation methodology. Any changes performed during an LRT execution must be reverted back if a failure occurs somewhere in the flow of the
transaction. However, instead of the traditional rollback, SOA assumes compensation, that is, execution of the most logical activity for maintaining data consistency and integrity in a certain context.

- **Transaction coordinator.** In the case of the asynchronously communicating services, a transaction manager or process coordinator should be used to orchestrate the process. Instead of inter-service communication, services are invoked and provide results to such a coordinator, which handles all compensation scenarios.

According to the Business Process Modeling Notation (BPMN) [2], a widely used graphical language for domain-level process analysis, a business process can be represented in the form of activities produced by humans or software applications, important events occurring in the process and control flow on the involved activities. Additionally, BPMN supplies a number of modeling concepts more typical for implementation-level languages, such as sub-processes 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 sub-process that represents a transaction: (i) successful completion when an execution token leaves the sub-process using the normal sequence flow (ii) failed completion when a transaction is successfully canceled, i.e., all its performed activities are compensated for and the token leaves the sub-process using a cancel intermediate event, and (iii) exception or hazard completion which means that neither successful nor failed completion are possible and the token leaves the sub-process 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 compensation handling mechanism for the specified transactions. Instead, WS-BPEL [3], a de-facto standard for web service composition, defines primitives to describe a process flow at the execution-level, including its Failure, Compensation and Termination (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.

It can be noticed 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 [4] specification, which identifies atomic transactions triggered using the classical ACID paradigm, and business activity transactions, which are managed by transaction coordinators. However, this specification does not aim at providing means for rigorous analysis of entities involved into transactions.
A number of formally grounded approaches have been proposed to examine LRT behavior both in general settings and within SOA-related solutions such as WS-BPEL [5–10]. While proposing valid models for specifying transactional behavior, none of these languages considers full-featured LRT management as a specific, although rather arduous, case of service coordination. A more detailed overview of these works is given in Section 2.

In this paper, we present an approach to unambiguous business process modeling with special focus on transactional behavior specification. Our approach relies on the channel-based exogenous coordination language Reo, which assumes that coordinated entities have no prior knowledge about each other. Reo has been successfully applied to service/component coordination [11, 12] as well as to building Mash-Ups [13], and, in our view, is suitable for representing logics of LRT managers. 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 of important process properties such as absence of deadlocks, livelocks or unreachable states, 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 BPMN, and executable languages used for process implementation, e.g., WS-BPEL, WS-CDL [14] or Java. In this way, we assume a-priori transactional behavior analysis which takes place before the system has been actually implemented.

This paper is organized as follows. Section 2 contains an overview of related work. Section 3 is a brief introduction to Reo and illustrates its application to business process modeling. In Section 4, we discuss service coordination in sequential transactions. In Section 5, we focus on transactions with parallel flows and service coordination in some complex workflow patterns. In Section 6, we describe Reo coordination tools from the perspective of their application to LRT modeling. Finally, Section 7 concludes the paper and outlines 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. Thus, Bocchi [5] studies the notion of LRTs incorporated into Microsoft BizTalk modeling environment. In this work, an extension of the asynchronous π-calculus, called πt-calculus, is proposed to deal with LRTs, including the semantics of arbitrary nested transactions [15]. However, this approach does not relate compensations with the control flow of the original process. For example, if one of the activities in the sequential flow fails, the compensations for all previously executed activities start simultaneously, while another (e.g., reverse) order may be required.

Butler and Ferreira [6] present an operational semantics for the StAC (Structural Activity Compensation) language. StAC is a business process modeling language inspired by the Communicating Sequential Processes (CSP), from which it borrows most of its constructs. Additionally, StAC introduces operators for
compensation and exception handling. Apart from the complex definition of StAC's operational semantics, its main shortcoming is that StAC does not support reasoning about the intended effects of the transaction in a compositional way. In [7], another CSP-based language for compensation orchestration, called compensating CSP or cCSP is proposed. This language overcomes some of the aforementioned drawbacks.

Another fundamental work in the area of formal LRT specification is Sagas Calculi [10]. Sagas has a more compact syntax, distinguishes compensation and exception handling and relates the behavior of the whole process with the success or failure of its atomic activities. An extensive set of process patterns is considered, including sequential, parallel and nested processes, as well as additional features such as discriminator choice or link dependencies. For parallel processes, two versions of Sagas are proposed, Naive and Revised. This approach can also deal with restricted programmable compensations, i.e., compensation procedures defined by a programmer for a specific subprocess as opposed to the implicit or default compensation.

A comparison of cCSP and Sagas [16] reveals that these two approaches account for different compensation policies when handling concurrent processes. In cCSP, parallel branches may be stopped when one flow aborts, but the activation of the compensation procedure is handled in a centralized way, that is, it can be executed only when all flows have been stopped. In Naive Sagas, parallel branches execute until completion, but can be compensated for without waiting for the completion of their siblings. In Revised Sagas, parallel branches can be interrupted and their compensation procedures activated independently from the rest of the flow.

Gaaloul et al. [17] propose an event-driven approach to validate the transactional behavior of web service compositions. In this work, service compositions are specified using transactional patterns [18], 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. Thus, this approach appears to be inefficient for complex processes where compensation behavior is “global”, i.e., involves previously executed activities, and is subject to separate modeling.

Several formalizations of FCT handling in WS-BPEL have been proposed. For example, Lucchi and Mazzara [19] 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 webπ. However, this approach relies on statically specified compensation handlers and does not represent the default compensation in WS-BPEL. Laneve and Zavattaro [20] focus on the encoding of the WS-BPEL scope construct into the webπ-calculus,
but this work suffers from the same problems as the above approach. In [21], the theoretical foundation of scope-based flow languages is established. The authors propose a language, called BPEL0, that formalizes a subset of WS-BPEL. Eisen- traut and Spieler [8] 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 [22]. This approach formalizes control and data flow in WS-BPEL my means of Open Workflow Nets (WFNs), a class of Petri nets extended with the interface for asynchronous message passing.

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 somewhat reminds 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, along with the compositional nature of Reo, enables more concise representation of complex workflow patterns, including ones with exception handling and compensation mechanisms. See [12] for a mode detailed comparison of Reo and Petri nets.

3 Reo Coordination Language

Reo [23] is a channel-based exogenous coordination model wherein complex coordinators, called connectors, are compositionally constructed from simpler ones. The simplest connectors are primitive binary connectors called channels. Connectors serve to provide the protocol which controls and organizes the communication, synchronization, and cooperation among the components/services that they interconnect. As a binary connector, each channel has two channel ends which can be of two types: source and sink. A source end accepts data into its channel, and a sink 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 together. Figure 1 shows the graphical representation of basic Reo channel types. A FIFO1 channel represents an asynchronous channel with one buffer cell which is empty if no data item is shown in the box (this is the case in Fig. 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 accepts any data item, but the value dispensed through its sink is always a data element \( d \in P \). (A)synchronous drains have two source ends and no sink end. A synchronous drain can accept a data item through one of its ends if 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. A timer channel with early expiration 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 [24].

Channels are joined together in a node which consists of a set of channel ends. The hiding operation is used to hide the internal topology of a component connector. 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 their connecting channels. The behavior of a Reo circuit is formalized by means of the data-flow at its sink and source nodes. Further details about Reo and its semantics can be found in [23–25].

Figure 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 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 available. 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. [26] define Reo connectors that simulate the behavior of basic BPMN modeling objects. By composing these connectors, one can model arbitrary 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 can be found in Sadiq et al. [27]. 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 FIFO1 channel, which intuitively means that such an activity is started by accepting a flow token (data object) and completes by asynchronously disposing this token (data object). Observe that annotations on Reo circuits merely provide clues to help human understanding; they in no way affect the semantics of the circuits.

4 Sequential Flows

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

As was 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 should be combined into a single sub-process. 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. 4(a). In this Reo circuit, after the task $T$ has executed, a token flows into the FIFO1 channel and 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” and we assume that the task effects cannot be undone or canceled anymore. 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 [26].

Figure 5 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 each compensation pair connector can be hidden using two FIFO1 and two synchronous drain channels as shown in Fig. 4(b). 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 messages have 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 of the connectors 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 FIFO1 channels. In this case, each compensation activity will be activated independently.

In both discussed variants of the above circuit the compensation activities are performed concurrently. However, a process may require ordered execution of compensation activities. For example, Fig. 6 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. Figure 7(a) models a compensation pair that admits a failure of its compensation activity. In this model, 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. 7(b). Using such connectors, we can model transactions with exceptions, called hazards in BPMN. Figure 8 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 [26]. First, all compensation pair connectors receive cancel messages analogously to the sequential process in Fig. 5. 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. 8. Instead, they should 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, easily understandable one by one.

5 Parallel Flows

Arbab et al. [26] examined how Reo can be used to coordinate parallel activities with exception handling. A Reo circuit for a parallel process $P \equiv C_1|C_2|...|C_n$ is essentially composed of a parallel fork and a parallel join gateways with $n$ outgoing and $n$ incoming branches, respectively. When an activity $T_i$, $1 \leq i \leq n$,
Fig. 8. Transaction consisting of $n$ sequential tasks with possible failures in compensation activities has completed, the 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 the remaining alternatives are stopped and compensated for. A Reo circuit modeling such a behavior is shown in Fig. 9(a). The first completed branch initiates the compensation for all other branches. The compensation is performed asynchronously when the corresponding compensation pair connector is ready to accept the cancel message.

This circuit has been designed to be extensible for an arbitrary number of involved activities. However, for practical cases easier patterns can be found. For example, Fig. 9(b) shows a discriminator with two alternative activities. Two asynchronous drains ($A_1, A_2$) and ($B_1, B_2$) assure that the result of only one of the performed activities is considered while another activity is compensated for. If both activities complete simultaneously, one of them will be accepted non-deterministically.
Some languages, e.g., WS-BPEL, provide a mechanism for adding control dependencies 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.

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. [10]). 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 executed 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. 10. 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 is subject to separate modeling. For example, Fig. 11 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
Fig. 11. Compensation of parallel flows with control links

compensations for the tasks $T_4$ and $T_5$ have completed, while the compensation for the task $T_2$ can be activated independently from the task $T_5$, but after the task $T_4$ has been compensated for.

Nested transactions can be handled by propagating messages in/out of the corresponding Reo connectors.

6 Tool Support

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

Reo coordination tools\(^1\) represent a set of plug-ins on top of the Eclipse platform\(^2\). Additionally, multiple other plug-ins including ones for business process design and execution have been developed for Eclipse. Thus, we assume that our framework consists of the following parts:

- **BPMN modeler\(^3\)** 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 XML. Generally, other modeling tools, e.g., Eclipse UML2\(^4\), can be used to design business processes as well.

- **BPMN2Reo converter** is a plug-in for mapping BPMN diagrams represented in the form of EMF models into Reo EMF models, which is currently under development. We assume that initial BPMN models will be represented by means of Reo connectors and further refined to remove any ambiguity or semantic errors in the desired process behavior. Additionally, there is ongoing work on converting UML Sequence Diagrams to Reo, which can be useful in our context as well.

- **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 incorpo-

\(^{1}\) [http://homepages.cwi.nl/~koehler/ect/](http://homepages.cwi.nl/~koehler/ect/)
\(^{2}\) [http://www.eclipse.org](http://www.eclipse.org)
\(^{4}\) [http://www.eclipse.org/modeling/mdt/?project=uml2](http://www.eclipse.org/modeling/mdt/?project=uml2)
rated into new coarser-grained circuits. A library of useful Reo connectors\(^5\) (e.g., simulating the most common workflow patterns) has been created. By reusing such pre-specified patterns, designers can model complex workflows easier and faster.

- **Reo reconfiguration plug-in** is a tool that allows designers to dynamically reconfigure Reo circuits. Reconfiguration rules can be generated automatically given a source connector and a target connector, and then applied to any Reo circuit with patterns similar to the source connector as many times as needed. The format and semantics of reconfiguration rules are based on the theory of algebraic graph transformations. In our context, this tool can be useful to deal with certain elements of business processes that presume run-time modification (e.g., BPMN macros for activity looping or multiple instances of the same activity).

- **Reo simulation engine** is a plug-in that generates flash animated simulations of Reo connectors. Two simulation 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.

- **Java code generation engine** is a plug-in that implements Reo circuits as a set of Java classes. Theoretically, the implementation of Reo using WS-BPEL is also possible [11].

- **Reo validation plug-in** is a tool that performs model checking over process models represented as constraint automata. This tool supports verification of properties expressed in LTL and CTL-like logics.

The Reo simulation engine can be used to assure that workflow models are compliant with important transaction properties such as (i) *durability* (no more than one output is reached for any process run), (ii) *eventuality* (output is achieved for any process run) or (iii) *atomicity* (all involved activities are either successfully completed or successfully canceled). With the help of Reo flash animations, designers can observe whether these and other properties hold for a certain process. However, since Reo connectors coordinating services in LRTs can be rather intricate, it is better to encode such properties in CTL-like logics and verify them automatically using an appropriate model checking tool [28].

7 Conclusions and Future Work

In this paper, we have shown how the Reo coordination language can be used to model business processes with transactional properties. We have designed a number of connectors that enable termination and compensation handling in the most typical workflow patterns. The resulting Reo models make it possible to formally verify the process transactional behavior using available verification tools.

\(^5\) [http://homepages.cwi.nl/~proenca/webreo/home.htm](http://homepages.cwi.nl/~proenca/webreo/home.htm)
Our approach has several advantages over existing frameworks. Most of the workflow modeling languages and dedicated process-algebra-based approaches to LRT specification need special extensions to deal with tricky patterns such as, e.g., *m-out-of-n choice* or *synchronizing merge*, while Reo is able to cope with this task in a unified manner. Moreover, Reo provides a comprehensive visual notation for its channels, which makes it easy to use by software designers without any experience in process calculi.

Due to the lack of space, several aspects of our work have not been considered here and will be covered in upcoming publications. For example, some business processes involve dynamically created activities and assume run-time workflow evolution. We plan to deal with such structures using dynamic reconfiguration of Reo connectors based on graph transformation rules. As immediate future work, we plan to complete the implementation of the BPMN2Reo converter and enable an automated formal business process and LRT design as discussed in this paper.

8 Acknowledgements

This work is part of the IST COMPAS project, funded by the European Commission, FP7-ICT-2007-1 contract number 215175, http://www.compas-ict.eu/.

References


