Specification of a Behavioral Model for Services

Project acronym: COMPAS
Project name: Compliance-driven Models, Languages, and Architectures for Services
Call and Contract: FP7-ICT-2007-1
Grant agreement no.: 215175
Project Duration: 01.02.2008 – 28.02.2011 (36 months)
Co-ordinator: TUV Vienna University of Technology (AT)
Partners: CWI Stichting Centrum voor Wiskunde en Informatica (NL)
UCBL Université Claude Bernard Lyon 1 (FR)
USTUTT Universitaet Stuttgart (DE)
TILBURG UNIVERSITY Stichting Katholieke Universiteit Brabant (NL)
UNITN Universita degli Studi di Trento (IT)
TARC-PL Apera sp. z o.o. (PL)
THALES Thales Services SAS (FR)
PWC Pricewaterhousecoopers Accountants N.V. (NL)

This project is supported by funding from the Information Society Technologies Programme under the 7th Research Framework Programme of the European Union.
Project no. 215175
COMPAS
Compliance-driven Models, Languages, and Architectures for Services
Specific Targeted Research Project
Information Society Technologies
Start date of project: 2008-02-01  Duration: 36 months

D3.1 Behavioral Model for Composite Services
Revision 2.0
Due date of deliverable: 2008-12-31
First submission date: 2008-12-31
Latest submission date: 2009-06-08

Organisation name of lead partner for this deliverable:
CWI Stichting Centrum voor Wiskunde en Informatica, NL

| Project funded by the European Commission within the Seventh Framework Programme |
| Dissemination Level | |
| PU | Public | X |
| PP | Restricted to other program participants (including the Commission Services) | |
| RE | Restricted to a group specified by the consortium (including the Commission Services) | |
| CO | Confidential, only for members of the consortium (including the Commission Services) | |
History chart

<table>
<thead>
<tr>
<th>Issue</th>
<th>Date</th>
<th>Changed page(s)</th>
<th>Cause of change</th>
<th>Implemented by</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>2008-10-29</td>
<td>All sections</td>
<td>New document</td>
<td>CWI</td>
</tr>
<tr>
<td>0.2</td>
<td>2008-11-19</td>
<td>Sections 1-3</td>
<td>Content added to Sections 1-3</td>
<td>CWI</td>
</tr>
<tr>
<td>0.3</td>
<td>2008-12-02</td>
<td>Section 4</td>
<td>Content added to Section 4</td>
<td>CWI</td>
</tr>
<tr>
<td>0.4</td>
<td>2008-12-04</td>
<td>Contents, Section 6</td>
<td>Content added to Section 6, document structure revised, lists of figures and abbreviations created</td>
<td>CWI</td>
</tr>
<tr>
<td>0.45</td>
<td>2008-12-07</td>
<td>Section 1, Section 2, Section 3</td>
<td>List of definitions created, Sections 1-3 revised</td>
<td>CWI</td>
</tr>
<tr>
<td>0.5</td>
<td>2008-12-11</td>
<td>Section 7</td>
<td>Content added to Section 7</td>
<td>CWI</td>
</tr>
<tr>
<td>0.6</td>
<td>2008-12-14</td>
<td>Section 5</td>
<td>Content added to Section 5</td>
<td>UCBL</td>
</tr>
<tr>
<td>0.7</td>
<td>2008-12-16</td>
<td>All sections</td>
<td>Proofreading and minor changes</td>
<td>CWI</td>
</tr>
<tr>
<td>0.8</td>
<td>2008-12-22</td>
<td>Section 5</td>
<td>Example added to Section 5</td>
<td>UCBL</td>
</tr>
<tr>
<td>0.9</td>
<td>2008-12-23</td>
<td>Title</td>
<td>Document renamed according to DoW</td>
<td>CWI</td>
</tr>
<tr>
<td>1.0</td>
<td>2008-12-29</td>
<td>All sections</td>
<td>Comments from RV-TUV-D3.1rev601 addressed</td>
<td>CWI</td>
</tr>
</tbody>
</table>

Authorization

<table>
<thead>
<tr>
<th>No.</th>
<th>Action</th>
<th>Company/Name</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Prepared</td>
<td>CWI</td>
<td>2008-12-12</td>
</tr>
<tr>
<td>2</td>
<td>Approved</td>
<td>TUV</td>
<td>2008-12-28</td>
</tr>
<tr>
<td>3</td>
<td>Released</td>
<td>TUV</td>
<td>2008-12-30</td>
</tr>
<tr>
<td>4</td>
<td>Approved</td>
<td>TUV</td>
<td>2009-06-08</td>
</tr>
<tr>
<td>5</td>
<td>Released</td>
<td>TUV</td>
<td>2009-06-08</td>
</tr>
</tbody>
</table>

Disclaimer: The information in this document is subject to change without notice. Company or product names mentioned in this document may be trademarks or registered trademarks of their respective companies.

All rights reserved.

The document is proprietary of the COMPAS consortium members. No copying or distributing, in any form or by any means, is allowed without the prior written agreement of the owner of the property rights.

This document reflects only the authors’ view. The European Community is not liable for any use that may be made of the information contained herein.
Contents

1. Introduction ................................................................................................................. 7
  1.1. Purpose and scope ................................................................................................. 7
  1.2. Document overview .............................................................................................. 7
  1.3. Definitions and glossary ....................................................................................... 8
  1.4. Abbreviations and acronyms ............................................................................... 9

2. Towards Automated Business Process Analysis .......................................................... 10
  2.1. Formal Process Analysis in the context of COMPAS ......................................... 12

3. Process Behavioral Models .......................................................................................... 13
  3.1. Business Process Modeling Notation (BPMN) .................................................. 14
  3.2. Business Process Execution Language (BPEL) .................................................. 16
  3.3. Unified Modeling Language (UML) .................................................................... 17
  3.4. Reo Coordination Language ............................................................................... 19
  3.5. Eclipse Coordination Tools for Business Process Modeling ............................ 22
  3.6. Mapping BPMN to Reo ...................................................................................... 24
  3.7. Mapping UML2 and WS-BPEL to Reo ................................................................. 34

4. Formal Behavioral Model for Service Composition: Extended Constrained Automata .. 35
  4.1. Constraint Automata .......................................................................................... 35
  4.2. Constraint Automata as Semantic Model for Reo ............................................. 37
  4.3. Practical Implications ......................................................................................... 38
  4.4. Web Services as Constraint Automata ............................................................... 39

5. Augmenting Process Behavioral Models with Data Domains ..................................... 41
  5.1. Data domains ...................................................................................................... 41
  5.2. Conversation models with data domains ........................................................... 43
  5.3. Compatibility and replaceability analysis ............................................................ 45

6. Compliance-aware Business Process Design in Examples ........................................... 47
  6.1. Control Flow Constraints ..................................................................................... 47
  6.2. Resource Constraints .......................................................................................... 49

7. Related work ............................................................................................................. 51
  7.1. Formal semantics and transformations of process modeling notations .......... 51
  7.2. Compliance-aware business process modeling .................................................. 53

8. Summary .................................................................................................................... 53

9. Reference documents ................................................................................................ 54
  9.1. Internal documents ............................................................................................. 54
  9.2. External documents ............................................................................................ 54
List of Figures

Figure 1: Compliance-aware business process modeling and verification with Reo/CA tools ........................................ 14
Figure 2: Subset of the BPMN modeling objects ........................................................................................................ 15
Figure 3: Basic elements of UML 2.0 Activity Diagram ......................................................................................... 18
Figure 4: UML 2.0 Sequence Diagrams .................................................................................................................... 19
Figure 5: Set of basic Reo channels .......................................................................................................................... 20
Figure 6: Examples of basic Reo connectors: Exclusive Router ................................................................................. 20
Figure 7: Examples of Reo connectors: Valve ................................................................................................................ 21
Figure 8: Examples of Reo connectors: K-counter ...................................................................................................... 21
Figure 9: Modeling BPMN events in Reo ................................................................................................................... 24
Figure 10: Representing BPMN messages in Reo ....................................................................................................... 24
Figure 11: Mapping of basic BPMN gateways to Reo ............................................................................................... 25
Figure 12: Mapping of OR decision/merge gateways .............................................................................................. 26
Figure 13: Modeling complex gateways: lossy m out of n join .................................................................................. 26
Figure 14: Modeling BPMN message flows in Reo ................................................................................................. 27
Figure 15: BPMN diagram for the Purchase-to-Pay scenario .................................................................................... 27
Figure 16: Reo model for a fragment of the Purchase-to-Pay scenario ................................................................. 28
Figure 17: Exception handling in a process consisting of n sequential atomic tasks ............................................ 28
Figure 18: Exception handling in a process consisting of n sequential sub-processes ............................................ 29
Figure 19: Exception handling in a process consisting of n parallel atomic tasks .................................................. 29
Figure 20: Exception handling in a process consisting of n parallel sub-processes .................................................... 30
Figure 21: Compensation association ....................................................................................................................... 30
Figure 22: Transaction consisting of n sequential tasks ............................................................................................ 31
Figure 23: I/O port hiding ......................................................................................................................................... 31
Figure 24: Transaction consisting of n sequential tasks whose cancelation occurs in the reverse order ................. 32
Figure 25: Compensation association with possible failure ...................................................................................... 32
Figure 26: Transaction consisting of n sequential tasks with possible failures in compensation activities .................. 33
Figure 27: Transaction consisting of the discriminator pattern with n parallel activities ........................................ 34
Figure 28: Constraint automata for basic Reo channels ............................................................................................. 37
Figure 29: Constraint automata for the on-line banking scenario .............................................................................. 40
Figure 30: Reo model for the BPMN compensation pair .......................................................................................... 40
Figure 31: Constraint automata for the BPMN compensation pair structure ............................................................ 41
Figure 32: Business protocol for the on-line banking scenario augmented with a timeout transition state ............... 43
Figure 33: Timed business protocol for the on-line banking scenario
Figure 34: Example of compatible business protocols
Figure 35: Modeling concurrent flow dependencies in Reo
Figure 36: Compensation in the processes with control flow dependencies
Figure 37: BPMN diagrams augmented with task authorization constraints
Figure 38: Reo process model satisfying the SoD (four eyes principle) constraint
Figure 39: BPMN diagram augmented with operation SoD constraint
Figure 40: Fragment of a Reo process model enforcing the SoD constraint
1. Introduction

Inadequate business process design and its further implementation by means of service composition without full and clear knowledge of service functionalities entail the danger that important norms and requirements on process behavior coming from legislative/regulatory documents and/or business contracts get violated or overlooked. This deliverable describes a formal model for unambiguous specification of business process behavior and verification of its compliance to formally expressed regulations. The main differences of our model compared to other available techniques for automated analysis of business processes and web service compositions is its unifying nature, extensibility and simplicity. Service behavioral descriptions and “glue code” are represented in a unified manner using a small number of modelling primitives. Due to the strong focus of the COMPAS project on compliance, we aim at enabling formal reasoning about various process properties including structural, temporal, transactional, resource-aware and Quality-of-Service (QoS) characteristics. Finally, our model has a simple graphical notation that can be successfully used by process designers without any prior experience in formal methods.

Additionally, we establish connection of our model and supporting verification tools with widely accepted business process modelling notations and service compositions languages by introducing corresponding converters which will be developed in the COMPAS project.

1.1. Purpose and scope

The purpose of this deliverable is to introduce a model and a framework that would leverage compliance-aware business process design and web service composition with formal reasoning and automated analysis. The model should be rigorous, expressive and extensible enough to incorporate various types of compliance constraints. Not less importantly, it should be easy to understand and use in practical applications.

This deliverable presents a graphical notation suitable for business process and web service composition modeling. This notation relies on the coordination language Reo and its formal semantics in terms of Constraint Automata (CA). We also establish connection between popular business process specification languages and Reo/CA coordination tools. Finally, we show by examples how compliance constraints can be modeled, verified and enforced with the help of our model.

1.2. Document overview

The rest of this document is structured as follows. In Section 2, formal behavioral modelling of web service compositions and their automated compliance analysis are discussed. In Section 3, we introduce a semantically unambiguous graphical language, Reo, and apply it in our context for process behavior specification. The main focus of this section is to illustrate how the objects and workflow patterns from widely-used business process modeling notations and web service composition languages, namely, BPMN, UML and WS-BPEL, can be represented and interpreted with the help of Reo. Additionally, software tools supporting our framework and enabling Model-Driven Software Development (MDSD) are introduced. Section 4 presents a formal behavioral model for compliance-aware web service composition. According to this model, web services are represented in the form of CA enriched with non-functional characteristics. In Section 5, business process models are extended with data domains. In Section 6, we provide examples of compliance-aware business process modeling.
and web service composition design in our framework. Finally, Section 7 compares our approach to related work.

1.3. Definitions and glossary

Business process: A structured collection of activities performed by human actors or software systems to achieve a certain business goal, together with their related resources (e.g. documents or data transformed during the execution of the process).

Business process activity: An atomic piece of work in a business process (e.g., approve a business trip), i.e. the smallest element in the model used to describe the work of involved actors. Activities are also called tasks or work items. Activities can be performed automatically (e.g. by a service) or manually (by humans) [D5.2].

Business process fragment: A process fragment is a connected sub graph of a process graph. It can also contain additional artifacts, such as variables, references to related processes, annotations etc. Some parts of a fragment may be explicitly stated as opaque, in order to mark points of variability and degree of freedom for reuse. So a process fragment is not necessarily directly executable [D5.2].

Business process model: An operative specification of a business process understandable by other humans or automated software tools.

Business process modeling: The act of representing existing or imaginary business processes in a form understandable by other humans or automated software tools.

Compliance: a process of ensuring that organization business processes, operations and practices are in accordance to a prescribed set of norms coming from legislative/regulatory documents and/or business contracts.

Exogenous coordination: Exogenous coordination refers to the ability, in a model or language, to coordinate the behavior of black-box entities (components, services, activities, etc.) from outside of those entities and without their knowledge about each other.

Formal process model: A formal specification of a business process.

Formal specification: A mathematical description of software that admits formal analysis and can be used for software implementation.

Formal verification: The act of proving or disproving the correctness of intended process behavior with respect to a certain formal specification or property using formal methods of mathematics.

Process modeling notation: A textual or graphical language used to create business process models.


Service coordination: Service coordination is a process of describing software systems by specifying interactions and dependencies among loosely-coupled services.

Web service: A Web service is a software application designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with a Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards. In this context, the service must have a published interface that can be invoked by the Internet [D5.2].
For technical definitions of Constraint Automata and related notions see Section 4.

1.4. Abbreviations and acronyms

[AD] Activity Diagram
[BDD] Binary Decision Diagram
[BPEL] Business Process Execution Language
[BPEL4People] BPEL Extension for People
[BPMN] Business Process Modelling Notation
[BSTL] Branching Time Stream Logic
[COMPAS] Compliance-driven Models, Languages, And Architectures for Services
[CA] Constraint Automata
[CARML] Constraint Automata Reactive Module Language
[CSP] Communicating Sequential Processes
[StAC] Structural Activity Compensation
[CTL] Computational Tree Logic
[FCL] Formal Contract Language
[GUI] Graphical User Interface
[GMT] Graphical Modeling Tools
[MDSD] Model-Driven Software Development
[OCL] Object Constraint Language
[QCA] Quantitative Constraint Automata
[QoS] Quality of Service
[LKS] Labeled Kripke Structures
[LTL] Linear Temporal Logic
[REALM] Regulations Expressed as Logical Models
[RSL] Reo Scripting Language
[RSTCA] Resource-aware Timed Constraint Automata
[RTOL] Real-time Temporal Object Logic
[SBVR] Semantics of Business Vocabulary and Rules
[SOA] Service Oriented Architecture
[SD] Sequence Diagram
[SoaML] Service oriented architecture Modelling Language
[SoD] Separation of Duty
2. Towards Automated Business Process Analysis

Compliance can be seen as a process of ensuring that an organization’s business processes, operations and practices are in accordance with a prescribed set of norms coming from legislative/regulatory documents and/or business contracts. These norms are very broad in nature. The survey in the area of compliance languages [D2.1] identifies five basic compliance concerns, namely, control flow, locative, information, resource and temporal requirements. A more detailed categorization of compliance regulations distinguishes monitoring, payment, privacy, QoS, retention, security and transaction compliance concerns. After interpretation in the context of each particular application, such norms result in various constraints on business process behavior and/or its non-functional characteristics.

Taking into account the complexity of modern cross-organizational business processes and the number of existing and ever changing regulations, it becomes a tedious task to verify whether such constraints hold for a particular process or supporting software. Therefore, it is highly desirable to leverage compliance-aware process development with formal methods enabling its automated analysis and verification. The key element in this problem is a usable, expressive and semantically precise model for specifying business processes and web service compositions. In this deliverable, we focus on the development of such a model.

Proliferation of new modeling languages requires significant standardization and dissemination efforts. Therefore, we aim at making our formal model compatible with the major existing industrial tools for software design by providing automated converters. Two most widely used graphical notations for business process modeling are BPMN and UML Activity Diagrams (ADs). Both notations are ambiguous, so the corresponding process diagrams require preprocessing and refinement before they become suitable for formal analysis and, finally, software implementation.

UML Sequence Diagrams (SDs) provide a conceptually different approach to system modeling. Since various scenarios involving interacting entities (e.g., auctions, service negotiation) are more convenient to represent in the form of sequence diagrams, we provide a converter from this notation to our behavioral model as well.

On the other hand, WS-BPEL is a language for describing executable business processes on top of WSDL service specifications, and, naturally, it has precise semantics. However, due to
the lack of graphical notation and the need to deal with implementation-level details, WS-BPEL is not suitable for domain analysis and conceptual business process modeling, although some efforts exist to adopt WS-BPEL for this purpose. Nevertheless, modern business processes are rarely developed from scratch. Thus, we must accommodate business process fragments with existing behavioral specifications in WS-BPEL that can be composed together and become a basis for a new process implementation. Therefore, in this deliverable, we aim at mapping the main WS-BPEL constructs to our formal behavioral model, and in this way enable formal verification of WS-BPEL process fragments as well as new composite processes with respect to compliance specifications.

The aforementioned notations lack support for compliance. However, both industry and academia have proposed numerous extensions for process compliance support on top of these languages. For instance, in [ADW08] BPMN processes are annotated with QoS information, in [WS07] a BPMN extension for specifying task authorization constraints is introduced, while in [LMX07] a language for specifying regulatory compliance on top of WS-BPEL is proposed.

Formal structures such as Petri-nets, transition systems or process algebras have been widely employed to model business processes [DDO08, WG08, WG08a] and web service compositions [HB03, YTX05], while various types of logic-based formalisms (First-Order Logic [DF07, HW03], Linear Temporal Logic (LTL) [LMX07], Computational Tree Logic (CTL) [MK+03, KTK02], deontic logic [SGN06, CCD+07], temporal deontic assignments [GV06], concurrent transaction logic [MDK03], etc.) have been applied to specify process constraints and compliance obligations. However, existing approaches are rather disparate and oriented toward process verification against a limited set of requirements representing particular compliance categories, e.g., control flow and temporal constraints [GMS06, GK07], security requirements expressed in terms of permissions/prohibitions/obligations on executable actions [BCC+07] or privacy policies [BDM+06, HBP07, MBS+08].

We aim at developing a unified extensible behavioral model that is able to incorporate various types of compliance constraints. Our solution is based on Constraint Automata (CA) which offers an operational model for specifying composite service behavior. CA is essentially a variant of a labeled transition system where transitions are augmented with pairs of synchronization and data constraints. The states of a CA stand for the process configurations while transition labels can be viewed as input/output operations performed in parallel (more precisely, sets of nodes where data flow is observed in parallel and boolean conditions on the data items observed on those models). This model is fully compositional, and can express arbitrary mixes of synchronous and asynchronous communication. CA were developed as a semantic model for the coordination language Reo [Arb04s] and later have been extended to express time dependent behavior, probabilistic and stochastic systems.

There are several reasons that motivate our choice of Reo and CA for specifying business process and web service composition behavior. First, Reo allows for specifying both choreography and orchestration of process activities as well as internal and external behavior of involved services in a unified formalism [SA07]. Second, Reo has a simple graphical notation which makes it applicable in practical applications by process designers without any prior experience in formal methods. As we show later in this deliverable, a small number of Reo modeling primitives (channels) is sufficient for representing rather complex behavioral protocols. Moreover, precise semantics of Reo in terms of CA enables automated process verification. Finally, the COMPAS service behavioral model should be flexible enough to incorporate various types of information (e.g., temporal requirements, data flow constraints, QoS characteristics) about services required to analyze process compliance to regulatory norms and con-
tract-driven constraints. Reo process models can be automatically translated into CA which are suitable for representing service compositions with QoS guarantees [ACS+07], time- and resource-aware processes [ABB+07, SA07a]. Moreover, CA can be further extended by associating various properties with states and transitions (e.g., with privacy policies [HBP07]).

2.1. **Formal Process Analysis in the context of COMPAS**

According to [DoW], the main goals of Work Package (WP) 3 are as follows:

1. Design an expressive formal model for specifying behavior of web services and service compositions.
2. Incorporate mechanisms in the model to express user goals, business constraints, and regulatory policies, in addition to and along with service interaction protocols.
3. Develop tools to support definition, formal verification, and reuse of process behavioral models.
4. Develop tools to analyze and manage behavior in compliance with policies.
5. Map the formal models for compliant service composition to the MDSD and runtime infrastructure.

This deliverable covers the first task of WP3 and provides insight on how goals 2-5 will be achieved with the help of the proposed formal model.

Below we set the content of this deliverable in the context of COMPAS architecture by establishing the relation of WP3 to other WPs.

**Relation of WP3 to WP1.** Different architectural views introduced in WP1 help to capture various aspects of service-oriented systems supporting business processes and clearly relate them to the corresponding compliance concerns. For example, if a certain norm regulates human involvement within a process or limits user access to private client data, it will affect a human view and/or a data flow view of the system. If required, formal methods can be employed to guarantee compliance of each of these views to corresponding formally expressed regulations. For instance, a control flow view may contain formal specifications of a process control flow which are verified against structural compliance requirements such as prescriptions to include particular activities (e.g., logging or monitoring) into the process. Moreover, WP1 establishes different scenarios such as green-field development or legacy software adjustment where the problems of compliance checking and enforcement may arise. We show how these scenarios can be addressed from the perspective of formal methods. We aim at dealing with both design-time notations (e.g., BPMN, UML) and executable languages (WS-BPEL) representing process fragments.

**Relation of WP3 to WP2.** The problem of business process design and compliance specification should be considered at two different levels of abstraction. At a higher level, user-friendly (graphical) notations are used to enable fast conceptual modeling of intended process behavior and identification of related compliance requirements with respect to this process. At a lower level, more detailed and technically sound models for specifying process behavior and compliance constraints should be employed. WP2 aims at providing expressive languages for compliance concerns and defines a query language to extract process fragments augmented with compliance information from a repository developed in WP4. WP3 deals with process design at the technical level and enables formal analysis of extracted process fragments and their composition by verifying its compliance to “global” constraints that may not be applicable to each fragment, but should, nevertheless, be enforced in the whole process.
Decomposition of goal-oriented concepts and high-level compliance policies introduced in WP2 to formally expressed low-level constraints on control/data flow verifiable in WP3 will be discussed in more detail in the forthcoming deliverables D3.2 and D2.3.

Relation of WP3 to WP4. Web services are generally provided by different vendors, therefore testing and compliance analysis of service compositions become a major problem. WP4 aims at creating a repository of process fragments augmented with compliance information which will be queried with the query language developed in WP2. WP3 enables formal analysis of a composition of the extracted process fragments and verifies its compliance to “global” constraints that may not be applicable to each fragment individually, but should be enforced in the whole process. Moreover, the Eclipse Coordination Tools (ECT) adopted in WP3 may serve for editing and testing of such process fragment compositions [AKM08].

Relation of WP3 to WP5. WP3 is complimentary to WP5 as the former aims at compliance-aware process design while the latter focuses mainly on run-time compliance monitoring. Additionally, our formal model can be used to visualize running business processes, display problems detected in WP5 and redesign the process to guarantee its further compliance.

3. Process Behavioral Models

The overall vision of our compliance-aware business process modelling framework is shown in Figure 1. We assume that business analysts use traditional notations for creating business process models such as BPMN or UML2 ADs. Alternatively, notations more specific for SOA, among which are WS-BPEL Graphical Modeling Tools (GMT) http://www.eclipse.org/bpel/ and Service-oriented architecture Modeling Language (SoaML) http://www.omg.org/docs/ad/08-08-04.pdf can be utilized. At this level, compliance concerns can be expressed using Domain Specific Languages (DSL) or GMT extensions; see [McC89] and [ADW08, LMX07] for examples of these two approaches. One of the goals of WP2 is to develop DSLs capable of expressing major categories of compliance concerns. These models will not necessarily guarantee the level of precision sufficient for the direct process implementation. Therefore, in WP3 we introduce an intermediate layer on which the high-level models will be verified and refined. The basic semantics of this layer is defined by Reo, a channel-based exogenous coordination language supported by a graphical editor, animation plug-in, model checking tools and code generation engines. These tools allow us to use Reo both for graphical process modelling and for formal process verification before its actual implementation. We assume that at this step, compliance rules are converted into automata transition constraints, temporal logic formulae, or result in statements whose validity can be verified through automata state reachability checking.

After model checking and refinement, the Reo/CA process models can be automatically translated into executable SOC languages such as WS-BPEL, as well as to Java code.
3.1. **Business Process Modeling Notation (BPMN)**

The Business Process Modeling Notation (BPMN) is a standard widely-accepted graphical notation for business process modeling. According to this notation, a process can be represented in the form of activities performed either by humans or software applications, important events occurring in the process, and the control flow over the involved activities. One of the reasons why BPMN stands out among other notations for business process modeling is its ability to define concurrent tasks and sub-processes with exception handling and compensation associations, which have been proven to be useful even at the stage of early design. As a trade-off to its expressive power, BPMN lacks semantic precision.

The basic BPMN concepts are *flow objects, connecting objects, swimlanes* and *artifacts*. Flow objects are the main graphical elements defining the behavior of a business process. BPMN distinguishes three types of flow objects, namely, *events*, *activities* and *gateways*. These elements are linked according to well-defined syntactic rules by two *connecting objects*: *sequence flow* and *message flow*. A third connecting object is *association* and it is used to connect flow objects with text and non-flow elements. Two types of *swimlanes*, *pools* and *lanes*, arrange the main BPMN elements into groups. Finally, *artifacts* are introduced to provide additional information about a process. This concept is extendable and besides the three standard artifacts of *data objects*, *groups*, and *annotations*, designers can introduce their own artifacts as well.
**Figure 2: Subset of the BPMN modeling objects**

Figure 2 shows the selected BPMN elements that are essential for modeling process behavior. BPMN identifies three types of events: a **start event** signals the start of a process, an **end event** signals the end of a process and an **intermediate event** is an event occurring during a process. Different triggers such as **message**, **timer**, **rule**, **link**, **error**, **cancel**, **compensation**, **terminate** and **multiple trigger** can be associated with events.

A start event with a **message trigger** indicates that the process starts after a message has been received. An end event with a message trigger indicates that the process finishes by sending a message. An intermediate event with a message trigger indicates that the process either receives or sends a message to another participant. A **timer trigger** implies that the normal or exception flow of the process is initiated (or continued) only when a specific time-date has been reached. A **rule trigger** event occurs if some context-dependent process condition is met.

A **link trigger** is used to connect the end of one process to the start of another. Paired intermediate link events correspond to the "go to" operation. A **multiple trigger** shows that there are multiple ways of triggering the process, and only one of them is required to proceed. An event with an **error trigger** is used to throw an exception if it is a part of the normal flow and to catch an exception if it is attached to the boundary of an activity. A **cancel trigger** event is used to indicate that a transaction sub-process should be canceled either because an internal failure has occurred or a cancel message has been received. A **compensation trigger** as a part of the normal flow indicates that compensation is required. When attached to the boundary of an activity, it reacts to a named compensation call. Finally, a **terminate trigger** indicates that all activities in the process should be immediately ended, and the process is ended without compensation or event handling.
An activity can be an atomic task or a sub-process. To each task a type can be assigned. Among the specific task types are service, receive, and send tasks. A sub-process is a compound of other activities plus a sequence flow over them. BPMN introduces two attributes that are commonly used to identify special types of activities (both tasks and sub-processes), namely, looping activities and multiple concurrent instances of the same activity.

A gateway is a construct used to control divergence and convergence of the sequence flows. A parallel fork gateway is used to split an incoming sequence flow into several concurrent branches, while a parallel join gateway synchronizes several concurrent sequence flows. A data/event-based XOR decision gateway selects one out of a set of mutually exclusive sequence flows according to some data-based condition or external event. An OR decision behaves similarly but it allows more than one alternative to be selected. An OR merge shows the convergence of several sequence flows into one sequence flow. Finally, complex decision/merge gateways are used to cover the advanced sequence flow control constructs that cannot be easily handled using other gateways. One such example is the so called m out of n choice when m arrived tokens out of n initiated parallel sequence flows are required to continue the process.

BPMN distinguishes two basic types of flow. The sequence flow prescribes the order of activities performed by one entity while the message flow regulates the flow between two communicating entities represented by separate pools. The sequence flow consists of a normal flow to which a transition guard can be assigned (uncontrolled, conditional or default flow) and exception flow that originates from some event and is used to handle exceptions. Finally, BPMN defines a number of advanced constructs such as compensation association and 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.

3.2. Business Process Execution Language (BPEL)

The WS-BPEL process model is layered on top of the service model defined by WSDL. A business process defines how to coordinate the interactions between a process instance and its partners (a participant, which itself is a process instance). In this sense, a WS-BPEL process definition provides the description of the behavior and interactions of a process instance relative to its partners and resources. Such a business process can be described in two different ways: either as an abstract business process or as an executable business process. Abstract business processes are partially specified processes that are not intended to be executed. Executable business processes model actual behavior of a participant in a business interaction.

An executable process definition consists of several parts describing partner links, process variables, correlation sets, main process workflow, fault and compensation handling activities. The partner link declarations are used to define the relation between the process and its partners. It defines the role of the process in this relation (consumer or provider of an interface), and the interfaces used/provided by that role. The interfaces, operations, as well as their
parameters and types, are specified in the corresponding WSDL documents. The *process variables* are used to represent the state of the business process, they contain the information received from or sent to the partners of the process. The variables may be of primitive data types (e.g., strings, boolean, integers) or of some complex types defined in a WSDL document. The *correlation sets* define the parts of message data that are used to associate and route a particular message to a particular instance of the business process. Such information tokens uniquely identify the instance of the business process. The *process flow* is defined by a set of process activities. They specify the operations to be performed, their ordering, conditional logic, reactive rules, etc. We distinguish the following groups of activities: *basic activities*, *structured activities*, and the specific operational blocks, namely *fault* and *compensation handlers*.

*Basic activities* represent primitive operations performed by the process, such as message emission/reception (*invoke*, *receive*, and *reply* activities), data modification (*assign*), process termination (*terminate*), waiting for a certain period of time (*wait*), or doing nothing (*empty*).

*Structured activities* define the order in which a collection of activities occurs. They compose the basic activities into structures that express their control flow patterns. The structured BPEL activities include *sequence*, *switch*, and *while* that model traditional control constructs; *pick* that models nondeterministic choice based on external events (i.e., message reception or timeout); *flow* activity that models parallel execution of the nested activities. The structured activities can be recursively nested and combined.

*Fault handling* in BPEL is thought of as a mode switch from the normal processing. It aims at undoing the unsuccessful work. The fault may arise on reception of the fault message, or on explicit invocation of a throw activity. The fault handler declaration specifies the activities to be performed when a fault arises. The *compensation handlers* are used to reverse the effect of some unit of work that has completed with a fault. The compensation is always initiated within a fault handler, and may also require a compensation of some nested, previously successful, activities. A compensation handler is always associated with a work unit (scope), and is invoked (explicitly or implicitly) using the BPEL compensate activity.


### 3.3. Unified Modeling Language (UML)

UML, a de-facto standard for software modeling, is a multi-purpose language offering a variety of notations for modeling different aspects of software structure and behavior. One of these notations, UML Activity Diagrams, is often used to capture the business control flow by depicting what activities are performed, in what order and under what conditions. Another notation, UML Sequence Diagrams, is typically employed to describe object-oriented software systems, but also to design system architectures, model dynamic process flows and analyze interaction protocols. UML comes along with the Object Constraint Language (OCL) [http://www.omg.org/docs/ptc/03-10-14.pdf](http://www.omg.org/docs/ptc/03-10-14.pdf), a formal declarative language used to impose constraints on UML models. OCL expressions typically specify invariant conditions that must
hold for the system being modeled or queries over objects described in a model. Additionally, extensions to OCL dealing with time-aware constraints [Fla03, HHM04], special purpose UML profiles such as one for modeling QoS and fault tolerance characteristics [http://www.omg.org/docs/ptc/04-06-01.pdf] and Semantics of Business Vocabulary and Rules (SBVR) language [http://www.omg.org/docs/dtc/06-03-02.pdf] can be helpful in expressing compliance requirements.

3.3.1. Activity Diagrams (ADs)

The main elements of UML ADs are shown in Figure 3. The fundamental behavioral unit is an action. Three special types of actions are distinguished, namely, invocation action, used for performing action calls, signal sending and event accepting; read/write action, used for accepting and modifying objects and their values; and computation action, transforming input values to output values. Actions within a control flow defined by means of connecting objects (edges) and control nodes form activities. UML ADs support four basic control nodes, namely, forks, denoting the beginning of parallel processing, join, denoting the end of parallel processing, decision, denoting conditional flow splitting, and merge, implying that one or more incoming flows must reach this point before processing continues, based on the guards on the outgoing flow. Additionally, there are two types of nodes that simplify reading of activity diagrams, namely, initial node, indicating the start of the control flow, and final nodes, indicating the end of flow/diagram. Similarly to BPMN, UML ADs can be organized into partitions, also called swimlanes, indicating who/what is performing the activities. Sub-activity indicators can be employed to show that an activity is described by a more finely detailed activity diagram.

![Figure 3: Basic elements of UML 2.0 Activity Diagram](image)

3.3.2. Sequence Diagrams (SDs)

The goal of UML SDs is to model dynamic system behavior in terms of entities, components/services or objects that exchange messages or functional calls. The diagram conveys the information along the horizontal and vertical dimensions: the vertical dimension shows the time sequence of messages/calls as they occur, and the horizontal dimension shows with the help of lifelines, the object instances that the messages are sent to.

Figure 4 provides an example of SDs which describes the interactions in the login phase of an on-line banking scenario. A UML SD is represented as a rectangular frame labeled by the keyword sd followed by the interaction name. The vertical lines in the SD represent lifelines for the individual participants in the interaction. Interactions among participants are shown as arrows representing messages. It can be either asynchronous (represented by an open arrow head) or synchronous (represented by a filled arrow head). UML2.0 SDs specification introduces some special kinds of messages such as e.g. lost and found messages, however, since
we are mostly interested in the modeling of interaction between components/services at the conceptual level, we will consider only a basic subset of the UML 2.0 SDs.

![Figure 4: UML 2.0 Sequence Diagrams](image)

### 3.4. Reo Coordination Language

*Reo* [Arb04] is a channel-based exogenous coordination model wherein complex coordinators, called *connectors*, are compositionally constructed from simpler ones. We summarize only the main concepts in Reo here. Further details about Reo and its semantics can be found in [ABB+07, Arb04, AR03, BSA+06].

Complex connectors in Reo are formed as a network of primitive connectors, called *channels*, that serve to provide the protocol which controls and organizes the communication, synchronization and cooperation among the services that they interconnect. 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 either sinks or sources. Reo places no restriction on the behavior of a channel and thus allows an open-ended set of different channel types to be used simultaneously together.
Figure 5: Set of basic Reo channels

Figure 5 shows the graphical representation of basic channel types in Reo. A **FIFO1 channel** represents an asynchronous channel with one buffer cell which is empty if no data item is shown in the box. 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 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 \( P \subseteq \text{Data} \) 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 \( d \in P \).

There are some more exotic channels permitted in Reo: (A)**synchronous drains** have two source ends and no sink end. A synchronous drain can accept a data item through one of its ends iff a data item is also available for it to simultaneously accept through its other end as well, and all data accepted by this 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 [ABB+07].

Complex connectors are constructed by composing simpler ones via the **join** and **hiding** operations. **Join** plugs to channel-ends together creating a node at the point of connection. To this node one can connect more channels via join afterwards. If more than one accepting channel-ends are connected to a node every incoming message is simultaneously written to all outgoing channels whenever all outgoing channels at the node are ready to accept data. Whenever more than one channel-end offers data at a node a non-deterministic choice decides which data 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.
**Example 1 (Exclusive Router).** Figure 6 shows an implementation of an exclusive router by composing five synchronous channels, one synchronous drain and two lossy synchronous channels together. 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 one is selected non-deterministically. To avoid writing an exclusive router every time it is used, we introduce a notation similar to a node to represent this connector. Informally, the behavior of the service composition shown in Figure 4 is as follow: after Service1 has performed an output, either Service2 or Service3 is executed depending on which of them is currently available.

We will also use XOR-nodes with more than two outputs. Such a connector can be defined by combining several two-output exclusive routers. 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 enabled.

**Example 2 (Valve).** A valve connector shown in Figure 7 is able to close and reopen the flow from $A$ and $B$. Initially, the circuit is in the “open” state, i.e., a data item arriving at the input port $A$ flows to the output $B$ until the close command arrives. After that, the circuit goes into the “close” state, i.e., the flow remains blocked until the open command arrives.

![Figure 7: Examples of Reo connectors: Valve](image)

**Example 3 (K-counter).** A $K$-counter shown in Figure 8 is a useful connector essentially consisting of $K+1$ FIFO1 channels whose data flow regulated by means of $K$ synchronous drains. After being initialized, the connector “counts” to $K$ by accepting incoming messages though the port “increase”. Each such message allows the data token to pass to the next buffer.

![Figure 8: Examples of Reo connectors: K-counter](image)

In the context of COMPAS project, we use Reo to specify business process behavior and obtain a glue code to coordinate web services realizing this behavior. The semantic model of Reo in terms of CA gives a formal model of the process control/data flow. This approach has some advantages over existing formalisms, most notably, Petri-nets, logic-based process specification approaches and process algebras. In particular, as shown in [Cla07], Reo is more
expressive than Petri-nets. Regarding our application, due to the combination of synchrony and asynchrony Reo is more suitable for specifying exception and compensation handling in business process flow than 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 multiple compliance categories as required by COMPAS case studies. Compositionality and extensibility are two major features that justify our choice of Reo as foundation for COMPAS. Among the most significant pragmatic benefits is the existence of software tools developed by CWI which can be easily adjusted and modified to enable compliance-aware process analysis.

3.5. Eclipse Coordination Tools for Business Process Modeling

Eclipse Coordination Tools (ECT) http://reo.project.cwi.nl/ represent a set of plug-ins on top of the Eclipse platform http://www.eclipse.org. Additionally, other plug-ins including ones for business process design and execution, have been developed for Eclipse. Here we briefly overview tools we exploit for compliance-aware business process design and analysis in the COMPAS project.

**Eclipse BPMN modeler.** The BPMN modeler http://www.eclipse.org/stp/bpmn/ 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.

**Eclipse UML2 modeler.** The Eclipse UML2 modeling toolset http://www.eclipse.org/modeling/mdt/?project=uml2 provides a useable implementation of the UML meta-model to support the development of modeling tools, a common XMI schema to facilitate interchange of semantic models, test cases as a means of validating the specification and validation rules as a means of defining and enforcing levels of compliance.

**Eclipse BPEL designer.** The Eclipse BPEL designer http://www.eclipse.org/bpel/ is a tool providing a comprehensive support for the definition, authoring, editing, deploying, testing and debugging of WS-BPEL 2.0 processes.

**BPMN2Reo converter.** The 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. Section 3.6 discusses the mapping of BPMN to Reo in more detail.

**UML2Reo converter.** The mapping of UML ADs to Reo is similar to the mapping of a subset of BPMN to Reo. Therefore, we focus mainly on a converter from UML SDs to Reo and CA. The theoretical basis for this work is given in [AS08].

**BPEL2Reo converter.** BPEL2Reo converter http://ece.ut.ac.ir/msirjani/B2ReoTool/B2R.jar is a tool provided by the University of Tehran for converting BPEL process specifications to Reo. The theoretical basis for this work is given in [TVM+08]. Currently there is an ongoing work on a new version of this tool and its integration into Eclipse platform.

**Reo graphical editor.** One of the key tools in our framework is the Reo graphical editor. The editor enables business process modelling by simple drawing operations and serves as a bridge to a number of other tools that can be either invoked from the context menus or directly interact with it. Process models are saved using an XML format. The model classes and
the parser for this format can be used also in stand-alone applications, which is possible for most of the tools presented here.

**Reo reconfiguration plug-in.** 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 [KLA08] 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 plug-in.** Business process analysis and simulation can be accomplished using a tool that generates flash animated simulations of Reo connectors. The plug-in depicts the connector that was previously shown in the editor in the animation view. The parts of the connector highlighted red represent synchronous data flow. Tokens move along these synchronous regions. On the left side there is a list of possible animations for this connector and the attached writers and readers. The underlying formalism for the synthesis of such animations is an extension of the coloring semantics of Reo [CCA07]. 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.

**Core automata tools.** The core automata tools include a graphical editor and a conversion tool to generate extended CA from Reo connectors. A detailed description of CA is given in Section 5 of this deliverable.

**Reo validation plug-in.** Reo validation plug-in is a tool that performs model checking over coordination models represented as constraint automata. Our model checker uses a symbolic model and a CTL-like logic as specification format. Properties like deadlock-freeness and behavioral equivalence of connectors can be verified. The model checker for Reo, called Vereofy [http://www.vereofy.de/], has been implemented at TU Dresden in the context of EU CR EDO project [http://www.cwi.nl/projects/credo/ and is integrated into the Eclipse Coordination Tools as a model checker view.

**Code generation tools.** Reo process models can be automatically converted to executable Java code. Similar to the model checker, the code generation is a two-step process. First, each Reo connector is translated to a constraint automaton. Then, the automaton is used to generate a Java state-machine coordinator. A runtime library implements constraint automaton ports as blocking zero-length buffers. Additionally, a graphical wiring editor allows users connect Java components to the coordinator by pairing corresponding ports together. Although the existing code-generator is limited to Java, a new retargetable generator capable of generating C, BPEL etc. is currently under construction. The new framework also supports loading and interpreting CA at runtime, which is useful for deploying Reo coordinators in Java application servers. The Distributed Reo Engine (REDrum) [http://reo.project.cwi.nl/cgi-bin/trac.cgi/reo/wiki/Tools#DistributedReoEngine is a tool that can generate Java code for executing Reo connectors on different machines. This engine can be used to generate distributed business processes from Reo models annotated with deployment information.

Examples of using these tools for web service composition and mash-up building can be found in [LA07, AKM08, MLA08].
3.6. Mapping BPMN to Reo

3.6.1. Basic Objects: Tasks, Events, Gateways and Message Flow

BPMN tasks and sub-processes correspond to external components or black-boxes whose collaboration is coordinated by Reo. However, it is still possible to simulate the behavior of certain activities using Reo channels. For example, an atomic task with one input and one output can be represented by a simple FIFO1 or a timer channel while a sub-process can be modeled by a Reo connector that preserves the number of its incoming and outgoing flows.

An event with no trigger (start, end, or intermediate) or an end event with a terminate trigger can be shown as a Reo node (source, sink, or mixed). Other event triggers can be modeled using the basic Reo channels. Thus, (i) a timer event can be represented with the help of a timer channel, (ii) an incoming message event can be simulated by a synchronous drain whose first end is an input port and its second end is an internal process node (see Figure 9), while an event with a rule trigger corresponds to a filter channel with an appropriate transition condition.

![Figure 9: Modeling BPMN events in Reo](image)

Other BPMN events such as outgoing messages, error, compensate, cancel, or link events occurring as a part of the sequence flow correspond to the immediate transitions into required places of the process where they will be triggered and can be represented by means of synchronous channels. However, if a process or a sub-process that must react to such an event is not ready to accept it, the current sequence flow will be blocked. This problem can be resolved by using either a lossy synchronous channel which indicates that if an event is not picked up at the destination point it will be lost, or a FIFO1 channel which indicates that a message generated by an event will wait until it can be processed. Figure 10 shows the Reo connectors corresponding to these three message sending protocols.

![Figure 10: Representing BPMN messages in Reo](image)

Composite conditions such as the case where the process execution continues when a required message has arrived or a certain deadline has been reached can be modeled by the combination of several Reo channels.

Figure 11 shows the Reo connectors for the basic BPMN gateways, namely, data-based XOR decision, event-based XOR decision, XOR merge, parallel fork and parallel join. A data-based XOR decision is modeled using a synchronous channel which represents the incoming flow and two (or more) filter channels with a common source that represent the alternative outgoing flows. Filter transition conditions (guards) are defined by boolean expressions $g_1$.
and \( g_2 \). The representation of an event-based XOR decision mainly depends on the semantics of the events that affect the decision. In our case, the lower branch is selected if a message arrives in a predefined period of time, and the higher branch is preferred otherwise. An XOR merge consists of two (or more) synchronous channels with a common sink. A parallel fork is composed of two (or more) diverging synchronous channels. A parallel join consists of two (or more) synchronous channels representing the incoming parallel sequence flows that are further synchronized with the help of a synchronous drain channel. Several lossy synchronous channels with a common sink can then used to get a single outgoing token by choosing one of the incoming data items non-deterministically. Since Reo deals with data flow observed on the channel ends, other variants of parallel join gateway are possible, namely, we can assume that the outgoing flow will need only a data item received from one of the incoming branches, or a new data item will be created as a function of data items received from all incoming branches.

\[
\begin{align*}
\text{Data-based OR/XOR decision} & \\
\text{Event-based XOR decision} & \\
\text{OR/XOR merge} & \\
\text{Parallel fork} & \\
\text{Parallel join} & \\
\end{align*}
\]

**Figure 11: Mapping of basic BPMN gateways to Reo**

An OR decision gateway can be modeled in Reo similarly as the data-based XOR decision whose guards are not necessarily mutually exclusive. Figure 12 shows the mapping of a BPMN sub-process consisting of three tasks, \( T_1, T_2 \) and \( T_3 \), its Reo model and execution simulation. In this case, if both branches have been activated, we assume that the task \( T_3 \) has to be executed twice. This behavior can be achieved by indicating that the circuit output port accepts more than one data item using the `reader` property: value of 2 indicates that two data items are accepted, while the value of -1 means that there is no limitation on the number of accepted data items.

\[
\begin{align*}
\text{a) BPMN model} & \\
\end{align*}
\]
b) Reo process model

**Or decision/merge (Network)**

Additionally, using Reo, the designer can define various complex control gateways. For example, Figure 13 shows a connector for an *m out of n* synchronizer pattern. This is a *lossy* version of the pattern, that is, the circuit loses its extra inputs before the next cycle. Alternatively, by substituting *n* lossy synchronous channels introducing the input data with *n* simple synchronous channels one can create a *sparing m out of n* pattern that delays to spare its extra inputs for the next cycle.

**Figure 13: Modeling complex gateways: lossy m out of n join**

In BPMN a message flow is used to show the flow of messages between two entities that are prepared to send and receive them. Therefore, by default we can represent the BPMN message flow using synchronous channels. However, BPMN does not aim at specifying any further details about entity communication except perhaps in textual annotations. In contrast, the Reo syntax enables the process designers to model this aspect. Thus, one can differentiate synchronous and asynchronous message exchanges. In the former case, the sequence flow is blocked until the acknowledgement message is received. In the latter case, other activities can
be performed while waiting for an acknowledgement message. Figure 14a shows a synchronous version of a Send/Receive Order scenario while Figure 14b demonstrates how the asynchronous messaging can be represented in Reo: after sending a message $M_1$ the entity can perform activities of the sub-process $P$ until a reply message $M_2$ is received. Here we assume that the output of the exclusive router being opened by the message $M_2$ has priority and a token will successfully leave the cycle. We use a small exclamation mark to show the prioritized output in the figure.

![Synchronous message exchange](image)

**a) Synchronous message exchange**

![Asynchronous message exchange](image)

**b) Asynchronous message exchange**

Figure 14: Modeling BPMN message flows in Reo

**Example (Purchase Order Scenario)**

Figure 15 shows a BPMN diagram for the Purchase-to-Pay scenario within a procurement application discussed in [SGN06] while Figure 16 presents the corresponding annotated Reo model for its fragment. 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 define or affect the semantics of the circuits.

![BPMN diagram](image)

**Figure 15: BPMN diagram for the Purchase-to-Pay scenario**
3.6.2. Processes with Exception Handling

As the above example shows, the translation of BPMN processes without exception handling into Reo circuits is rather straightforward. The occurrence of an exception event within a subprocess interrupts the execution of the sequence flow and spawns the exception flow that often affects other sub-processes and must be appropriately handled. There are two major issues here:

- Interrupt a sub-process at any point of its execution,
- Clean all tokens/data in the circuit including those used to propagate exception events.

The composition of Reo connectors implementing these issues depends on the structural aspects of sub-processes. We consider four basic constructs, namely,

- sequential execution of atomic tasks,
- sequential execution of sub-processes,
- parallel execution of atomic tasks, and
- parallel execution of sub-processes.

Figure 17 depicts a Reo circuit that simulates the execution of a process $P$ consisting of $n$ serial atomic tasks. The normal flow traverses tasks $(T_1, T_2, \ldots, T_n)$ from the start to the end. Each two neighbor tasks are interconnected using an exclusive router with priority that is used to interrupt the process. Another exclusive router is used to direct a cancel message into the point where the execution token currently resides. The cancel message opens the output of the prioritized exclusive router and two tokens fire in the corresponding synchronous drain. Simultaneously, the cancel message is directed to the exception output which signals that the process has been interrupted.

---

**Figure 16: Reo model for a fragment of the Purchase-to-Pay scenario**

**Figure 17: Exception handling in a process consisting of $n$ sequential atomic tasks**
In the above circuit we assumed that an atomic task, once invoked, always completes successfully. This may not be the case for some activities. Figure 18 depicts a Reo circuit that simulates the execution of a process consisting of $n$ serial sub-processes ($T_1$, $T_2$, ..., $T_n$). Each sub-process $T_i$ can be interrupted from outside by a cancel message or can generate an internal exception. The exception handling in the former case is analogous to the case of atomic tasks. In the latter case, the exception flow originating from a sub-process is redirected to the exception output of the process $P$.

Figure 19 shows a process consisting of $n$ parallel atomic tasks. This Reo circuit is essentially composed of a parallel fork and a parallel join gateways with $n$ outgoing and $n$ incoming branches, respectively. When a task $T_i$ is completed, the corresponding token waits in the FIFO1 channel until other tasks are completed as well. After that, the token flows to the circuit output. For interrupting the process, the cancel message is directed to each of the prioritized exclusive routers. FIFO1 channels are used to avoid synchronization of task cancelations. Indeed, the fact that some tasks were not completed when a cancel message has arrived should not prevent the interruption of the tasks in other branches. Additionally, the cancel message is directed to the exception output to signal the interruption of the process $P$.

If an internal exception occurs in a sub-process that is executed in parallel with other sub-processes within a process, this exception should be propagated to all other branches in order to interrupt them as well. Figure 20 shows how a Reo connector looks in this case. In each branch, an additional exclusive router is employed to propagate a cancel message (originating either from an internal or external event) to an executing sub-process or to the point where the token waits for synchronization with other sub-processes.
Figure 20: Exception handling in a process consisting of n parallel sub-processes

In the Reo connectors for processes with sequential activities we assumed that once a process has been invoked it will not be invoked again until the first invocation has completed. Such mutual exclusion behavior can be ensured by a FIFO1 channel whose source end coincides with the start state of the process and whose sink end is connected using a synchronous drain with the end state of the process. When a process is invoked, one token flows into the FIFO1 channel and waits until the execution reaches the end state, thus, preventing other tokens from entering the circuit through the process input port. It is easy to see that this assumption can be relaxed without significant changes in the circuit behavior. The main difference is that in this case a cancel message will choose one of the executions non-deterministically and stop it without affecting the others.

3.6.3. Transactional Processes

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 $T \triangleright \sim T$, can be represented as shown in Figure 21. 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.

Figure 21: Compensation association
Figure 22 shows a Reo model for a transactional process $P = C_1, C_2, ..., C_n$ consisting of a set of sequentially executed compensatable activities. Here, each connector $C_i = T_i \sim T_i$ 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 Figure 23. 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.

![Figure 22: Transaction consisting of n sequential tasks](image1)

![Figure 23: I/O port hiding](image2)

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 $C_i$ 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, Figure 24 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 $C_i$ only if the circuit $C_i$ produces an output signaling that its task has been compensated for.

![Diagram of a transaction consisting of $n$ sequential tasks whose cancelation occurs in the reverse order](image)

**Figure 24: Transaction consisting of $n$ sequential tasks whose cancelation occurs in the reverse order**

Figure 25 models a compensation pair that admits a failure of its compensation activity. In this model, the task $\neg T$ 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 with five I/O ports. Using such connectors, we can model transactions with exceptions or hazards.

![Diagram of a compensation association with possible failure](image)

**Figure 25: Compensation association with possible failure**
Figure 26: Transaction consisting of \( n \) sequential tasks with possible failures in compensation activities

Figure 26 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 (see Figure 20). First, all compensation pair connectors receive cancel messages analogously to the sequential process in Figure 22. 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 \( y_i \) 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 FIFO1 channels and wait until all compensation activities have disposed their tokens, either through the cancel output or through the exception output.

A Reo circuit for a parallel process \( P = C_1 | C_2 | ... | C_n \) is essentially composed of a parallel fork and a parallel join gateways with \( n \) outgoing and \( n \) incoming branches, respectively. When an activity \( T_n \) 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 a transaction. Additionally to the aforementioned pattern, we must commit each activity after all branches have completed successfully.
Below we consider transactions 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 Figure 25. 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.

![Figure 25: Discriminator Choice Circuit](image_url)

**Figure 25: Discriminator Choice Circuit**

Note that designers are not supposed to directly construct complex circuits like the one in Figure 26. Instead, we assume they use BPMN or UML ADs, while Reo circuits can be regarded as the semantics of such higher-level specifications and generated automatically. In some cases refinement will be required; however, 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. Moreover, Reo graphical editor supports compositional construction of process models by means of generics, i.e., self-contained connectors (e.g., Valve, K-counter) that can be further used to compose more complex behavioral patterns.

### 3.7. Mapping UML2 and WS-BPEL to Reo

The comparative analysis of UML ADs and BPMN [Whi04] reveals that both notations have similar modeling objects and are both capable of expressing basic workflow patterns using similar constructs. Therefore, we assume that the mapping of UML ADs to Reo can be done analogously to the mapping of a subset of BPMN to Reo. A more detailed pattern-based analysis of these notations [WAD+05, WAD+06] shows some differences concerning their ability to model complex control and data flow patterns, however this is out of the scope of our work.

The mapping of UML SDs to Reo is extensively described in [ASB08]. Synchronous messages are modeled as synchronous channels while asynchronous messages are modeled as FIFO1 channels. The order of messages sent/received by each partner is preserved with the help of a special Reo circuit which uses a Sequencer connector as its basic element alone with filter channels corresponding to particular types of incoming/outgoing messages and synchronous drain channels used to coordinate the behavior of different entities (lanes). UML SD operators such as parallel, optional or alternative message composition are modeled using structures similar to those used for BPMN gateways.
The mapping of BPEL to Reo is described in [TVM+08]. Due to the block-based nature of BPEL such mapping is more intricate than for graph-based notations such as UML ADs or BPMN. Apart from the constructs similar to those used to model BPMN and UML objects, some novel Reo connectors are involved in this work to simulate BPEL variables and links.

4. Formal Behavioral Model for Service Composition: Extended Constrained Automata

In this section, we briefly define CA and their extensions for modeling time-, resource- and QoS-aware web service compositions.

4.1. Constraint Automata

Let \( \mathcal{N} \) be a finite set of nodes, and \( \text{Data} \) be a fixed, non-empty set of data that can be sent and received via Reo channels. A data assignment denotes a function \( \delta: \mathcal{N} \rightarrow \text{Data} \) where \( \mathcal{N} \subseteq \mathcal{N} \). We use \( DA(\mathcal{N}) \) for the set of all data assignments for the node-set \( \mathcal{N} \). CA use a symbolic representation of data assignments by data constraints which are propositional formula built from the atoms “\( d_A \in P \)”, “\( d_A = d_B \)” or “\( d_A = d \)” and standard Boolean connectors, where \( A, B \in \mathcal{N} \), \( d_A \) is a symbol for the observed data item at node \( A \) and \( d \in \text{Data} \), \( P \subseteq \text{Data} \). We write \( DC(\mathcal{N}) \) to denote the set of data constraints that at most refer to the observed data items \( d_A \) at node \( A \in \mathcal{N} \), and \( DC \) for \( DC(\mathcal{N}) \). Logical implication induces a partial order \( \leq \) on \( DC: g \leq g' \) iff \( g \Rightarrow g' \).

Definition (Constraint Automaton (CA)) A CA is a tuple \( \mathcal{A} = (S, S_0, \mathcal{N}, E) \) where

- \( S \) is a finite set of control states (also called locations),
- \( S_0 \subseteq S \) is a set of initial states,
- \( \mathcal{N} \) is a finite set of node names (e.g. I/O ports of components/services),
- \( E \) is a finite subset of \( S \times 2^\mathcal{N} \times DC \times S \) (called the transition relation of \( \mathcal{A} \)),
- \( DC \) is a data constraint that plays the role of guard for transition. For example, \( d_A = d_B \) is a data constraint that imposes the observed data on ports \( A \) and \( B \) must be equal.

More details about CA and an example of their application to service composition modeling can be found in [BSA+06].

4.1.1. Timed Constraint Automata

Using proper time-sensitive channels, Reo can be used to specify temporal constraints in various business processes. The operational model for time-aware Reo connector circuits is given with the help of Timed Constraint Automata (TCA), which can be defined as follows.

Additionally to the notation introduced for CA, let \( \mathcal{E} \) be a finite set of clocks. A clock assignment means a function \( \nu: \mathcal{E} \rightarrow \mathbb{R}_{\geq 0} \), where \( \mathbb{R}_{\geq 0} \) stands for a set of non-negative real numbers. If \( t \in \mathbb{R}_{\geq 0} \) then \( \nu + t \) denotes the clock assignment that assigns the value \( \nu(x) + t \) to every clock \( x \in C \). If \( C \subseteq \mathcal{E} \) then \( \nu[C:= 0] \) stands for the clock assignment that returns the value 0 for every clock \( x \in C \) and the value \( \nu(x) \) for every clock \( x \in \mathcal{E} \setminus C \). A clock constraint (denoted \( cc \)) for \( \mathcal{E} \) is a conjunction of atoms of the form “\( x \, \Delta \, n \)” where \( x \in \mathcal{E}, \, \, \Delta \in \{<, \leq, >, \geq, =\} \) and \( n \in \mathbb{N} \). \( CA(\mathcal{E}) \) (or \( CA \)) denotes the set of all clock assignments and \( CC(\mathcal{E}) \) (or \( CC \)) the set of all clock constraints.

Definition (Timed Constraint Automaton (TCA)) A TCA is a tuple \( \mathcal{A} = (S, S_0, \mathcal{N}, \mathcal{E}, E, ic) \) where
• \( S \) is a finite set of control states (locations),
• \( S_0 \subseteq S \) is a set of initial states,
• \( \mathcal{N} \) is a finite set of node names,
• \( E \) is a finite subset of \( S \times 2^\mathcal{N} \times DC \times CC \times 2^C \times S \) such that \( dc \in DC(\mathcal{N}) \) for any edge \( e = (s, \mathcal{N}, dc, cc, C, s') \in E \),
• \( \mathcal{C} \) is a finite set of clocks,
• \( \text{id}: S \to CC \) is a function that assigns to any location \( s \) an invariance condition \( \text{id}(s) \).

Moreover, we assume that all data and clock guards on the edges and the invariance conditions are satisfiable. (For edges with the empty node-set, we require a data constraint \( dc \equiv \text{true} \).)

A detailed discussion of TCA along with the examples of their application to component/service-driven software system modeling can be found in [ABB+07].

4.1.2. Resource-aware Timed Constraint Automata

The time required to perform certain actions in the process may depend on the availability of resources. For example, the time for downloading a file from a server depends on the bandwidth of the network. Moreover, most of the systems have to change their states if an interaction has not occurred or an operation has not been completed within a certain timeout. For modeling such requirements in business processes, we extend Reo with time and resource-aware information. The formal model for this extension relies on the notion of Resource-aware Timed Constraint Automata (RSTCA).

**Definition (Resource-aware Timed Constraint Automaton (RSTCA))** A RSTCA is defined as a tuple \( \mathcal{A} = (S, S_0, \mathcal{N}, E, R) \) where

• \( S \) is a finite set of control states (locations),
• \( S_0 \subseteq S \) is a set of initial states,
• \( \mathcal{N} \) is a finite set of node names,
• \( R \) is a finite set of resource names,
• \( E \) is a finite subset of \( \left( S \times T \times S \right) \cup \left( \bigcup_{\mathcal{N} \subseteq N} S \times \{N\} \times DC(\mathcal{N}) \times RC \times \{R \to T\} \times S \) \) that denotes the transition relation. There are two types of transitions:
  o timeout transitions \((s, t, s') \subseteq E \) where \( t \in T \),
  o interactive transitions \((s, \mathcal{N}, g, rc, C, s') \subseteq E \), where \( \mathcal{N}, g \) are as in ordinary constraint automata, \( rc \) is the resource constraint that should be satisfied to trigger the execution of the transition, and \( C: R \to T \) returns the time value that the transition need to be completed, which depends on the available resource values.

A configuration in \( \mathcal{A} \) is a pair \(<s, x>\) where \( s \subseteq S \) is a state and \( x \) is the tuple of resource assignments.

A more detailed description of RSTCA and the example of a system composed of interacting web service interaction protocol modeled with the help of RSTCA can be found in [SA07a].

4.1.3. Quantitative Constraint Automata

We enable the QoS analysis of Reo process models by assigning four values to Reo basic channels, namely, execution time required to transmit a data item, cost of a single data transmission, bandwidth that limits simultaneous data transmission and reliability which represents the probability of a successful data transmission. Formal modeling of QoS-aware business processes in our framework is done using a notion of \( Q \)-algebra.
**Definition (Q-algebra)** A Q-algebra is an algebraic structure \( R = (C, \oplus, \otimes, ||, 0, 1) \) where \( C \) is the domain of \( R \) and represents a set of QoS values. The operation \( \oplus \) induces a partial order on the domain of \( R \) and used to define a preferred value of the QoS dimension, \( \otimes \) is an operator for sequential automata composition, while \( || \) is an operator for parallel automata composition.

The Q-algebras corresponding to the above QoS dimensions can be defined as follows:

- Execution time: \((R + U \{\infty\}, \text{min}, +, \text{max}, \infty, 0)\),
- Cost: \((R + U \{\infty\}, \text{min}, +, +, \infty, 0)\),
- Bandwidth: \((N U \{\infty\}, \text{max}, \text{max}, +, 0, \infty)\),
- Reliability: \(([0, 1], \times, \times, 0, 1)\).

Taking this definition into account, the quantitative CA can be defined as follows:

**Definition (Quantitative Constraint Automaton (QCA))** A QCA is a tuple \( \mathcal{A} = (S, S_0, N, R, E) \) where

- \( S \) is a finite set of control states,
- \( S_0 \subseteq S \) is its initial state,
- \( N \) is a finite set of nodes,
- \( R = (C, \oplus, \otimes, ||, 0, 1) \) is a labeled Q-algebra with domain \( C \) of costs,
- \( E \) is a finite subset of \( \bigcup_{N \subseteq N} S \times \{N\} \times DC(N) \times C \times S \) that denotes the transition relation.

Examples of QoS-aware service compositions modeled with the help of QCA can be found in [ACS+07].

### 4.2. Constraint Automata as Semantic Model for Reo

The compositional construction of Reo process models discussed in Section 3 corresponds to the compositional construction of those models using constraint automata: the constraint automaton of a Reo circuit can be obtained by composing the CA of its sub-circuits, which in turn are derived by composing the CA of their constituent channels. Therefore it is enough to map the basic Reo channels to CA and define the operations to compose them. Figure 28 shows the CA for the four basic Reo channels, namely, synchronous drain, FIFO1, synchronous and synchronous lossy channels. The mapping of other channels and some useful connectors is extensively discussed in [BSA+06].

![Figure 28: Constraint automata for basic Reo channels](image)

Constructing complex connectors out of simpler ones is done by the join operation in Reo. Joining two nodes destroys both nodes and produces a new node on which all of their coincident channel ends coincide. Each channel in Reo is mapped to a constraint automaton. The join operation can be realized by the synchronized product construction of constraint automata.
Definition (Product of CA). The product for two given constraint automata $\mathcal{A}_1 = (S_1, s_{01}, \mathcal{N}_1, E_1)$ and $\mathcal{A}_2 = (S_2, s_{02}, \mathcal{N}_2, E_2)$ is defined as a constraint automaton $\mathcal{A}_1 \times \mathcal{A}_2$ with the components $(S_1 \times S_2, s_{01}, s_{02}, \mathcal{N}_1 \cup \mathcal{N}_2, E)$ where $E$ is given by the following rules:

- If $e_1 = (s_1, N_1, g_1, s_1')$, $e_2 = (s_2, N_2, g_2, s_2')$, $N_1 \cap N_2 = \emptyset$ and $g_1 \land g_2$ is satisfiable, then $e = (s_1, s_2, N_1 \cup N_2, g_1 \lor g_2, <s_1', s_2'>)$.
- If $e_1 = (s_1, N, g, s_1')$ where $N \cap N_2 = \emptyset$, then $(s_1, s_2, N, g, <s_1', s_2>)$.
- If $e_2 = (s_2, N, g, s_2')$, where $N \cap N_1 = \emptyset$, then $(s_1, s_2, N \cup N_1, g, <s_1, s_2>)$.

The first rule is applied when there are two transitions in the automata which can be fired together. This happens only if there is no shared name in the two automata that is present on one of the transitions but not present on the other one. In this case the transition in the resulting automaton has the union of the name sets on both transitions, and the data constraint is the conjunction of the data constraints of the two transitions. The second rule is applied when a transition in one automaton can be fired independently of the other automaton, which happens when the names on the transition are not included in the other automaton. The third rule is symmetric to the second one.

Another operator for abstraction purposes in Reo to build connectors from networks by declaring the internal topology of the network as hidden is the hiding operator. Hiding takes an input a constraint automaton $\mathcal{A} = (S, s_0, N, E_M)$ and a non-empty node-set $M \subseteq \mathcal{N}$. The result is a constraint automaton $\text{hide}(\mathcal{A}, M)$ that behaves as $\mathcal{A}$ except that data flow at the nodes $A \in M$ is made invisible. Formally, $\text{hide}(\mathcal{A}, M) = (S, s_0, N \setminus M, E_M)$ where $(s, N', g', E_M, s')$ if there exists a transition $(s, N, g, s')$ such that $N' = N \setminus M$ and $g = \exists M[g]$. Here $\exists M[g]$ stands short for $\bigvee_{\delta \in DA(M)} g[d_A / \delta.A | A \in M]$, where $g[d_A / \delta.A | A \in M]$ denotes the syntactic replacement of all occurrences of $d_A$ in $g$ for $A \in M$ with $\delta.A$. Therefore, $\exists M[g]$ formalizes the set of data assignments for $N$ that are obtained from a data assignment $\delta$ for $N$ where $g$ holds by dropping the assignments for the nodes in $N \cap M$.

4.3. Practical Implications

The main goal of the formal model presented above is to enable automated verification of compliance-aware business processes and web service compositions. This can be done by using the Vereofy [http://www.vereofy.de/] model checking tool developed as part of the EU CREDO project. Vereofy uses two input languages, namely, Reo Scripting Language (RSL), and a guarded command language called Constraint Automata Reactive Module Language (CARML) which are syntactic versions of Reo and CA, respectively. For handling the state space explosion problem typical for model checking tools, Vereofy generates an internal symbolic representation of the CA for services, the glue code for service composition. This symbolic representation is based on switching functions which are stored and manipulated by means of Binary Decision Diagrams (BDDs). The theoretical background for this work can be found in [KB07]. Vereofy allows for linear and branching time model checking adapted to the Reo and CA framework. Thus, we assume that process properties, including compliance-related constraints, will be expressed in LTL or Branching Time Stream Logic (BSTL). If required, other formats will be adopted in COMPAS as well.

Certain compliance requirements coming from legislative/regulatory documents can be seen as informal descriptions of ideal process fragments in a certain application domain. Such process fragments in their turn can be modeled using Reo and/or CA. In this case, the
conformance of software systems actually used in organizations to the ideal processes can be established by checking bisimulation equivalence of corresponding CA models.

For a constraint automaton $\mathcal{A} = (S, S_0, \mathcal{N}, E)$, $s \in S$, $N \subseteq \mathcal{N}$ and $P \subseteq S$ we define $dc_\mathcal{A}(s, N, P) = \bigvee \{s: (s, N, g, p) \text{ for some } p \in P\}$. If the automaton is clear from the context we leave out the subscript and write $dc(s, N, P)$.

**Definition (Bisimulation).** Let $\mathcal{A} = (S, S_0, \mathcal{N}, E)$ be a constraint automaton. An equivalence relation $R$ on $S$ is called bisimulation for $\mathcal{A}$ if for all pairs $(s_1, s_2) \in R$, all $R$-equivalence classes $P \in Q / R$ and every $N \in \mathcal{N}$, $dc(s_1, N, P) \equiv dc(s_2, N, P)$ holds.

Algorithms to compute bisimulation, and, thus, check for the behavioral equivalence of two given CA or Reo circuits can be found in [BB07].

In addition to the properties mentioned in Section 1, namely, compositionality and extensibility, an interesting property of CA as a formal model for web service compositions is their ability to deal with interaction transactions. For example, if a user has to provide his name, birth date, passport number and address to a system, it is often not important in what order he/she introduces these data. CA allow for the abstraction from such details by modelling the whole interaction as a single transition.

### 4.4. Web Services as Constraint Automata

In this section we provide examples that illustrate the use of CA to model business processes and web service compositions.

Figure 29 shows an automaton that models a service composition for the login phase of an online banking scenario depicted in Figure 4. Additionally to the standard definition of the CA, here we use final states to show the end of the interaction protocol. Such an automaton can be obtained from the corresponding UML sequence diagram using UML2Reo and Reo2CA converters [AS08].
Figure 29: Constraint automata for the on-line banking scenario

Figure 30 models the BPMN compensation pair introduced in Figure 21 using the Reo graphical editor. In this circuit, we introduce two auxiliary nodes $x_1$ and $x_2$ along with a FIFO1 and two synchronous channels connecting the “cancelled” and “committed” states with the “start” state of the circuit in order to prevent its multiple invocations and, thus, obtain a simple automaton corresponding to a single execution of the source activity $T$. Such an automaton can be obtained using the Reo2CA converter. Figure 31 shows a snapshot of the CA editor for the compensation pair with data flow in ports “start”, “performed”, “cancel”, “commit”, “cancelled” and “committed”. Data flow in all auxiliary states has been hidden using the hide operator defined in Section 5.2.

Figure 30: Reo model for the BPMN compensation pair
5. Augmenting Process Behavioral Models with Data Domains

In this section we introduce data domains as a mean for refining external behaviour specification of processes and services. Data domains can also be used to integrate specifications coming from other compliance properties with the behavioural specification in order to discover and handle possible interaction between those specifications. One example is the interaction between the behaviour specification and access control specification on the service’s operations [MOP+06].

In the following of this section, we concentrate in interactions between two parties expressed as a sequence of XML messages, each message coming with a time information (the time at which the message was send) and a direction expressing which side it came from.

**Definition (Conversation):** A conversation is a sequence of XML messages, each message coming with a time information (the time at which the message was send) and a direction expressing which side it came from.

**Definition (Conversation):** A conversation is a sequence of XML messages, each message coming with a time information (the time at which the message was send) and a direction expressing which side it came from.

5.1. Data domains

A data domain consists in possible specifications on messages, possible instance values associated to actual messages and corresponding to some specifications and functions used to express which instance value matches which specification and how to associate these values to messages in a conversation. For example, message specifications can be XML schemas, access control policies or constraints on values in messages.

**Definition (Data domain):** A data domain is a tuple of a partially ordered set of specification values, a set of instance values, an instantiation morphism and a function annotating the last message of a timed message sequence with an instance value.
5.1.1. Examples of data domains

**XML Schema specification:** The set of specification values \((V, \leq_V)\) is the set of XML Schemas, ordered by semantic inclusion [FCB02]: a schema S1 is smaller than or equals to a schema S2 if every XML document valid w.r.t. S1 is also valid w.r.t. S2. The set \(I\) of instance values is the set of XML documents. \(f_i\) associates to each schema the set of all XML documents which are valid w.r.t. the given schema. \(f_a\) associates to \((m_n, t_n, \leftrightarrow)\) the message \(m_n\) itself.

**Message meaning:** Ontology concepts can be associated to messages in WSDL by the way of e.g. [SAWSDL]. The set of specification values \((V, \leq_V)\) is a set of concepts defined in the ontology describing the meaning of operations and ordered by concept inclusion. Instance values are also the concepts of this ontology and \(f_i\) associates to each concept the set of all concepts that are subsumed by this one. \(f_a\) associates to \((m_n, t_n, \leftrightarrow)\) the concept associated to \(m_n\) by the SAWSDL definitions.

**Values in messages:** Specification values are constraints on the value of certain parts of messages. The constrained parts could be identified by e.g. an XPath expression. The constraints could be equality constraints, type constraints, … Instance values are XML documents and \(f_i\) associates to each constant the set of all XML documents which the constraint evaluates to true. \(f_a\) associates to \((m_n, t_n, \leftrightarrow)\) the message \(m_n\) itself.

**Access control:** We consider a policy language for describing access control on input messages and attributes used by the policies to determine whether access is granted. The set of specification values \((V, \leq_V)\) is the set of policies. A policy is smaller than another one if it is more restrictive. Instance values are tuples of attributes and \(f_i\) associates to each policy the set of tuples such that access is granted by the policy for the values of the tuple. This example can be instantiated as follows. The policy language can be an ontology of credentials, concepts being (set of) credential specifications and objects being sets of credentials. The set of specification values \((V, \leq_V)\) is the set of concepts of the ontology ordered by concept inclusion. Instances are sets of actual credentials and the instantiation function is given by the ontology. We assume that for each credential and each message that if the credential appears in the message, then there is a specification of where the credential is in the message. This can be achieved for example by annotating the XML schema of messages. \(f_i\) associates to the set of all credentials that appear in message of the form Error! Objects cannot be created from editing field codes in the conversation. Error! Objects cannot be created from editing field codes. conversations are treated in a similar way, using messages of the form Error! Objects cannot be created from editing field codes.

**Composite domains:** different data domains can be used together using cartesian product operations for combining them. Thus two data domains Error! Objects cannot be created from editing field codes. and Error! Objects cannot be created from editing field codes. specifications can be combined into Error! Objects cannot be created from editing field codes. ordered by the product order, Error! Objects cannot be created from editing field codes.. Error! Objects cannot be created from editing field codes. and Error! Objects cannot be created from editing field codes.
5.2. Conversation models with data domains

5.2.1. Business Protocols

A business protocol [BCT04] is an automata where transitions are annotated by input or output messages names, or timeouts. A business protocol therefore expresses sets of timed message sequences that are accepted by the service. Such protocols can be extracted from other behavioural models such as UML sequence diagrams, BPEL processes. Moreover the formalism of business protocols is a special case of constraint automata, which make it easy to integrate this model in the general behavioural model of WP3.

We propose to extend such business protocols using the data domains previously introduced. A transition then corresponds to a message name with a specification value or to a timeout. One can remark that the message name used in business protocols is just another kind of data domain: specification values are message names, and instance values are messages as XML documents. Since composite data domains can be used to combine data domains, message transition can therefore be annotated with just a specification and a input/output indication without loss of generality, which results into the following definition for extended business protocols:

**Definition (Extended Business Protocols):** An extended business protocol is a tuple $(S, s_0, F, T_t, T_m)$ where $S$ is a set of states; $s_0$ is the initial state; $F$ is a set of final states; $T_t$ is a set of timeout transitions of the form $(s_i, s_{i+1}, t)$ where $s_i$ and $s_{i+1}$ are states and $t$ is a duration; and $T_m$ is a set of message transitions of the form $(s_i, s_{i+1}, sp, pol)$ where $s_i$ and $s_{i+1}$ are states, $sp$ is a specification value and $pol$ is either $+$ for input messages or $-$ for output messages. Moreover an extended business protocol satisfies the following conditions: there is at most one timeout transition originating from a given state and it is not ambiguous, i.e. if two message transitions are originating from the same state, the instantiation of their specification is disjoint.

The following example shows a business protocol expressing the interactions between the bank and the user extracted from Figure 29 from the bank’s perspective, augmented with a timeout transition from state $S_3$ and a subsequent transition CloseSession. This protocol could be extended with schema information extrcated from the WSDL and credential annotation on the AccountNo&password transition and access control policy on the DisplayAccount transition.

![Business protocol diagram](image)

**Figure 32: Business protocol for the on-line banking scenario augmented with a timeout transition state**

In the rest of this report we will use the term business protocol instead of extended business protocol. While using timeouts in this definition is convenient for design purposes, it is not easy to work with from the analysis point of view. Therefore we define timed business proto-
cols where timeout transitions are replaced with constraints on message transitions, resulting in the following definition:

**Definition (Timed Business Protocols):** A timed business protocol is a tuple $\langle S, s_0, F, T_m, \mathrm{pol} \rangle$, where $S$ is a set of states; $s_0$ is the initial state; $F$ is a set of final states; and $T_m$ is a set of message transitions of the form $(s_i, s_{i+1}, sp, t, \mathrm{pol})$ where $s_i$ and $s_{i+1}$ are states, $sp$ is a specification value, $t$ is a time interval and $\mathrm{pol}$ is either $+$ for input messages or $-$ for output messages.

Given an extended business protocol, it is possible to obtain an equivalent timed business protocol by applying the following transformation: first, for each message transition $(s_i, s_{i+1}, sp, t, \mathrm{pol})$, if there is a timeout transition $(s_{i+1}, s_{i+2}, t)$ then the transition is transformed into $(s_i, s_{i+1}, sp, [0, t], \mathrm{pol})$, else it is transformed into the transition $(s_i, s_{i+1}, sp, [0, \infty], \mathrm{pol})$. Then for each path of timeout transitions $(s_k, s_{k+1}, t_k)$ $(s_{k+1}, s_{k+2}, t_{k+1})$ … $(s_l, s_{l+1}, t_l)$, for each transition $(s_k, s_{k+1}, sp, [0, t], \mathrm{pol})$ added in the first step, add a transition $(s_i, s_{k+2}, sp, [t_i + \ldots + t_k, t_i + \ldots + t_k + t], \mathrm{pol})$.

The following diagram shows the timed business protocol corresponding to the previous example:

![Timed business protocol for the on-line banking scenario](image)

Figure 33: Timed business protocol for the on-line banking scenario

A conversation is recognized by a timed business protocol if there exists a path in the protocol starting at $s_0$ and such that for each step in the conversation, the $k^{th}$ step $(s_k, s_{k+1}, sp_k, t_k, \mathrm{pol}_k)$ in the path is such that $\leftrightarrow$ is $\rightarrow$ then $\mathrm{pol}_k$ is $-$ else it is $+$. A conversation is total if the corresponding path ends in a final state.

### 5.2.2. Interaction traces

Given a conversation and two timed business protocols it is possible to determine whether the conversation corresponds to sequence of messages recognized by the protocols. To achieve this, introduce a notion of interaction trace as a conversation annotated with protocol states and instance values:

**Definition (Interaction trace):** Given two timed business protocols $\langle S, s_0, F, T_m, \mathrm{pol} \rangle$ and $\langle S', s'_0, F', T'_m, \mathrm{pol}' \rangle$, an interaction trace is a sequence of steps $(s_i, m, t, iv)$ where $s_i$ is a state, $m$ is a message, $t$ is a timestamp and $iv$ is an instance value.

Given a conversation and two business protocols, if the conversation is recognized by the first protocol and if the opposite conversation is recognized by the second protocol, then an inter-
action trace can be build as follows: for each step \((m_k, t_k, \leftrightarrow)\) of the conversation corresponding to steps \((s_{1k}, s_{1k}', s_{1k}^+, t_{1k}^+, p_{1k}^+)\) and \((s_{2k}, s_{2k}', s_{2k}^+, t_{2k}^+, p_{2k}^+)\) in the recognition path in the first and second protocols, the corresponding step in the interaction trace is \((s_{1k}^+, s_{1k}', s_{2k}^+, s_{2k}', m_k, t_k, \leftrightarrow)\). Such an interaction trace is called a correct interaction trace. A correct interaction trace is total if the conversation is total for both protocols.

5.3. Compatibility and replaceability analysis

One issue when composing services is to determine whether each service interacting with another one will respect its behaviour, not only in terms of message sequencing but also in terms of the specification provided through data domains. This can be expressed as a notion of compatibility between the two services. This notion can be informally stated as: “Any message sent by one service should be receivable by the other and there is no live or deadlock during an interaction between those two services.” This can be formally stated in terms of interactions traces between the protocols of the two services:

Definition (Compatibility in terms of interaction traces): For each correct interaction trace \(IT\), in the set of correct interaction traces between two timed protocols \(T_1\) and \(T_2\):

- For every transition \((s_{1k+1}^+, s_{1k+2}^+, s_{1k+1}, t_{1k+1}, -)\) in \(T_1\), for every \(t\) in \(t_{1k+1}\), for every \(m_{k+1}\), if \(Error! Objects cannot be created from editing field codes.\) then there exists \(s_{2k+2}^+\) such that \(IT.(s_{1k+1}^+, s_{1k+2}^+, s_{2k+1}^+, s_{2k+2}^+, m_{k+1}, t_k+t, \rightarrow)\) is a correct interaction trace between the two protocols.

- For every transition \((s_{2k+1}^+, s_{2k+2}^+, s_{2k+1}, t_{2k+1}, -)\) in \(T_2\), for every \(t\) in \(t_{2k+1}\), for every \(m_{k+1}\), if \(Error! Objects cannot be created from editing field codes.\) then there exists \(s_{1k+2}^+\) such that \(IT.(s_{1k+1}^+, s_{1k+2}^+, s_{2k+1}^+, s_{2k+2}^+, m_{k+1}, t_k+t, \leftarrow)\) is a correct interaction trace between the two protocols.

- Each correct interaction trace between the two protocols is a prefix of a total interaction trace between the two protocols.

If the annotation function in the data domains only depends on the last message and if \(Error! Objects cannot be created from editing field codes.\) is a lattice, the compatibility in terms of interaction traces can be translated into a property on the protocols themselves. This first requires to define a notion of product of two timed protocols:

Definition (Product of two business protocols): Let \(Error! Objects cannot be created from editing field codes.\) and \(Error! Objects cannot be created from editing field codes.\) be two timed business protocols. The product of those two protocols is a tuple \(Error! Objects cannot be created from editing field codes.\) where \(T\) is the smallest set of transition of the form \((s_{1k+1}^+, s_{1k+2}^+, s_{1k+1}, t_{1k+1}, \leftrightarrow)\) such that for every transition \((s_{1k+1}^+, s_{1k+2}^+, s_{1k+1}, t_{1k+1}, \leftrightarrow)\) in \(T\) and every transition \((s_{2k+1}^+, s_{2k+2}^+, s_{2k+1}, t_{2k+1}, \leftrightarrow)\) in \(T\), if \(pol \neq pol^1\), \(sp \cap sp^2 \neq \emptyset\) and \(ti \cap ti^2 \neq \emptyset\) then \((s_{1k+1}^+, s_{1k+2}^+, s_{1k+1}, t_{1k+1}, \leftrightarrow) \in T\), with \(\leftrightarrow \leftarrow \rightarrow\) if \(pol = -\), \(\leftarrow\) otherwise.

The compatibility can then be expressed as a condition on the protocols and their product:

Definition (Compatibility in terms of timed business protocols): The timed protocols \(Error! Objects cannot be created from editing field codes.\) and \(Error! Objects cannot be created from editing field codes.\) are compatible if there exists a relation \(R \subseteq S1 \times S2\) such that:
- \((s^1_0, s^2_0) \in R\).

- For every \((s^1, s^2) \in R\), for every \((s^1, s^1', sp, ti, -) \in T^1\), there exists \(n\) transitions \((s^2, s^2_j, sp_j, t_j, +) \in T^2\) such that \((sp, ti) \leq \text{Error! Objects cannot be created from editing field codes.}(sp_j, t_j)\) and for \(1 \leq j \leq n\), \((s^1, s^2_j) \in R\).

- For every \((s^1, s^2) \in R\), for every \((s^2, s^2', sp, ti, -) \in T^2\), there exists \(n\) transitions \((s^1, s^1_j, sp_j, t_j, +) \in T^1\) such that \((sp, ti) \leq \text{Error! Objects cannot be created from editing field codes.}(sp_j, t_j)\) and for \(1 \leq j \leq n\), \((s^1_j, s^2) \in R\).

- Every pair of states in \(R\) is co-accessible in the product of the two timed protocols.

Note that although this definition is close to the definition of bisimulation in constraint automata, the two notions are distinct as shown by the following example, in which the two protocols are compatible but not bisimilar. One can remark that one does not simulate the other one, even when reversing polarities:

A timed business protocol can either be the protocol of a service or a process, or it can be the specification of a service to put at some point in a composition. It is then interesting to compare two protocols w.r.t their compatibility with other protocols, in particular from the replaceability point of view. A timed business protocol can replace another one if it is compatible at least with all the protocols the second one is compatible with. Then we say that the first protocol subsumes the second one. The subsumption order between timed business protocols is noted \(\leq P\). If the annotation function in the data domains only depends on the last message and if \text{Error! Objects cannot be created from editing field codes.} is a lattice, then the subsumption can be determined directly on the protocols:

**Definition (Timed business protocol subsumption):** The protocol \text{Error! Objects cannot be created from editing field codes.} subsumes the protocol \text{Error! Objects cannot be created from editing field codes.} if there exists a relation \(R \subseteq S1 \times S2\) such that:

- \((s^1_0, s^2_0) \in R\).

- For every \((s^1, s^2) \in R\), for every \((s^1, s^1', sp, ti, -) \in T^1\), there exists \(n\) transitions \((s^2, s^2_j, sp_j, t_j, -) \in T^2\) such that \((sp, ti) \leq \text{Error! Objects cannot be created from editing field codes.}(sp_j, t_j)\) and for \(1 \leq j \leq n\), \((s^1, s^2_j) \in R\).
- For every \((s^1, s^2) \in R\), for every \((s^3, s^2, sp, ti, +) \in T^2\), there exists \(n\) transitions \((s^1_j, s^1_j, sp_j, t_{ij}, +) \in T^1\) such that \((sp, ti) \leq (sp_j, t_{ij})\) and for \(1 \leq j \leq n\), \((s^1_j, s^2) \in R\).

- Every pair of states in \(R\) is co-accessible in the product of the first protocol with the opposite of the second one.

6. Compliance-aware Business Process Design in Examples

In this section, we provide several examples illustrating how the Reo/CA framework can be used for technical-level business process modeling and compliance enforcement by design.

6.1. Control Flow Constraints

Some regulations may impose control flow constraints by prescribing the order in which certain activities must be executed. For example, a foreign resident can open a bank account and ask for a loan only after he/she has obtained a valid national security number. Such constraints are especially difficult to enforce in the distributed processes that involve several organizations that are not coordinated by a single partner.

The current version of BPMN does not provide good support for specifying dependencies in concurrent flows. WS-BPEL instead can deal with them 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. For example, consider a process \(P = C1; (C2; C4|C3;C5);C6\) consisting of six compensatable activities \(C_i = T_i \rightarrow T_i\) (adopted from Bruni et al. [BMM05]). 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 Figure 35. 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.

![Figure 35: Modeling concurrent flow dependencies in Reo](image)

Using Reo flash animations or model checking tools we can assure that the constraint stating that the data flow cannot be observed on port \(A\) before it has been observed in port \(B\) holds for this process. Other important control flow properties such as *eventuality* (output is achieved for any process run) or *durability* (no more than one output is reached for any process run) can be verified as well.
6.1.1. **Transactional constraints**

Transactional process behavior is essential for many business processes while often it is unclear how compensation for a failed transaction must be made. A number of formally grounded approaches have been proposed to examine process transactional behavior both in general settings and within SOA-related solutions [BLZ03, BF04, BHF05, BMM05, ES08]. While proposing valid models for specifying transactional behavior, none of these languages considers full-featured transaction management as a specific, although rather arduous, case of business process modeling and web service coordination. Our approach addresses this problem in a unified manner.

For example, control links for synchronizing concurrent flows discussed above obscure the desired compensation behavior in the case of process failure. We assume that such behavior can vary in different scenarios and is subject to careful modeling. Figure 36 shows a Reo circuit for the process compensation after executing the activity \( T_6 \) of the aforementioned process. 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 the task \( T_3 \) but after the task \( T_4 \) has been compensated for.

![Figure 36: Compensation in the processes with control flow dependencies](image)

Generally speaking, all transactional processes must be compliant to the requirement that states that all involved activities are either successfully completed or successfully canceled (known as *atomicity*). Whether this constraint holds for Reo process models and service compositions can be checked with the help of Reo animation engine or the *Vereofy* model checker. Specifically for our example, an BSTL constraint formula ensuring that for any process run in the compensation mode there cannot be data flow observed on the port “cancel” of the component \( C_3 \) before it has been observed on the port “cancelled” of the component \( C_4 \) can be expressed as

\[
\forall[(\neg C_4.\text{cancelled})*]\neg\exists(C_3.\text{cancel})\text{true}.
\]

Some other examples of structural business process analysis with Reo Coordination Tools can be found in our recent publications [AKM08, TVM+08].
6.2. Resource Constraints

6.2.1. Separation of duties

One of the popular resource-aware constraints is a dual control or so-called four eyes principle [SS75]. It is applied, for example, in investment banking, to separate the duties of a trader from the duties of an internal auditor. In the corresponding process model, it is important to ensure that generally each bank clerk can play both roles, but he/she cannot play both roles in a single instance of the process. Later the term Separation of Duties (SoD) was introduced for referring to a principle of information protection and fraud prevention by limiting user access to vulnerable data and/or operations. This category of compliance requirements is extensively analyzed, e.g., in [GGF98, SLS06].

Standard BPMN notation provides poor support for specifying such constraints. This problem is well-recognized and several extensions have been proposed to incorporate resource allocation patterns such as separation of duty, role-based allocation, case handling, or history-based allocation in business processes. Figure 37 shows an example of BPMN diagrams with additional textual annotations expressing task authorization constraints in the form of

\[ c = (T_c, n_u, m_{th}), \text{ with } n_u, m_{th} \in \mathbb{N}, \]

where \( T_c \) is a set of tasks, \( n_u \) defines the minimal number of different users that have to allocate a task \( t_k \in T_c \), while \( m_{th} \) is defined as the threshold of task instances for any \( t_k \in T_c \) a user \( u_n \) is allowed to allocate [WS07].

![Figure 37: BPMN diagrams augmented with task authorization constraints](image_url)

Generally, to guarantee that SoD requirements hold in a certain process, runtime process monitoring and rule-based event analysis are required. This issue is considered in WP5 of the COMPAS project. However, the problem can be resolved also by enforcing a proper task allocation policy at the design time. The above notation intuitively describes task assignment to users, but it does not make clear how such an assignment is implemented. Moreover, in the case of fully automated service-oriented systems, task allocation has to be performed automatically. Using our approach, a designer can develop an exact model of a process compliant to SoD requirements. For example, Figure 38 shows a Reo process model for the above scenario: this circuit guarantees that tasks \( T_1 \) and \( T_2 \) in each process instance will be executed by separate users modelled as data items given to ports \( A \) and \( B \).
The compliance of an arbitrary Reo process model to the four eyes principle can be established using LTL formula for the model checker stating that a data item observed on each of the ports, A or B, cannot be observed on both input ports of tasks $T_1$ and $T_2$.

An example shown in Figure 39 further illustrates the usage of technical-level process modeling to guarantee compliance to SoD requirements by design. In this scenario, the constraint says that two users have to participate in the execution of a process consisting of four sequential tasks. This constraint can be enforced in different ways. For instance, the first user can be assigned to execute the first two tasks while the second user completes the process by executing the remaining two tasks. Although this task assignment algorithm is compliant to the initial SoD constraint, it is not efficient from the point of view of resource allocation in general. Figure 40 shows an extract from the exact graphical model of this process, where each user can execute at most three tasks from each process instance. This Reo circuit exploits a $K$-counter Reo connector defined in Figure 8 to count tasks performed by each user. If three tasks have been executed by one user, the corresponding token will be removed from the circuit, and, thus, this user will not be able to execute the remaining tasks. A designer can establish whether the process model behaves as intended with the help of the Reo animation engine or the Vereofy model checker.

![Figure 38: Reo process model satisfying the SoD (four eyes principle) constraint](image)

![Figure 39: BPMN diagram augmented with operation SoD constraint](image)
7. Related work

In this section, we briefly overview related work and compare our model with existing approaches to formal compliance-aware business process modeling and behavioral composition specification. Detailed survey on this subject can be found in [D2.1].

7.1. Formal semantics and transformations of process modeling notations

Two attempts have been made to define formal semantics for BPMN [DDO08, WG08]. In the first approach [DDO08], a core subset of BPMN is mapped into Petri nets. However, this approach encounters problems with reflecting the behavior of multiple concurrent activities in presence of exception handling. In our work we consider a significantly larger subset of BPMN constructs, in particular, compensation pairs and transactional sub-processes. The second approach [WG08] formalizes the BPMN semantics (including time-aware semantics [WG08a]) in a more consistent way using Communicating Sequential Processes (CSP). The main drawback of this model is that it does not preserve the structure of BPMN diagrams which makes the mapping difficult to follow. Additionally to these approaches, a number of works provide insights and tools for automated translation of BPMN into BPEL processes [RM06, ODH+07]. Such translations bridge the gap between the process modeling and their implementation using web services technology. However, they pose significant restrictions on admissible BPMN patterns and do not prevent developers from implementing erroneous processes. Later on, BPEL processes can be verified using a wide range of formal techniques [LM07, Loh08, OVA+07] and model checking tools [Nak06], but this scenario shifts the process verification to the implementation phase and thus slows down the incremental process design. Petri-nets-based semantics for UML ADs can be found in [SH05].

One of the most difficult aspects of workflow modeling is the design of transactional processes. A number of attempts have been made to formally specify exception and compensation handling in various workflow systems. Bocchi et al. [BLZ03] proposes an extension of the asynchronous $\pi$-calculus to deal with long-running business transactions. This approach does not relate compensations with the control flow of the original process. In particular, this means that if one of the activities in the sequential flow fails, the compensations for all pre-

Figure 40: Fragment of a Reo process model enforcing the SoD constraint

7. Related work

In this section, we briefly overview related work and compare our model with existing approaches to formal compliance-aware business process modeling and behavioral composition specification. Detailed survey on this subject can be found in [D2.1].

7.1. Formal semantics and transformations of process modeling notations

Two attempts have been made to define formal semantics for BPMN [DDO08, WG08]. In the first approach [DDO08], a core subset of BPMN is mapped into Petri nets. However, this approach encounters problems with reflecting the behavior of multiple concurrent activities in presence of exception handling. In our work we consider a significantly larger subset of BPMN constructs, in particular, compensation pairs and transactional sub-processes. The second approach [WG08] formalizes the BPMN semantics (including time-aware semantics [WG08a]) in a more consistent way using Communicating Sequential Processes (CSP). The main drawback of this model is that it does not preserve the structure of BPMN diagrams which makes the mapping difficult to follow. Additionally to these approaches, a number of works provide insights and tools for automated translation of BPMN into BPEL processes [RM06, ODH+07]. Such translations bridge the gap between the process modeling and their implementation using web services technology. However, they pose significant restrictions on admissible BPMN patterns and do not prevent developers from implementing erroneous processes. Later on, BPEL processes can be verified using a wide range of formal techniques [LM07, Loh08, OVA+07] and model checking tools [Nak06], but this scenario shifts the process verification to the implementation phase and thus slows down the incremental process design. Petri-nets-based semantics for UML ADs can be found in [SH05].

One of the most difficult aspects of workflow modeling is the design of transactional processes. A number of attempts have been made to formally specify exception and compensation handling in various workflow systems. Bocchi et al. [BLZ03] proposes an extension of the asynchronous $\pi$-calculus to deal with long-running business transactions. This approach does not relate compensations with the control flow of the original process. In particular, this means that if one of the activities in the sequential flow fails, the compensations for all pr-
viously executed activities start simultaneously, while another (e.g., reverse) order may be needed. Our approach allows designers to explicitly specify a required compensation process.

Butler and Ferreira [BF04] 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 [BHF05], 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 transaction specification is Sagas Calculi [BMM05]. Sagas 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. 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 sub-process as opposed to the implicit or default compensation. A comparison of cCSP and Sagas [BBF+05] 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. [GRG+07] propose an event-driven approach to validate the transactional behavior of web service compositions. In this work, service compositions are specified using transactional patterns, 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.

Lucchi and Mazzara [LM07] introduce an orchestration language, called webπ, which is based on the idea of event notification as the unique error handling mechanism. 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. Several other works propose Petri net semantics for WS-BPEL. The most complete of them is given in [Loh08]. This approach formalizes control and data flow in WS-BPEL by means Petri nets extended with the interface for asynchronous message passing.
7.2. Compliance-aware business process modeling

As [D2.1] shows, compliance policies are very broad in nature. Clearly, some policies relate to business processes, while others may only partially do or may not relate to them at all. Business process modeling languages and their graphical representations are relevant for capturing, describing, formalizing, executing and enforcing policies that can be expressed in a form of local or global constraints or permissions and obligations on control or data flow. Often, process definition languages are augmented with modal or temporal logic formulae to encode certain kinds of compliance rules such as that some condition will eventually be true or will not be true until another statement becomes true. Several frameworks exploit this approach for modeling legislative/regulatory compliance rules [LMX07, GK07].

In particular, Liu et al. [LMX07] introduce the Business Process Specification Language (BPSL) for expressing compliance concerns on top of BPEL processes. Then, BPSL constructs are automatically translated into LTL while BPEL processes are first translated into π-calculus and finally into finite state machines to enable static process verification by means of model-checking techniques. Ghose and Koliadis [GK07] deal with BPMN processes that are further refined and represented in a form of semantically-annotated digraphs called semantic process networks. Compliance rules in this work are modeled using CTL. Giblin et al. [GLM+05] introduce REALM (Regulations Expressed as Logical Models), a metamodel and a method for modeling compliance rules over concept models in UML. Since the OCL does not support temporal predicates, REALM specifically focuses on time-based properties expressed in a specially designed Real-time Temporal Object Logic (RTOL).

Several other approaches consider specific categories of compliance rules. For example, Governatori et al. [GMS06] developed a Formal Contract Language (FCL) for representing compliance requirements extracted from service contracts. FCL expresses normative behavior of the contract signing parties by means of chains of permissions, obligations, and violations. Brunel et al. [BCC+07] use labeled Kripke structures which are a state/event extension of LTL both for specifying system behavior and related security requirements, also defined in a form of permissions, obligations, and violations.

In this context, we see Reo and its underlying mathematical formalism in terms of extended CA as a common operational semantics for unambiguous modeling of compliance-aware business processes. Existing model-checking and bisimulation tools for Reo are able to automatically verify important properties of process models expressed in LTL and BSTL or represented in a form of Reo connectors.

8. Summary

In this deliverable, we presented a core behavioral model for service description, thus, fulfilling the objectives of Task 3.1 [DoW]. Our model is based on extended constraint automata for description of service behavioural interfaces. A set of converters from widely used business process modeling notations, a graphical language, GUI and verification tools are involved to support the specification and formal analysis of process models and corresponding service compositions. Several examples illustrate how these models can be used to verify business process compatibility and ensure process compliance to external regulations imposing constraints on process control flow and resources.

In our further work we will focus on the implementation of the presented tool prototypes, their integration with the overall COMPAS architecture [D2.1, DA.1] and application to
COMPAS case studies [D6.1]. Theoretical results presented here will be extended to address other categories of compliance requirements identified in [D2.2].

9. Reference documents

9.1. Internal documents


9.2. External documents


