Universität Stuttgart

Fakultät Informatik, Elektrotechnik und Informationstechnik

A Model-Driven Approach to Implementing Coordination Protocols in BPEL

Oliver Kopp, Branimir Wetzstein, Ralph Mietzner,
Tobias Unger, Stefan Pottinger, Dimka Karastoyanova,
Sabine Michael, Frank Leymann

Report 2008/02

Institut für Architektur von Anwendungssystemen

Universitätsstraße 38
70569 Stuttgart
Germany

CR: H.4.1, K.4.4

Abstract WS-Coordination defines a framework for establishing protocols for coordinating the outcome agreement within distributed applications. The framework is extensible and allows support for multiple coordination protocols. To facilitate the realization of new coordination protocols we present a model-driven approach for the generation of BPEL processes used as implementation of coordination protocols. We show how coordination protocols can be modeled in domain-specific graph-based diagrams. Then we describe how to transform such graphs into abstract BPEL process models representing the behavior of the coordinator and the participants in the protocol. We present a prototypical implementation of the approach.

1 Introduction

Web services are the most recent middleware technology for application integration within and across enterprises [CLS$^+$05]. Through the use of standards like SOAP, WSDL, UDDI, and BPEL this technology enables interoperable service interactions in heterogeneous environments. Coordination is an important technology used in distributed computations with multiple participants that must jointly agree on the outcome of the computation. A well-known example for the use of coordination are distributed transactions using atomic commitment protocols to agree on the success or failure of a transaction [JG93]. The aspect of coordination in the domain of Web services is addressed by WS-Coordination [OAS07d]: it defines an extensible framework for coordinating the outcome of a set of Web services contributing to a distributed computation using a generalized notion of a coordinator and so called coordination protocols. In the context of WS-Coordination, coordination protocols describe the messages exchanged between the coordinator and the participants of a distributed computation and thus realize a one-to-many coordination. Two types of protocols (aka coordination types) have already been defined to cover "traditional" atomic transactions (WS-AtomicTransaction) and long-running business transactions (WS-BusinessActivity). However, the use of WS-Coordination is not restricted to transaction processing systems. Other types of protocols can also be defined for distributed computations such as protocols describing auctions [LP05], protocols for split BPEL loops and split BPEL scopes [KL07] and protocols for externalizing the coordination of BPEL scopes as a whole [PML07].

Coordination protocols can be quite complex. The coordinator has to deal with a variable number of participants. Each participant is in a well-defined state which potentially differs from the state of another participant at the same time. The implementation of a coordination protocol is difficult and error-prone. To simplify and accelerate the implementation, and eliminate errors, we propose a model-driven approach (MDA) in this paper.

The protocol is first modeled as a state-based graph, which we call coordination protocol graph (CPG). A CPG captures the different states and state changes based on the messages exchanged between coordinator and participant. The graph diagram is the domain specific language (DSL) we use for specifying coordination protocols. It contains only those elements which are needed for coordination protocol modeling and is therefore well suited for protocol designers. In MDA terms a coordination protocol graph specifies a Platform Independent Model (PIM) [Fra03]. The CPG is independent of any hardware or programming platform.

In general, coordination protocols define a sequence of steps and messages to be exchanged between participants in a coordinated interaction, timing issues, and how exceptional situations must be tackled. In that respect, modeling coordination protocols is similar to modeling business processes. The standard language for Web services based process descriptions is BPEL. In this work we generate abstract BPEL processes for both the coordinator and the participant roles in coordination protocols. These BPEL process models capture the essential parts of the message exchange between the parties and the resulting protocol state changes. The generated code reduces the need for tedious and error-prone programming concerning the communication between the coordinator and participants in the protocol. Additional protocol logic, which cannot be captured in the CPG, has to be manually added by the programmer.

The generated BPEL process models are platform specific models (PSM): they are compositions of Web Services and hence focus on the Web Services Platform, as the participants in a WS-Coordination protocol are assumed to be Web Services. As an alternative for the implementation of coordination protocols, Java code could also be generated. BPEL, however, enables programming on a higher abstraction level which makes the code generation easier. It has native WSDL support needed for interoperability and native support for concurrency, backward and forward recovery. BPEL engines persistently store all events related to process execution in an audit log and thus automatically support reliable recording of protocol execution out of the box. The audit log enables checking the execution of coordination for compliance [SM01].

The rest of the paper is organized as follows: Section 2 gives an overview of BPEL and WS-Coordination. In Section 3 we present the syntax and semantics of the coordination protocol graph (CPG). After depicting our model-driven approach in Section 4, we describe the generation of the BPEL process models in Section 5. In Section 6 we describe how we implemented the approach as an Eclipse plug-in. We finalize with the discussion of related work, conclusion and future work.

2 Background

2.1 WS-Coordination

WS-Coordination [OAS07d] defines an extensible framework for coordinating interactions between Web services. Coordinated interactions are called (coordinated) activities in the context of WS-Coordination. The framework enables participants to reach agreement on the outcome of distributed activities using a coordinator and an extensible set of coordination protocols. Two specifications already exist that build on WS-Coordination and define transaction protocols, namely the WS-AtomicTransaction [OAS07a] specification for atomic (2PC) transactions and the WS-BusinessActivity [OAS07b] specification for long-running transactions.

The framework defines three services a coordinator has to provide: activation service, registration service, and protocol services. When an application, in the role of an initiator, wants to start a coordinated activity, it requests a coordination context from an activation service. The coordination context contains an activity identifier, the coordination type (e.g. atomic transaction) as requested by the initiator, and the endpoint reference of the registration service. When the initiator distributes work, it sends the coordination context with the application message to the participant. Before starting work, the participant registers at the registration service of the coordinator. At some later point the protocol service is started which coordinates the outcome according to the specific protocol of the coordination type.

While the logic of the activation and registration service are fixed, the framework allows the definition of arbitrary coordination types, which specify different protocol services. In this paper we demonstrate how these protocol services can be implemented automatically using a model-driven approach. The implementation of the protocol services in this work is in BPEL. In the following when referring to “coordinator” and “participant”, we mean the protocol service implementations at the coordinator and participant, respectively.

2.2 BPEL

The Web Services Business Process Execution Language [OAS07c] is an orchestration language for Web services. A BPEL process is a composition of Web services, which are accessed through partner links referencing WSDL port types. The process is itself exposed as a Web service.

The BPEL process model comprises two types of activities: basic activities cope with invoking other Web services (invoke), providing operations to other Web services (e.g. receive and reply), timing issues and fault handling; structured activities nest other activities and deal with parallel (flow) and sequential execution (sequence), conditional behavior and event processing. Process data is stored in variables, while the assign activity is used for data manipulation. Activities can be enclosed in scopes to denote sets of activities that are to be dealt with as a unit of work. Scopes can be modeled to ensure all-or-nothing behavior, support data scoping, exception handling, compensation, and sophisticated event handling. Instance management is done using correlation sets. Correlation sets define which fields in incoming messages are to be used as identifiers to route the messages to one of possibly several running instances of the same process model.

BPEL processes can be either abstract or executable. An executable BPEL process provides a process model definition with enough information to be interpreted by a BPEL process engine. An abstract BPEL process hides some of the information needed for execution. Abstract process profiles define what information may be hidden. The profile used in our approach is the abstract process profile for templates. It allows marking sections of the process model as “opaque” using opaque tokens. It is thus explicitly specified which sections of the process model have to be later replaced by concrete activities, expressions etc. to make the process executable.

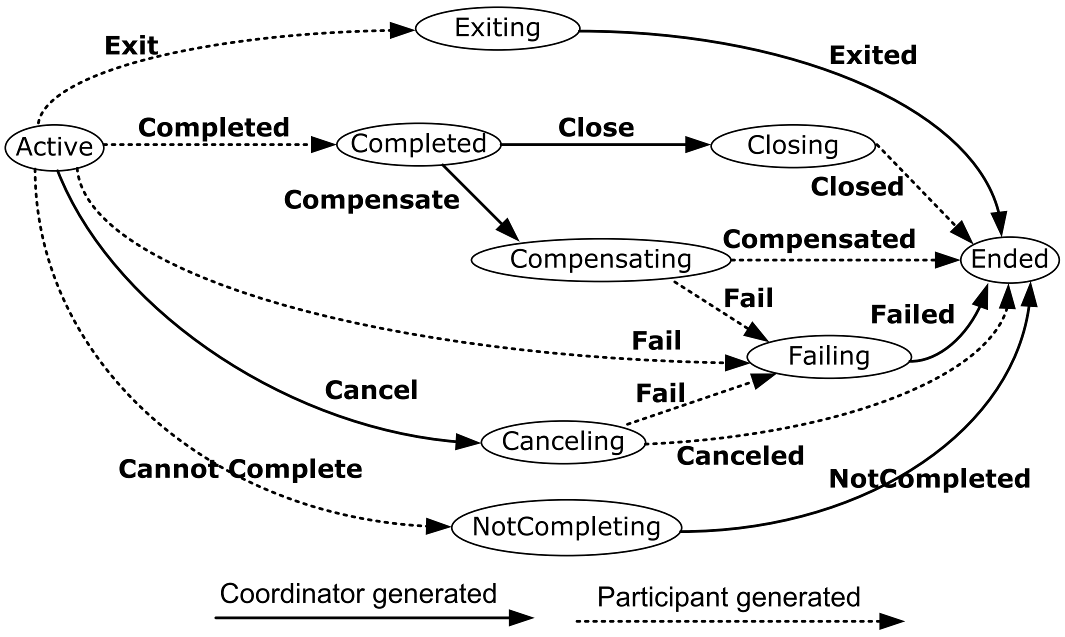


Figure 1: WS-BA with Participant Completion Protocol [OAS07b]

3 Modeling Coordination Protocols

There is no standard notation for modeling coordination protocols. The specifications in this area use either a proprietary or a generic diagram type (e.g. UML sequence diagram), or a combination of these. For modeling coordination protocols we have adopted the diagram type from the WS-AtomicTransaction (WS-AT) and WS-BusinessActivity (WS-BA) specifications. This diagram type can be seen as a domain specific language for modeling coordination protocols. WS-BA contains two protocols: *WS-BA with Participant Completion*, where the participant signals when it has completed its work and *WS-BA with Coordinator Completion*, where the coordinator notifies the participant when it has to complete his work. Figure 1 shows the WS-BA with Participant Completion protocol as an example, which we will also use in the rest of the paper for illustration of mapping concepts. It is important to note that it is possible to use any coordination protocol graph as input, such as the ones presented in [PML07].

The diagram defines a state-based graph, which we name coordination protocol graph (CPG). A CPG is a directed graph with labeled edges and labeled nodes. The nodes denote the states of the coordination protocol between a coordinator and a participant. The node labels describe the semantics of the states. The edges depict the messages exchanged by the protocol parties; the edge labels describe the semantics of the message. Since messages can be sent by a participant and by a coordinator, the set of all edges is divided into two disjoint sets: edges denoting coordinator messages (solid lines) and edges denoting participant messages (dashed lines). Each CPG has exactly one node with no incoming edges (source) and at least one node without outgoing edges (sink). No two coordinator edges or participant edges with the same label may leave the same node, because this would lead to non-determinism. A CPG does not contain cycles.

The semantics of the graph is as follows: A CPG models a state machine which describes the states and possible state changes of the protocol between the coordinator and one participant registered for the coordination activity. At a certain point in time each participant can be in a different states. For example, one participant can be in the state “Failing” while another is in the state “Closing”. Since coordinator usually interacts with more than one participant, the coordinator has to hold the state of each state machine.

Outgoing edges of a CPG denote messages which may be sent and each state denotes the possible state of a participant. The sender of the message (participant or coordinator) transitions to the next state when sending a message. The recipient of the message transitions to the next state when receiving the message. For the period of time when the message is transported, the coordinator and participant thus are in different states. In addition to the obvious behavior of state changes there are three special cases: (i) ignoring same messages which are sent more than once, (ii) precedence of participant messages over coordinator messages, (iii) invalid messages.

If the message leading to a new state is received more than once, it is simply ignored. For example, if the coordinator being in state “Exiting” receives the message “Exit” again, that message is ignored. This case can arise, when messages are resent because it is suspected that the first message hasn’t been transmitted successfully.

If a state has both outgoing participant and coordinator messages, then it can happen that the coordinator sends a protocol message and enters the corresponding new state, but later receives a protocol message from the participant which is consistent with the former state. This can happen when both the coordinator and the participant send their messages at about the same time, which leads to different views on the protocol state on coordinator and participant side. In that case the participant messages have precedence over coordinator messages. In Figure 1 the state is “Active” at the beginning of the protocol. Let us assume the coordinator sends “Cancel” to the participant and sets the state to “Canceling”. At the same time, however, the participant sends the message ”Completed” and changes his state to “Completed”. When the coordinator receives the message “Completed” while being in state “Canceling” for the participant, he has to revert to the former state “Active”, accept the notification “Completed” and change the state to “Completed”. The participant on the other side just discards the coordinator message “Cancel”.

Finally, if in a state other messages than the allowed ones are received, a fault message should be generated and sent to the sender of the invalid message. The protocol execution is aborted.

It is important to note, that a CPG captures only the possible interactions and state changes between the coordinator and participant. A CPG does not capture the reason of these state changes. For example, if a participant is in the state “Completed” it can receive either a “Close” or a “Compensate” message from the coordinator. Which of the two messages is sent, is part of the protocol logic. For example, if another participant has failed and all-or-noting semantics is needed a “Compensate” message would be sent. Because not all of the protocol logic is captured by the graph, it has to be additionally implemented after the generation of the BPEL process.

The CPG and its semantics are derived from WS-BA and WS-AT protocols. In summary, the CPG graph captures the exchanged messages between a coordinator and a participant, and the resulting state changes, however not the cause of the state changes.

4 Model-Driven Implementation Approach

For the implementation of coordination protocols we adopt a model-driven approach. Our goal is to model the coordination protocol using a domain-specific language suitable for coordination protocol designers, and then generate BPEL code which implements the coordination protocol.

The DSL, in our case the CPG, is used for creating a platform-independent model (PIM) of the coordination protocol. The PIM can be transformed to platform-specific models (PSM) for different kind of platforms. In this paper we use BPEL and the Web service platform, in particular WS-Coordination as the coordination framework.

As the CPG does not contain enough information to be executed, the additional information has to be added to the PSM after generation. We thus do not achieve 100% BPEL code generation, but still avoid much of tedious and error-prone programming. The use of a model-driven approach ensures higher productivity in development and better quality of the implemented code.

We have decided to use BPEL as the platform for the implementation of coordination protocols. It has several benefits when compared to a 3GL programming language such as Java. BPEL enables programming on a higher abstraction level which makes the generation easier in our case. As BPEL supports graph-based models, coordination protocol graphs can be more easily and naturally transformed into BPEL. BPEL has native WSDL support needed for interoperability in WS-Coordination and native support for concurrent execution, which is more difficult to implement in Java. A BPEL engine persistently stores all events related to process execution in an audit log and thus automatically supports reliable recording of coordination protocol execution out of the box. The audit log enables checking the execution of coordination for compliance with the protocol. A BPEL engine typically also provides a monitoring tool, which enables observing the execution of coordination protocols in real-time. Finally, as the state of a BPEL process instance is persistently saved after each state transition, the coordination protocol can be stopped and resumed at any time using a monitoring tool.

The approach is shown in Figure 2. In the first step, the CPG is created using a corresponding graphical CPG modeling tool. The CPG models the interaction between the coordinator and the participant in a platform-independent way.

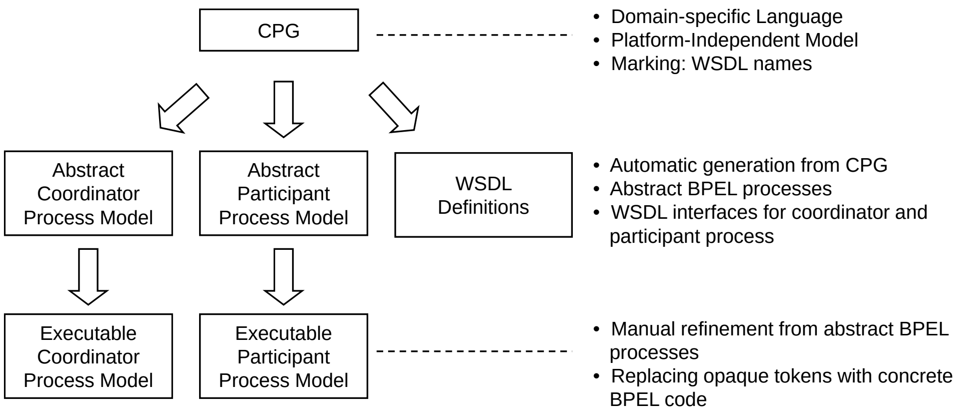

CPG
• Domain-specific Language
• Platform-Independent Model
• Marking: WSDL names
Abstract Coordinator Process Model
Abstract Participant Process Model
WSDL Definitions
• Automatic generation from CPG
• Abstract BPEL processes
• WSDL interfaces for coordinator and participant process
Executable Coordinator Process Model
Executable Participant Process Model
• Manual refinement from abstract BPEL processes
• Replacing opaque tokens with concrete BPEL code

Figure 2: Model-Driven Implementation Approach

In the next step, the CPG is transformed into two abstract BPEL processes, one for the coordinator and one for the participants. Therefore, the abstract BPEL processes and corresponding WSDL definitions are generated. If the WSDL definitions already exist, as for example in the case of the WS-AT and WS-BA specification, then the CPG has to be correspondingly marked. One has to specify the names of the WSDL port types for both the participant and coordinator process, the WSDL message and operation names, which correspond to the labels of the state transitions in the CPG, etc. That ensures that the generated BPEL processes and WSDL descriptions are compliant to the already existing probably standardized ones. The two corresponding WSDL interface descriptions of the processes can be completely generated. Using standardized WSDL interfaces ensures that the coordinator process can be used to coordinate arbitrary protocol participants apart from the generated BPEL-based participants. This is also the case for the generated participants, which can be used with another protocol-compliant coordinator. Thus, our approach supports heterogeneous environments.

As discussed in the previous section, the generated process models cannot be executable, because the CPG does not capture the whole protocol logic. The locations in the process model where missing logic has to be added are "marked" in the generated BPEL code using opaque tokens, as defined in the abstract process profile for templates [OAS07c]. These opaque tokens show to the developer where additional logic has to be added to make the process executable. The abstract BPEL process profile for observable behavior [OAS07c] cannot be used, since it does not allow the addition of interaction activities with existing partner links when replacing opaque activities. However, that is needed in certain cases: For example, in the coordinator process after the interaction activity receiving a "Fail" from one participant, one might want to add interaction activities (BPEL invoke) which send "Cancel" notifications to other participants.

As already described in Section 2.1, WS-Coordination defines three services a coordinator has to provide: activation, registration, and protocol services. While protocol services can be additionally defined in separate specifications such as WS-BA, the implementation of the activation and registration services stays the same. The activation and registration service of the coordinator can thus be fully generated. Both services in addition to the protocol service are implemented by the coordinator process model (see Section 5.3).

After generation, the abstract process models are refined manually by a developer who replaces the opaque tokens by the missing coordination protocol logic. The resulting executable BPEL process models can finally be deployed on a BPEL engine.

5 Generating BPEL Process Models

In the following we describe in detail how CPG graphs are transformed to abstract BPEL process models. We generate two abstract BPEL process models, one for the coordinator and one for the participants.

We have chosen different approaches for the generation of the two process models. For the participant process model, we keep the graphical structure of the CPG in the BPEL process model by mapping the CPG graph directly to a BPEL flow. The BPEL flow activity together with BPEL links enables graph-based workflow modeling. The generated BPEL process structure closely resembles the CPG structure and thus increases the readability of the process.

For the coordinator process model the participant approach is not feasible, since the coordinator holds a different state for each participant (Section 5.3). In the second approach we define global event handlers for each message that can be received by the coordinator. That means, we implement a state-machine by specifying rules of the form: if received message x, then perform some logic which handles that message x.

5.1 Generation of the Participant Process Model

The general idea of the mapping is to keep the graphical structure of a CPG in the BPEL process model (Figure 3): the CPG graph as a whole is mapped to a BPEL flow element; each state of the CPG is mapped to a BPEL scope which internally implements the logic of that state; each edge of the CPG is mapped to a BPEL link connecting two scopes. The navigation through the process model is driven using a state variable evaluated in the transition conditions of the links. The protocol logic inside the scope determines the next state of the protocol and sets the value of the state variable correspondingly. The transition conditions are mutually exclusive, because only one scope can be active at a time.

BPEL employs a concept called dead path elimination (DPE) which has to be activated in our case. DPE is a mechanism that propagates the disablement of activities down the execution path so that following activities do not wait forever (deadlock). The activation of DPE leads to following behavior: If a scope is not entered because the transition

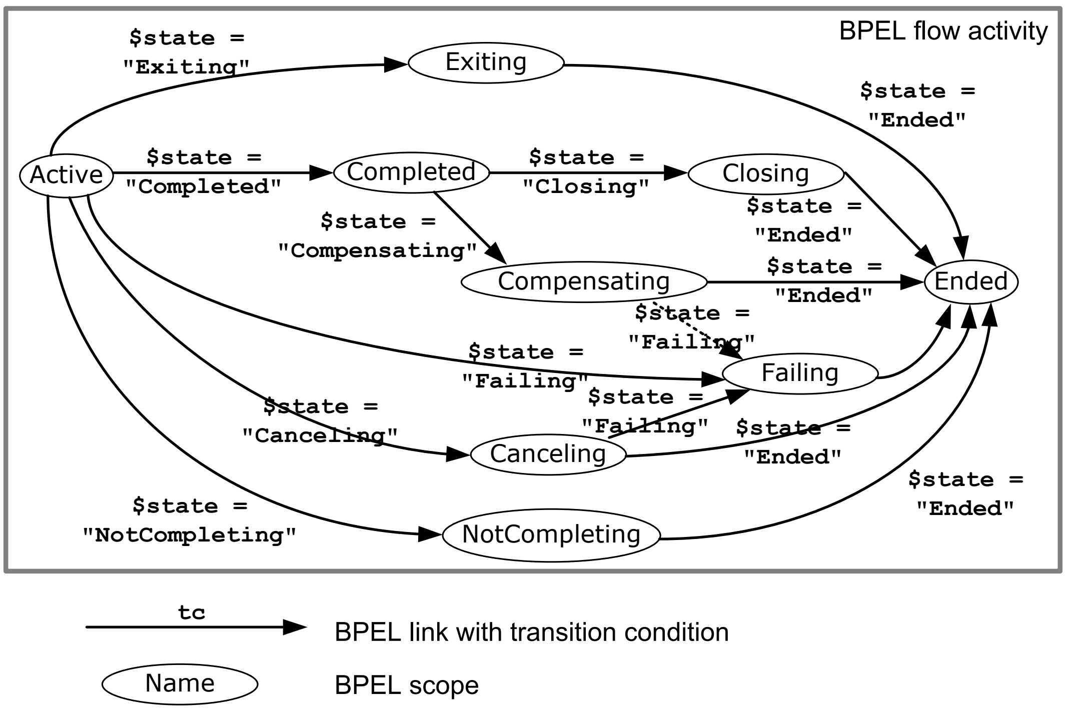


Figure 3: Participant process model

condition of the incoming links is false, the transition conditions of the outgoing links are fired and automatically set to false. As an activity cannot start until of its incoming links have fired, DPE ensures that an activity doesn't wait forever for the firing of its incoming links. After all incoming links have fired, an activity uses a join condition to evaluate the transition conditions of its incoming links. The default join condition, which is sufficient for our process models, is a disjunction on all incoming links, i.e. at least one of the transition conditions has to evaluate to true. If for example the participant wants to go from the state "Active" into the state "Failing", the state variable is set to "Failing" and the corresponding link fired, however the scope "Failing" cannot be started, until also the other incoming links from "Canceling" and "Compensating" have fired. These links are activated by the DPE mechanism. For a detailed explanation of DPE in general see [CKLW03].

Thus, each state in the CPG can be mapped to a BPEL scope and connected by links which are derived from the edges connecting the states in the CPG. The transition conditions on the links ensure that the target scope will only be activated if the protocol has reached the state represented by the target scope. The incoming links to the scope ensure that the scope is only executed, if all paths to the scope are visited (either by execution or by disablement).

The generation of the process model can be done by traversing the CPG in a depth-first search generating the scopes as soon as the corresponding nodes are visited.

The implementation of the scopes copes with the communication between coordinator and participant and the resulting state changes. The places where protocol specific logic not captured by the CPG has to be implemented are marked with opaque activities.

In the following sections we will describe the implementation of the scopes in detail. For the sake of simplicity and because of verbosity of BPEL code we will not use the complete BPEL syntax. For example, we use `$state := successor state i` instead of following syntax:

```
<assign>
  <copy>
    <from>i</from>
    <to>$sate</to>
  </copy>
</assign>
```

5.2 Scope Implementation in the Participant Process Model

The communication allowed between coordinator and participant in a particular state is described by the state's outgoing edges; the incoming edges are handled by the predecessor scopes. In case of an outgoing participant message, the scope in the participant process model has to send a message to the corresponding scope in the coordinator process model. In case of an outgoing coordinator message it has to receive a message from the coordinator. For the generation of scopes, four cases have to be distinguished:

(i) The state has no outgoing edges.

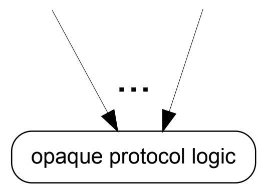
(ii) The state only has outgoing participant edges.

(iii) The state only has outgoing coordinator edges.

(iv) The state has both outgoing coordinator and participant edges.

All states belonging to the first case are end states. All other states may be start states or states on a path from a start state to an end state.

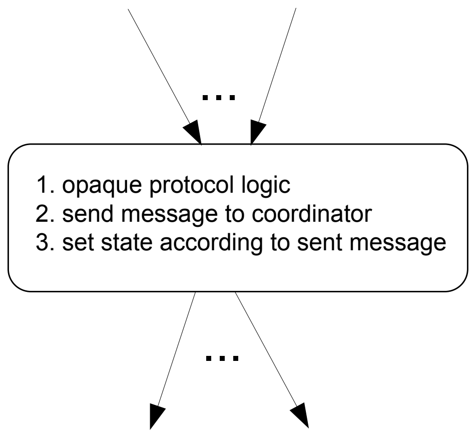
Independent of these cases, the WS-BA specification requires that invalid coordinator messages have to be handled. For example, the message "Close" must not be sent to a participant in the state "Active". Thus, for each invalid coordinator message, an event handler for that message is generated in the respective state. If the coordinator message is valid for the previous state, the message is ignored. Otherwise, the event handler replies to the message with the "Invalid State" fault defined in WS-Coordination. To handle the "getStatus" message, another event handler is added. The current status is read from the status variable and sent back to the coordinator. For each given CPG graph, a this additional logic is generated.



```
<scope name="state">
  <targets> … </targets>
  <opaqueActivity/>
</scope>
```

Graphical BPEL code representation   Generated BPEL code

Figure 4: Generated scope for a state with no outgoing messages


...
1. opaque protocol logic
2. send message to coordinator
3. set state according to sent message
...

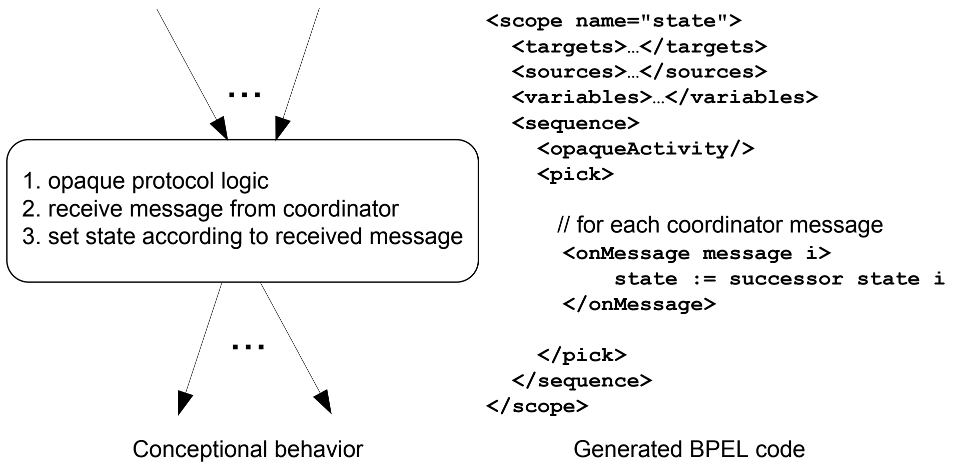
```
<scope name="state">
  <targets>…</targets>
  <sources>…</sources>
  <variables>…</variables>
  <sequence>
    <opaqueActivity/>
    <if>
      <condition opaque="yes"/>
      <invoke coordinator, m1_op />
      $state := successor state 1

    // one elseif for successor state 2 to i-1
    <elseif> … </elseif>

    <else> … </else>
    </if>
  </sequence>
</scope>
```

Conceptional behavior   Generated BPEL code

Figure 5: Generated scope for a state with outgoing participant messages

Case (i): State without Outgoing Messages For a state without outgoing edges, no send or receive logic has to be implemented. The scope thus contains only an opaque activity, which is used to denote that the programmer can insert protocol logic there (Figure 4).

Case (ii): State with Outgoing Participant Messages Only In this case, there are no coordinator message received, but a message sent to the coordinator. The scope is implemented as follows: The participant first executes some opaque protocol logic corresponding to the state. There it decides what message to send, for example whether to commit or abort a transaction. Finally it sends this message to the coordinator and assigns the corresponding successor state to a global state variable. That global state variable holds the state of the protocol and is used in the transition conditions to navigate to the next scope. For each outgoing participant message an if-branch with an opaque condition is generated. The last branch (the else branch) does not contain any condition to ensure that a message is sent in all cases (Figure 5).


...
1. opaque protocol logic
2. receive message from coordinator
3. set state according to received message
...
Conceptional behavior
Generated BPEL code

```
<scope name="state">
  <targets>…</targets>
  <sources>…</sources>
  <variables>…</variables>
  <sequence>
    <opaqueActivity/>
    <pick>

      // for each coordinator message
      <onMessage message i>
          state := successor state i
      </onMessage>

    </pick>
  </sequence>
</scope>
```

Figure 6: Generated scope for a state with outgoing coordinator messages

Case (iii): State with Outgoing Coordinator Messages Only In this case, where no message may be sent, but a coordinator message is received, a BPEL pick activity is used. For each coordinator message an onMessage handler is generated (Figure 6). The variables used to store the received messages are defined at the beginning of the scope. Before the coordinator message is processed, the participant is executing protocol logic corresponding to the state. After a message is received, the corresponding state is set.

Case (iv): State with Both Outgoing Participant and Coordinator Messages This case is addressed by the construct presented in Figure 7. There are two paths of execution now, which are executed in parallel. (i) Executing the participant protocol logic and (ii) Handling incoming coordinator messages. (i) As in the other cases, the main path first executes some opaque protocol logic. Based on an opaque condition it is decided whether a message has to be sent to the coordinator or a coordinator message has to be received. If a coordinator message is to be received and it hasn't yet arrived, the main path is waiting for it using a pick activity. (ii) The other path of execution is needed for handling a message from the coordinator, which can arrive during the execution of the opaque activity in the main path. The coordinator message is received using a BPEL event handler attached to the opaque activity. When the coordinator message is received by the event handler, the opaque activity can (a) ignore the coordinator message, (b) set variables to signal the receipt of the message to the main path, or (c) decide to cancel processing and to move to the state belonging to the coordinator message. If the third option is chosen, the opaque condition on the link at the event handler evaluates to "true" and the successor_state_i fault is thrown. This cancels the execution of the other activities in the scope. The opaque activity in the fault handler allows a clean up after canceling the processing.

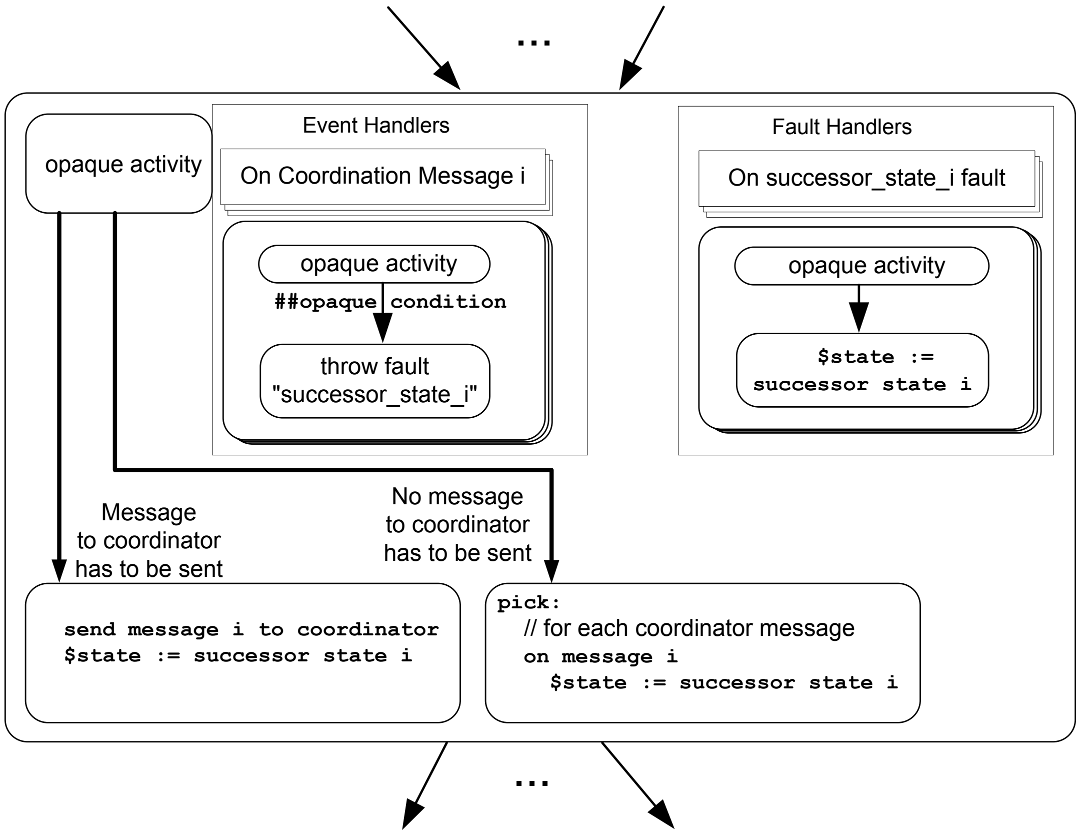

...
opaque activity
Event Handlers
On Coordination Message i
opaque activity
##opaque condition
throw fault "successor_state_i"
Fault Handlers
On successor_state_i fault
opaque activity
$state := successor state i
Message to coordinator has to be sent
No message to coordinator has to be sent
send message i to coordinator
$state := successor state i
pick:
// for each coordinator message
on message i
$state := successor state i
...

Figure 7: Generated scope for a state with outgoing participant and outgoing coordinator messages

5.3 Generation of the Coordinator Process Model

The participant mapping approach is not feasible for the coordination process model. The coordinator has to be able to cope with several participants. The coordinator cannot leave the scope “Active” until all registered participants have been handled for that scope. In the meantime, however, several participants could have declared that they want to exit the protocol by sending the message “Exit” to the coordinator. In that case the coordinator should immediately send the notification “Exited” to the participant. However, this is not possible, since the coordinator is in the scope “Active” and waits for other participants to complete their work. When the coordinator finally leaves the scope “Active”, a new participant could register for the protocol. Since the scope “Active” has already finished, the new participant cannot be handled.

Figure 8 illustrates the pattern for the implementation of the coordinator scope. An instance of the coordinator process model is started when a new WS-Coordination activity is created. This is done by an application by sending a “CreateCoordinationContext” message to the coordinator endpoint which replies with a “CreatedCoordinationContext” message to indicate successful creation of the context.

Having received such a message the coordinator process is now ready to accept registration messages from participants that wish to participate in the coordination, and to react on messages sent by participants that have already registered. The coordinator leaves this state if the application determines that the coordination should end and sends a corresponding message.

The abstract BPEL process template for the coordinator is generated as follows: In order to manage the participants for the activity an array is generated that holds the status of all participants of the activity as well as the endpoint references of the participants. The endpoint references are obtained during registration and are needed to send coordination messages to the right endpoints.

Regarding the control flow, at first a process instance creating receive activity is added that is triggered by WS-Coordination “CreateCoordinationContext” messages. The user can then replace the following opaque activity by inserting arbitrary BPEL activities that handle the message. Afterwards the confirmation for the successful creation of the coordination context is sent. The control-flow now enters the scope that handles the coordination protocol specific messages as well as the registration of participants for the activity. Both types of messages are received and handled via event handlers.

We place opaque activities throughout the process template during generation to allow the actual coordination logic to be inserted as needed. We do not explicitly mention those in the following discussion, but Figure 8 shows where the opaque activities are placed in detail. In the following description we concentrate on the control flow and leave out details such as correlation of messages to the right process instance. For now, we assume that upon reception of each message the coordinator knows which participant has sent the message and that messages only are received by coordinator process instances that handle the participant that has sent the message. Means to ensure these assumptions

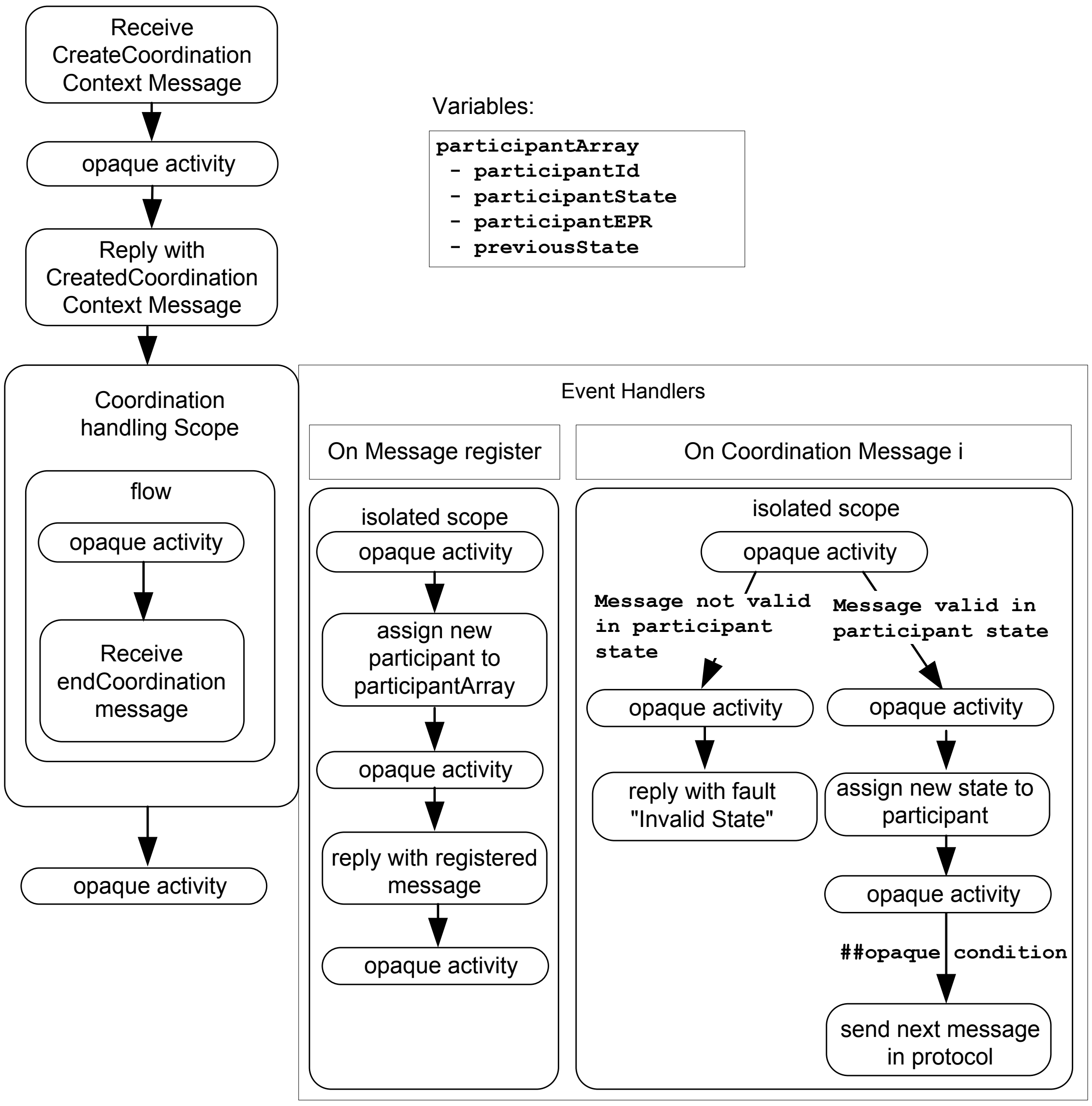

Receive CreateCoordination Context Message
opaque activity
Reply with CreatedCoordination Context Message
Coordination handling Scope
flow
opaque activity
Receive endCoordination message
opaque activity
Variables:
participantArray
- participantId
- participantState
- participantEPR
- previousState
Event Handlers
On Message register
isolated scope
opaque activity
assign new participant to participantArray
opaque activity
reply with registered message
opaque activity
On Coordination Message i
isolated scope
opaque activity
Message not valid in participant state
Message valid in participant state
opaque activity
reply with fault "Invalid State"
opaque activity
assign new state to participant
opaque activity
##opaque condition
send next message in protocol

Figure 8: Pattern for the generation of coordinator scopes

are presented in Section 5.4.

Registration of Participants As shown in Figure 8 registration of participants is handled via a dedicated event handler. The event handler includes an assign activity that adds the new participant into the participant array and sets its current state to the first state that follows registration in the coordination protocol the coordinator has been created for. Afterwards the event handler responds with a "Registered" message. Both "Register" and "Registered" messages are defined in WS-Coordination. Opaque activities allow the handling of special cases by special coordination logic. Such a case may be the reception of registration messages after other participants have already faulted or completed. For example, WS-BA demands that such cases are allowed.

Handling of Protocol Specific Messages For each protocol specific message the coordinator can receive (i.e., for each dashed line in a CPG) a separate event handler is created that handles that type of message. Upon receipt of a participant message, one out of two paths can be followed: The first path is followed if the message is not allowed in the state of the participant. In that case the "Invalid State" message is sent back. In case the message is allowed in the current state, the state of the participant is updated via an assign activity. The generated model contains opaque activities that can be replaced by arbitrary BPEL activities that perform the actual coordinator logic. For example, one or more invoke activities can be inserted that send the corresponding messages that follow the received message in the coordination protocol. However, the decision on whether a message is sent and which message in particular is sent depends on the actual coordinator logic and therefore is marked as opaque and needs to be completed during the customization of the template.

The second path also handles two special cases: (i) ignoring messages which were resent by the participant, (ii) reverting to a previous state. Both the WS-AtomicTransaction and the WS-BusinessActivity specification demand that not only messages that are allowed in the current state of the participant are allowed but also messages corresponding to the previous state of the participant. In order to comply with this demand an additional field in the array is introduced that stores the previous state. On reception of a message a new assign activity is introduced that reverts the state of a participant if a message corresponding to that state is received. Then the control flow can proceed as if it had originally received the message in the correct state.

Concurrent Reception of Messages All messages that can be received concurrently by the coordinator are handled by event handlers. Thus, we ensure that the BPEL engine can deal with the concurrent message reception. However in order to ensure that concurrent access to shared variables, such as the participant arrays, and resulting problems are avoided the logic of the event handlers is placed in isolated scopes. An isolated scope is a BPEL means to synchronize parallel access to variables.

5.4 Correlation Issues

When a coordinator sends a message to a participant or vice versa, the BPEL engine has to know to which process instance this message has to be routed to. This is because (i) in the coordinator case several coordinated activities and thus process instances can be active at the same time; (ii) in the participant case several participants can be involved in a coordination activity. This problem is handled by BPEL's correlation mechanism.

When a new coordination activity is to be created, the initiator sends a "CreateCoordinationContext" message to the coordinator process endpoint. A new coordinator process instance is created, and a coordination context identifier for the new coordination activity is generated. Together with the endpoint reference (EPR) of the registration service it is sent back to the initiator. When an application decides to participate in the coordination activity it creates a new participant process instance by sending the EPR and the coordination context identifier it got from the initiator to the participant process endpoint. A new participant process is created and after generating a participant identifier, the participant process sends it together with the coordination context identifier to the registration service, which is again implemented by the coordinator process model. The coordinator saves the EPR of the participant and the participant identifier in the participant array as described in Section 5.3.

For the correlation to work, the coordinator process model creates a correlation set which includes the coordination context identifier; the participant process model includes additionally the participant identifier in the correlation set. Both identifiers are transported in message headers as defined by WS-Addressing [W3C06]. The BPEL engine has to support WS-Addressing and access to the message headers in the process model. The messages which are exchanged between the coordinator and the participant thus have to contain the coordination context identifier and the participant identifier in the message header.

6 Implementation

We have implemented a graphical CPG editor as an Eclipse plug-in based on the EMF[1] and GEF[2] frameworks [Mic06]. The editor supports the creation of CPG graphs and the generation of the abstract BPEL process models and the corresponding WSDL files. The WSDL namespace, port type, operation and message names can be specified in the editor by setting the corresponding attributes. If not explicitly specified, the operation and message names are derived from the edge labels in the CPG.

After generation of the abstract process models, the additional logic can be inserted into the process models replacing the opaque activities by using a BPEL editor like the

[1]Eclipse Modeling Framework. `http://www.eclipse.org/emf/`

[2]Graphical Modeling Framework. `http://www.eclipse.org/gmf/`

ActiveBPEL Designer[3] or a BPMN-based BPEL Designer [Sch08].

7 Related Work

There are several approaches to map business processes modeled graphically to BPEL (e.g. [MLZ06, ODBH06]). The approaches are similar to our work, since they are also generating BPEL processes, but the authors deal with generating a single BPEL process: they focus on orchestrations only. Hence, these approaches do not tackle the communication between processes as it is the case between the coordinator and the participant process.

In contrast to orchestrations, choreographies provide a global view on the interactions of all participants involved. If a coordination protocol is modeled in a choreography language, the messages from a coordinator to the participants have to be explicitly modeled. For example, if the coordinator receives the message "Fail" from one participant, it has to send out "Cancel" to all participants in the state "Active" and "Compensate" to all participants in the state "Completed". That implies that the modeling language has to offer constructs to model states of participants and the possibility to model the state change. In addition, the participants are unknown in advance. Thus, the modeling language has to be capable of modeling sets of participants with a-priori unknown size of the set.

Currently, there are two kinds of choreography models: (i) interconnection models and (ii) interaction models [DKB08]. An *interconnection model* contains the observable behavior description of each participant and an interconnection between each behavior description. BPMN [Obj08] and BPEL4Chor [DKLW07] are the most popular languages realizing interconnection models. Both of them support to model sets of participants: BPMN by extensions [DP07] and BPEL4Chor natively. However, both of them do not offer modeling constructs to express that a participant is removed from one set and put into another set. In contrast to interconnection models, *interaction models* describe the message exchanges between the participants. Two languages realizing interaction models are WS-CDL [KBRL05] and Let's Dance [ZBDH06]. While there is a mapping to BPEL available for WS-CDL [MH05], WS-CDL does not support modeling a choreography with an a-priori unknown number of participants [DOZ06]. Let's Dance supports the modeling of choreographies without the need to explicit state the concrete participants. Sets of participants are modeled using different types. It is not possible to assign a new type to a participant. All in all, none of the existing choreography approaches can be used to model coordination protocols.

Another approach to model transactions is the UN/CEFACT's Modeling Methodology (UMM, [UN/06]). While UMM can be mapped to BPEL [HHL$^+$07], UMM does not support modeling of sets of a-priori unknown participants.

[3]ActiveBPEL Designer 2.0. `http://www.activebpel.org`

8 Conclusions and Future Work

The main contributions of this report are: (i) the introduction of a model-driven approach for implementing coordination protocols, (ii) the concrete transformation of the CPG graph to abstract BPEL process models.

We have shown how a WS-Coordination-based coordination protocol can be modeled as a CPG graph. A CPG graph captures the essence of a coordination protocol: the states of the protocol and messages produced by both the coordinator and the participant. The generated BPEL processes are abstract and comply with the abstract process profile for templates. Opaque activities and expressions mark the locations where the programmer can include additional protocol logic not captured by the CPG to make the processes executable.

We have defined the CPG to be acyclic. The protocols described in WS-AtomicTransaction and WS-BusinessActivity also do not use cycles. However, there are coordination protocols such as the protocol for split loops [KL07]. We intent to adopt the work of [ZHB+06] to address this issue in our future work.

References

[CKLW03] F. Curbera, R. Khalaf, F. Leymann, S. Weerawarana. Exception Handling in the BPEL4WS Language. In W. M. P. V. der Aalst, A. H. M. T. Hofstede, M. Weske, editors, *Business Process Management*, volume 2678 of *Lecture Notes in Computer Science*, pp. 276–290. Springer, 2003. doi:10.1007/3-540-44895-0_19.

[CLS+05] F. Curbera, F. Leymann, T. Storey, D. Ferguson, S. Weerawarana. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More.* Prentice Hall PTR, 2005.

[DKB08] G. Decker, O. Kopp, A. Barros. An Introduction to Service Choreographies. *Information Technology*, 50(2/2008), 2008.

[DKLW07] G. Decker, O. Kopp, F. Leymann, M. Weske. BPEL4Chor: Extending BPEL for Modeling Choreographies. In I. C. Society, editor, *Proceedings of the IEEE 2007 International Conference on Web Services (ICWS 2007), Salt Lake City, Utah, USA, July 2007*, pp. 296–303. IEEE Computer Society, Salt Lake City, 2007. doi:DOI:10.1109/ICWS.2007.59.

[DOZ06] G. Decker, H. Overdick, J. M. Zaha. On the Suitability of WS-CDL for Choreography Modeling. In *EMISA*. 2006.

[DP07] G. Decker, F. Puhlmann. Extending BPMN for Modeling Complex Choreographies. In *CoopIS*. 2007.

[Fra03] D. S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing.* Wiley, 2003.

[HHL$^+$07] B. Hofreiter, C. Huemer, P. Liegl, R. Schuster, M. Zapletal. Deriving executable BPEL from UMM Business Transactions. In *IEEE SCC*, pp. 178–186. IEEE Computer Society, 2007. doi:10.1109/SCC.2007.49.

[JG93] A. R. Jim Gray. *Transaction Processing: concepts and techniques.* Morgan Kaufman Publishers, 1993.

[KBRL05] N. Kavantzas, D. Burdett, G. Ritzinger, Y. Lafon. *Web Services Choreography Description Language Version 1.0, W3C Candidate Recommendation*, 2005. `http://www.w3.org/TR/ws-cdl-10`.

[KL07] R. Khalaf, F. Leymann. Coordination Protocols for Split BPEL Loops and Scopes. Technical Report Computer Science 2007/01, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, University of Stuttgart, Institute of Architecture of Application Systems, 2007.

[LP05] F. Leymann, S. Pottinger. Rethinking the Coordination Models of WS-Coordination and WS-CF. In *Third IEEE European Conference on Web Services (ECOWS 2005)*, pp. 160–169. IEEE Computer Society, 2005. doi:10.1109/ECOWS.2005.20.

[MH05] J. Mendling, M. Hafner. From Inter-Organizational Workflows to Process Execution: Generating BPEL from WS-CDL. In *OTM*. 2005.

[Mic06] S. Michael. *Generierung von BPEL mit Hilfe von koordinierten Kommunikations-Graphen auf Basis transaktionaler Protokolle für Web Services.* Diploma thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, 2006.

[MLZ06] J. Mendling, K. B. Lassen, U. Zdun. Transformation Strategies between Block-Oriented and Graph-Oriented Process Modelling Languages. In *XML4BPM*. 2006.

[OAS07a] OASIS. *Web Services Atomic Transaction (WS-AtomicTransaction). Version 1.1*, 2007. `http://docs.oasis-open.org/ws-tx/wstx-wsat-1.1-spec-os.pdf`.

[OAS07b] OASIS. *Web Services Business Activity Framework (WS-BusinessActivity). Version 1.1*, 2007. `http://docs.oasis-open.org/ws-tx/wstx-wsba-1.1-spec-os.pdf`.

[OAS07c] OASIS. Web Services Business Process Execution Language Version 2.0 – OASIS Standard. Technical report, Organization for the Advancement of Structured Information Standards (OASIS), 2007.

[OAS07d] OASIS. *Web Services Coordination (WS-Coordination) Version 1.1*, 2007. `http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.1-spec-os.pdf`.

[Obj08] Object Management Group. *Business Process Modeling Notation, V1.1 – OMG Available Specification*, 2008.

[ODBH06] C. Ouyang, M. Dumas, S. Breutel, A. ter Hofstede. Translating Standard Process Models to BPEL. In *Advanced Information Systems Engineering*, volume 4001, pp. 417–432. Springer Berlin / Heidelberg, 2006. doi:10.1007/11767138_28.

[PML07] S. Pottinger, R. Mietzner, F. Leymann. Coordinate BPEL Scopes and Processes by Extending the WS-Business Activity Framework. In R. Meersman, Z. Tari, editors, *Proceedings of the 15th International Conference on Cooperative Information Systems (CoopIS 2007)*, volume 4803 of *Lecture Notes in Computer Science*, pp. 336–352. Springer, 2007. doi:10.1007/978-3-540-76848-7_22.

[Sch08] D. Schumm. *Graphische Modellierung von BPEL Prozessen unter Verwendung der BPMN Notation.* Diploma thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, 2008.

[SM01] M. Sailer, M. Morciniec. Monitoring and Execution for Contract Compliance. Technical Report HPL-2001-261, Hewlett Packard Laboratories, 2001.

[UN/06] UN/CEFACT. *UN/CEFACT's Modeling Methodology (UMM), UMM Meta Model - Foundation Module*, 2006. Technical Specification V1.0, `http://www.unece.org/cefact/umm/UMM_Foundation_Module.pdf`.

[W3C06] W3C Recommendation. *Web Services Addressing Version 1.0*, 2006.

[ZBDH06] J. M. Zaha, A. Barros, M. Dumas, A. ter Hofstede. A Language for Service Behavior Modeling. In *CoopIS*. Montpellier, France, 2006.

[ZHB$^+$06] W. Zhao, R. Hauser, K. Bhattacharya, B. R. Bryant, F. Cao. Compiling business processes: untangling unstructured loops in irreducible flow graphs. *International Journal of Web and Grid Services*, 2:68–91, 2006. doi:10.1504/06.8880.

All links were last followed on May 8, 2008.