Supporting Business Process Fragmentation While Maintaining Operational Semantics: A BPEL Perspective

Von der Fakultät Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von
Rania Y. Khalaf

aus Beirut (Libanon)

Hauptberichter: **Prof. Dr. rer. nat. Frank Leymann**
Mitberichter: **Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel**

Tag der mündlichen Prüfung: 31. März 2008

Institut für Architektur von Anwendungssystemen der Universität Stuttgart
2008

Khalaf, Rania Y.:
Supporting Business Process Fragmentation While Maintaining Operational Semantics :
A BPEL Perspective / Rania Y. Khalaf. –
Als Ms. gedr.. – Berlin : dissertation.de – Verlag im Internet GmbH, 2008
Zugl.: Stuttgart, Univ., Diss., 2008
ISBN 978-3-86624-344-6

Bibliographic information published by Die Deutsche Bibliothek

Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data is available on the Internet at <http://dnb.ddb.de>.

Exclusively, totally chlorine free (TCF) bleached
paper is used according to DIN-ISO 9706.
Printed in Germany.

dissertation.de - Verlag im Internet GmbH
Pestalozzistr. 9
10625 Berlin
Germany

URL: http://www.dissertation.de

Zusammenfassung

Globalisierung und steigender Wettbewerbsdruck verlangen von Unternehmen eine rasche Anpassung ihrer existierenden Geschäftsprozesse einschließlich der damit verbundenen Notwendigkeit, die meist zentralisierten Prozesse ganz oder teilweise auszulagern bzw. verteilt auszuführen. Wurde ein solches Vorgehen in den letzten Jahren noch durch die mangelhafte Infrastruktur behindert, so lassen steigende Bandbreiten und sinkende Kommunikationskosten diese Hindernisse zunehmend verschwinden.

Eine wesentliche Voraussetzung für die Auslagerung ist die Aufteilung der Geschäftsprozesse in klar abgegrenzte Fragmente. Dies geschieht gegenwärtig weitgehend manuell und zur Entwicklungszeit. Zwar ließe sich einerseits das Konzept der Sub-Prozesse zur Auslagerung anwenden, andererseits gibt es jedoch keine Möglichkeit zu ermitteln, welche Teile eines Prozesses extrahiert werden müssen. Darüber hinaus ist mit dem derzeitigen Stand der Technologien nur eine statische und keine dynamische Fragmentierung möglich. Aus diesem Grunde gibt es steigende Nachfrage nach flexibleren Möglichkeiten, Geschäftsprozesse zu fragmentieren, zu verteilen und derartig miteinander zu verknüpfen, dass ihre verteilte Ausführung das Verhalten des Originalprozesses nachbildet. Um derartigen Anforderungen gerecht zu werden, spielt das Prinzip der Service-orientierten Architekturen eine wesentliche Rolle.

Die Service-orientierte Architektur (SOA) ist ein relativ neuer Ansatz um dynamische, lose gekoppelte und verteilte Systeme in einer heterogenen Umgebung zu entwickeln, wobei die einzelnen Teile der Anwendung als Service über das Netzwerk zur Verfügung gestellt werden. Web Services sind eine Möglichkeit, um eine SOA zu realisieren. Sie basieren auf einem modularen Schichtenmodell von XML Standards und deren Implementierung, welche die verschiedenen Teilaspekte der Service-orientierten Architektur umsetzen. Hierbei ist die Business Process Execution Language for Web Services (kurz „BPEL“) der geeignete Standard zur Modellierung Web-Service-basierter Geschäftsprozesse. Besonders hervorzuheben ist das Konzept des Scope, welches die Zusammenfassung einer Anzahl von Aktivitäten ermöglicht und für sie ein gemeinsames Verhalten der Fehlerbehandlung und Kompensation definiert. Darüber hinaus kombiniert BPEL die Graph- und Kalkül-basierten Ansätze der Prozessmodellierung.

Diese Dissertation beschreibt, wie Fragmente in Geschäftsprozessen identifiziert, erstellt und ausgeführt werden können, ohne die operationale Semantik des Originalprozesses zu verlieren. Dies geschieht im Rahmen der Web Service Technologien, insbesondere unter Verwendung von BPEL.

Der wissenschaftliche Beitrag dieser Arbeit sind eine Kategorisierung verschiedener Techniken zur Aggregation von Web Services, ein Metamodel für Web-Service-basierte Geschäftsprozesse auf Grundlage eines graph-basierten Formalismus, eine Methode zur automatisierten Dekomposition solcher Geschäftsprozesse unter Erhaltung der operationalen Semantik, sowie eine Architektur und Implementierung einer Entwicklungs- und Laufzeitumgebung. Es folgt eine Zusammenfassung der einzelnen Kapitel.

Grundlagen und Verwandte Arbeiten
Es gibt eine große Vielzahl von Ansätzen und Mechanismen zur Aggregation von Web Services, zumeist mit dem Ziel, Geschäftsprozesse umzusetzen. Ein erster Beitrag dieser

Arbeit ist eine Kategorisierung dieser Ansätze, wobei diese im Wesentlichen in zwei breite Kategorien unterteilt werden: bedingte and unbedingte Aggregation. Für jede dieser Kategorien werden zusätzliche Subkategorien eingeführt, die jeweils mit Beispielen und charakterisierenden Eigenschaften untermauert werden.

Eine Bestandsaufnahme der verwandten Arbeiten in diesem Forschungsfeld zeigt, dass sich bisherige Ansätze im Bereich Web-Service-basierter Geschäftsprozesse hauptsächlich auf die Interaktionen zwischen den Partnern spezialisiert haben, jedoch nicht auf die Zuteilung verschiedener Aktivitäten an unterschiedliche Partner. Darüber hinaus wird aufgezeigt, dass existierende Ansätze zur Prozessfragmentierung eine oder mehrere wesentliche Unterschiede zu dieser Arbeit aufweisen: (1) Es können keine strukturierten Aktivitäten (Scopes und Schleifen) verteilt werden bzw. die Fehlerbehandlung und das Kompensationsverhalten werden nicht berücksichtigt. (2) Sie basieren nicht auf Standards. (3) Sie benötigen eine spezielle Middleware, die jeder Partner einsetzen muss. (4) Konstrukte der Modellierungsebene werden durch systemspezifische Laufzeitkonstrukte abgebildet. (5) Die Fragmentierung basiert auf einer Optimierung von Laufzeiteigenschaften (z.B. Latenz) und nicht auf geschäftsspezifischen Anforderungen.

Meta-Model für Geschäftsprozesse
Diese Arbeit definiert einen graph-basierten Formalismus zur Modellierung von Geschäftsprozessen, welcher insbesondere die Konzepte der Verwendung von Scopes, die mehrstufige Behandlung von Fehlern und die Spezifikation der Reihenfolge von Kompensationsschritten berücksichtigt. Auf diese Weise ermöglicht der Formalismus eine natürliche Abbildung sowohl graphartiger als auch struktureller Elemente realer Modellierungssprachen, wie zum Beispiel BPEL. Diese Arbeit definiert sowohl eine Laufzeitsemantik des entwickelten Formalismus' als auch eine Überführung von BPEL-Prozessen in diesen.

BPEL-D
Diese Arbeit stellt mit BPEL-D eine Erweiterung der Modellierungssprache BPEL vor. BPEL-D ist eine Variante von BPEL, deren Kontrollfluss sich nicht vom Original unterscheidet, deren Datenfluss jedoch - im Gegensatz zur impliziten Modellierung in BPEL - durch explizite Datenflusskanten abgebildet wird. Jede Aktivität in BPEL-D besitzt Container für die Eingabe- und Ausgabedaten. Datenflusskanten verbinden Aktivitäten. Jede Datenflusskante besitzt eine Abbildungsvorschrift, die angibt, welche Container der Quellaktivität in welche Container der Zielaktivität nach der Beendigung der Quellaktivität kopiert werden sollen. Insbesondere betrachtet die Arbeit Datenflusskanten, die Daten zwischen Schleifeniterationen transportieren und die die Grenzen von Blöcken zur Fehlerbehandlung und Kompensation überschreiten.

Fragmentierung der Geschäftsprozesse
Der erste Schritt der Fragmentierung ist die Identifizierung der einzelnen Teile. Der präsentierte Ansatz legt großen Wert auf die Möglichkeit der einfachen Veränderung einer gewählten Aufteilung. Dies erreichen wir durch die Spezifikation der Aufteilung ohne Berücksichtigung der Prozessstruktur, d.h. die Fragmentierung eines Prozesse wird durch die Partitionierung der Menge seiner Aktivitäten festgelegt. Mit anderen Worten: Jede Aktivität wird dem Verantwortungsbereich genau eines Partners zugeordnet.

Das eigentliche Problem der Fragmentierung ist die Berücksichtigung von Abhängigkeiten, welche entweder explizit spezifiziert oder implizit gegeben sind. Explizite Abhängigkeiten

resultieren aus Kontrollflusskanten und BPEL-D Datenflusskanten. Implizite Abhängigkeiten hingegen resultieren aus strukturellen Elementen wie Schleifen und Scopes, sowie dem impliziten Datenfluss in BPEL, d.h. dem Lesen und Schreiben gemeinsamer Variablen. Diese Arbeit demonstriert, wie sich diese Abhängigkeiten mit Hilfe von standardisierten Web-Service-Technologien in ein verteiltes Modell übertragen lassen. Im Ergebnis entstehen durch die Fragmentierung eines Prozesses eine Menge von standardkonformen BPEL-Prozessen zusammen mit der Spezifikation ihrer Interaktion. Diese Fragmente lassen sich in jeder konventionellen Laufzeitumgebung für BPEL ausführen, vorausgesetzt Schleifen und Scopes werden nicht auseinander gerissen. Darüber hinaus ist die Fragmentierung nachvollziehbar, die Veränderungen folgen wohl-definierten Mustern und bewegen sich auf demselben Abstraktionsniveau, wie der ursprüngliche Prozess.

Die Aufteilung expliziter Abhängigkeiten geschieht im Wesentlichen durch explizite Kommunikation zwischen den Partnern: Aus einer Kontrollflusskante wird eine sendende Aktivität bei dem einen Partner und eine empfangende Aktivität bei dem anderen, wobei der Status der Kante (positiv oder negativ) durch den Inhalt der Nachricht ausgedrückt wird. Das gleiche geschieht mit einer BPEL-D Datenflusskante, wobei lediglich die Werte der abzubildenden Parameter den Inhalt der Nachricht ausmachen.

Die Aufteilung impliziter Abhängigkeiten ist wesentlich komplexer. Der vorgeschlagene Ansatz ist, die Ordnung der Schreibzugriffe auf eine gemeinsame Variable mit Bezug auf einen Lesezugriff zu erhalten. Für einen gegebenen BPEL-Prozess, die Partition seiner Aktivitäten und die Ergebnisse der Datenflussanalyse gilt es, die passenden BPEL-Konstrukte zum Austausch der notwendigen Nachrichten für jeden Partner zu generieren. Jeder Partner, der eine gemeinsame Variable schreibt, muss eine Nachricht an jeden diese Variable lesenden Partner senden, in welcher sowohl der Wert der Variable als auch die Information über den Erfolg des Schreibzugriffes übermittelt wird. Jeder Partner, der eine gemeinsame Variable liest, empfängt diese Nachrichten und rekonstruiert daraus den letztendlichen Wert der Variable. Dazu verwendet der Empfänger eine Graphstruktur aus Empfangs- und Zuweisungsaktivitäten, welche die Kontrollstrukturabhängigkeiten der schreibenden Partner nachbildet.

Protokolle der Koordinierung zwischen Fragmenten
Das Problem der Aufteilung impliziter Kontrollflussabhängigkeiten, die aus Schleifen und Scopes resultieren, wurde auf der Modellierungsebene durch die Erweiterung von BPEL um drei neue Attribute und auf der Ausführungsebene durch die Definition zweier neuer Koordinierungsprotokolle gelöst. Hierbei erwies sich WS-Coordination als geeigneter Rahmen der Koordinierung und somit wurde die Logik zur Koordinierung für fragmentierte Schleifen und Scopes in zwei neuen Protokollen abgebildet, die in das WS-Coordination Framework eingebettet werden können.

Die vorliegende Arbeit ermöglicht das beliebige Verteilen von Schleifen und Scopes, wobei die Behandlung von Scopes sowohl die Fehlerbehandlung als auch die Kompensation berücksichtigt. Der vorliegende Ansatz ermöglicht also nicht nur die Aufteilung des Inhalts eines Scopes sondern auch die seiner Komponenten zur Fehlerbehandlung und Kompensation. Der Koordinator verwendet die neu entwickelten Protokolle zur Koordinierung der Fragmente von Schleifen und Scopes derart, dass diese sich wie eine logische Einheit verhalten: Fehler werden in der Hierarchie der Scopes weitergeleitet bis die passende Komponente zur Fehlerbehandlung gefunden ist, wobei die jeweiligen Aktivitäten der betroffenen Scopes abgebrochen werden. Des Weiteren werden vollständig abgearbeitete

Scopes kompensiert, jeweils eine Hierarchieebene nach der anderens und in der umgekehrten Reihenfolge der ursprünglichen Kontrollflussabhängigkeit. Zur Sicherstellung dieses Verhaltens benötigt der Koordinator a priori Informationen über die Hierarchie der Scopes, die Fragmentierung und die Reihenfolge der Standardkompensation verteilter Scopes. Im Ergebnis können Prozessfragmente in den verteilten Laufzeitumgebungen der Partner ausgeführt werden, wobei das resultierende Verhalten dem des lokal ausgeführten Originalprozesses entspricht.

Architektur und Implementierung
Die vorliegende Arbeit beschreibt die Architektur und Implementierung eines vollständigen Systems zur Fragmentierung von Geschäftsprozessen sowohl in der Entwicklung als auch in der Ausführung. Das System besteht aus einer Komponente zur Modellierung von Geschäftsprozessen einschließlich der Spezifikation der Aufteilung und der Überführungslogik und einer Laufzeitumgebung, die die Ausführung von Prozessfragmenten ermöglicht. Die Modellierungskomponente erweitert den Open-Source-BPEL-Editor von Eclipse und generiert neben den Prozessfragmenten alle notwendigen Artefakte. Die Laufzeitumgebung ist eine Erweiterung der Open-Source-BPEL-Runtime von ActiveBPEL auf Basis des modularen Konzepts für die Einbindung von Erweiterungen. Darüber hinaus erweitert die Laufzeitumgebung die Implementierung von WS-Coordination insbesondere um die Unterstützung der neu entwickelten Protokolle zur Koordinierung verteilter Schleifen und Scopes.

Die Arbeit schließt mit einer Zusammenfassung der erzielten Ergebnisse und dem Ausblick auf verschiedene Bereiche für weitere Forschungen.

Abstract

Globalization and the increase of competitive pressures created the need for agility in business processes, including the ability to outsource, offshore, or otherwise distribute its once-centralized business processes or parts thereof. While hampered thus far by limited infrastructure capabilities, the increase in bandwidth and connectivity and decrease in communication cost have removed these limits.

An organization that aims for such fragmentation of its business processes needs to be able to separate the process into different parts. Today, this is a manual, design-time endeavor. For example, it may use the concept of subprocesses as parts to be outsourced. However, there is often no way to foresee, in advance, which parts of the process need to be cut-off. Thus, today's technology for outsourcing is static and not dynamic at all.

Therefore, there is a growing need for the ability to fragment one's business processes in an agile manner, and be able to distribute and wire these fragments so that their combined execution recreates the function of the original process. Additionally, this needs to be done in a networked environment, which is where 'Service Oriented Architecture' plays a vital role.

'Service Oriented Architecture' (SOA) is a relatively new approach to software that natively deals with the very dynamic, distributed, loosely coupled, and heterogeneous features of today's networked environment, offering application functions as networked services. Web services is one instantiation of an SOA, consisting of a modular, layered stack of XML standards and corresponding implementations that address the different aspects of this environment. The standard covering business processes for Web services is the Business Process Execution Language for Web Services (also known as 'BPEL'). Relevant characteristics of BPEL are that it is SOA-centric, has a scope construct that groups activities providing them with common behavior such as fault and compensation handlers, and combines graph and calculus based approaches to process modeling.

This thesis describes how to identify, create, and execute process fragments without loosing the operational semantics of the original process models. It does so within the framework of the Web services stack of standards, BPEL in particular.

The contributions are a categorization of existing Web services aggregation techniques, a meta-model of Web services business process mechanisms using a graph-based formalism, a solution for the automatic and operational semantics-preserving decomposition of such processes, and an architecture and implementation for a corresponding build-time and runtime environment. A summary of this thesis is presented, following the organization of its chapters.

A summary of each chapter follows.

Background and Related Work

A large number of mechanisms and approaches have been created for aggregating Web services, usually with business processes at the forefront. The first contribution of this thesis is to categorize these approaches. We find that they fit into two broad categories: constrained and unconstrained aggregation. Within each, additional subcategories are identified. Examples and characterizing features of each of the categories are presented.

A survey of the related work in this space shows that existing Web services-based inter-organizational workflow efforts have focused on the interactions between partners, not on assigning activities to different partners. Additionally, we show that existing process splitting approaches exhibit one or more of the following characteristics that distinguish them from our work: (1) not handling splitting structured activities (scopes, loops) or propagation of faults and compensation behavior, (2) not being standards-based, (3) requiring every participant to run dedicated, specialized middleware, (4) mapping process model artifacts to lower level runtime artifacts, (5) performing the split based on optimizing runtime properties (i.e.: latency), not for a business need.

A Business Process Meta-Model
The thesis presents a graph-based formalism for business processes that includes the use of scopes, the propagation of fault handling, and the design-time computation of compensation order. This support makes the new formalism able to naturally support mappings from languages like BPEL that combine graph and calculus based approaches to process modeling. We define the run-time navigational semantics of this model and present a mapping from BPEL processes to this new formalism.

BPEL-D
The thesis introduces new extensions to BPEL called 'BPEL-D'. BPEL-D is a variant of BPEL having the same control semantics but substituting BPEL's implicit data flow with explicit data flow through the use of data links. In BPEL-D, activities have containers into and out of which data flows. Data links connect activities, with maps specifying which containers of the source activity should be copied into which containers of the target activity upon the source activity's completion. It also presents the use of data links between loop iterations and crossing boundaries of a scope's fault or compensation handlers.

Process Fragmentation
The first step in fragmenting a process is defining the different parts to be split off. Our solution places strong emphasis on enabling easily changing the fragmentation of a process. This is achieved by separating the definition of a business process from the specification of its fragments. The fragmentation is encoded as a set-theoretic partition, i.e. by just marking a set of activities with the participant responsible for them.

The problem of fragmenting a process is actually the problem of splitting dependencies present in the process. These dependencies are either explicit or implicit. Explicit control dependencies are embodied by control links. Explicit data dependencies are embodied by BPEL-D data links. On the other hand, implicit control dependencies are embodied by scopes and loops and implicit data dependencies are embodied by implicit data flow in BPEL (read/write to shared variables). This thesis shows how each of these dependencies can be split using standard Web services technology.

The result of the fragmentation is a set of standard BPEL process models including proper communication of data and control between participants. The fragments can run on any BPEL engine provided that there are no split loops and scopes. Furthermore, the splits are transparent, i.e. it is clear where the changes are and they are done in the same modeling abstractions as the main process model.

The explicit control dependency (a control link) that is broken across participants is flowed via explicitly exchanged messages. These messages are exchanged by sending and receiving

activities that propagate the status in the message payload. The same is done, with the addition of the data itself to the message payload, for an explicit data dependency (a BPEL-D data link) that is broken across participants.

Splitting implicit data dependencies is more complex. The proposed approach for handling their split is to reproduce the order of writers to a shared variable for a particular reader of that variable. A summary is: Given a BPEL process, a partition, and the results of data analysis on that process, produce BPEL constructs in the process of each participant to exchange the necessary data. For every reader of a variable, writer(s) in different participants need to send both the data and an indicator of whether or not the writer(s) ran successfully. The participant's process that contains the reader receives this information and assembles the value of the variable. The recipient uses a graph of receive and assign activities reproducing the control dependencies between the original writers.

Protocols for Inter-Fragment Coordination

The problem of splitting the implicit control dependencies found in loops and scopes is solved, at the language level, by adding just three new attributes to BPEL. At the runtime level, it is solved by using two new coordination protocols. WS-Coordination is chosen as the coordination framework. The coordination logic for fragmented loops and scopes is thus modeled as two new protocols that plug into the WS-Coordination framework.

This thesis enables one to arbitrarily split loops and scopes. The work of a scope includes both fault handling and compensation handling. Therefore, we allow one to split not only the scope body but also its handlers. The coordinator uses the new protocols to coordinate the fragments of loops and scopes so they can behave as logical units: Faults propagate up the scope hierarchy until a matching fault handler is found, aborting nested activities. Compensation occurs on completed scopes, one level of scope nesting at a time and in an order that reverses control dependencies between peer scopes. To coordinate this behavior, the coordinator has a-priori information about the scope hierarchy, the fragments, and the default compensation order of split scopes.

As a result, process fragments can run at different participants' sites such that the resulting behavior is the same as that of the original business process running locally.

Architecture and Implementation

The thesis describes the architecture and implementation of a complete system supporting the design-time and runtime of business process fragmentation. It consists of a business process editor that also supports the specification of a partition and the transformation logic, and a runtime that supports running fragmented processes. The editor extends the open-source Eclipse BPEL editor. It produces the process fragments and any additional artifacts needed. The runtime extends the open-source ActiveBPEL engine using a modular approach for plugging in extensions. It also extends an implementation of WS-Coordination. In particular, our runtime supports the coordination protocols we have defined for split loops and scopes.

Finally, a summary of the contributions is presented and several areas for future research are identified.

Acknowledgement

There are a number of people who stand out in their critical role in having made this work possible.

Prof. Dr. Leymann for his insight, time and effort in supervising this thesis, as well as his guidance and support over the years in matters of work and life. He managed to make a nearly impossible proposition into a truly great journey. Thank you. Prof. Dr. Rothermel for his time and valuable feedback as my second advisor.

Sanjiva Weerawarana for talking me into this crazy venture and his invaluable mentoring and friendship ever since my first job as a summer intern in his group. Francisco (Paco) Curbera for insisting it was possible to take on a PhD on another continent while holding down a full-time job - and ensuring I had the means needed to do so.

The IBM Corporation, especially my line of management Paco, Sharon Adler, John Turek, and Chun Sheng Li, for supporting this effort both financially and otherwise. Thomas Mikalsen for discussions on the implementation of WS-Coordination. Dieter König, Dieter Roller and Simon Moser for valuable discussions on BPEL 2.0 - and the traditions of Swabian breweries. My team members in the Component Systems Group, in particular Florian Rosenberg and Axel and Heike Martens for help with translation.

Stefan Tai for tips on life, thesis writing, WS-Coordination, and the German system of education, while caffeinating from La Bergamotte in NYC to Café Hawelka in Vienna.

The entire IAAS team for their friendship and support with bridging the geographic gap and helping with – in addition to the exciting technical discussions - housing, regulations, translation, fun times, and moral support. In particular, Dimka Karastoyanova, Oliver Kopp, and Jussi Vanhatalo for their input on different aspects. Fabienne Marquardt for being a fabulous hostess. Matthias Wieland and my mother, Rima, for help with formatting.

My family for believing in me and for their unwavering support, regardless of the little time I had left for them. Jean-Claude Saghbini for acting as a sounding board for some ideas and supporting me throughout – in spite of years of scrapped vacations, and seven-day work-weeks.

Finally, I would like to thank the many voices over the years that I may have forgotten, the anonymous reviewers of my publications, and the workflow and SOA research communities in general.

Table of Contents

List of Figures

List of Tables

List of Symbols and Primary Functions

The symbols and primary functions used in this thesis, in order of presentation.

Symbol	Description	Page
P	A process in the meta-model	40
$\pi_i(t)$	The projection to the i-th component of a tuple *t*	40
$\wp(X)$	Power set of set X	40
N	Set of nodes, where each node is an activity	40
E	Set of links weighted by Boolean transition conditions	40
C	Set of all conditions	40
S	Set of named scopes	41
$H = (N, E_H)$	Hypergraph of scopes	41
$G = (N_G, E_G)$	Graph of the hypergraph of scopes	41
F	Set of all fault handlers	41
M	Set of fault names	41
J	Set of compensation handlers	41
W	Set of named loops	41
$\psi(a)$	Implementation of activity *a*	41
$\phi(a)$	Join condition of activity *a*	41
$E^{\leftarrow}(a)$	Set of links entering *a*	41
$E^{\rightarrow}(a)$	Set of links leaving *a*	41
A(s)	The set of activities in a scope *s*	41
$M(s)$	Subset of M containing the fault names caught by handlers of scope *s*	41
$F(s) = \{f_1, ..., f_n\}$	Set of fault handlers of scope, where $f_i = (\mu, p_f, s), \mu \subset M(s), p_f \subset N$	41
$A(f)$	The set of activities of a fault handler	41
$j(s) = (P_j, s)$	Partial function returning the compensation handler associated with a scope *s*	41
S_{comp}	The domain of *j(s)*, containing scopes having an explicit compensation handler	41
$e_H(s)$	The hyperedge corresponding to scope *s*	43
E(s)	The links whose sources are in *A(s)*	44
M_0	The set of names of built-in faults	45
$w = (c, A_l)$	A loop, where $c \in C, A_l \subseteq N$	45
$K(w)$	Maps a loop w to an activity in *P*	47
Υ	The activity and predicate state map	47
ω	The activity state map	47
λ	The completed activities map	48
δ	The disabled activities map	48
ρ	The faulted activities map	48
z	The waiting fault handlers map	48
N'	Process start activities	48
N''	Fault handler start activities, where $N''(f)$ are those of fault handler *f*	48
N'''	Loop start activities, where $N'''(w)$ are those of a loop w	49
ξ	The predicate state map	49
?	Unknown	49
$[\![p]\!]_t$	The truth value of a predicate *p* at time *t*	49
$[\![e]\!]_t$	the status of a link *e* at time *t*	50
$N_G(a)$	Set of all scopes containing activity *a*	51
$s'_0(a)$	The immediately enclosing scope of *a*	51
$body(s)$	Returns the activities in the body of scope s	51

$c(a, s, t)$	Returns true if a completes s at time t and false otherwise	52
$\delta E(s, t)$	The dead-link list of a scope *s* at time *t*	52
$\perp$	Not found	53
$parent(s)$	Retrieves the parent of scope *s*	53
$s_0(a)$	The scope where fault handler lookup starts	54
$y(n, s, a)$	Returns a matching fault handler for fault *n* thrown by a starting from scope *s*	54
$fh(a, n)$	Returns a fault handler for a fault named *n* thrown by *a*	54
$p\_faulted(a, t)$	Returns whether *a* threw a primary fault at time *t*	55
$A_{disable}$	The set of activities to disable in a scope	56
$wait\_fh(a, t)$	Returns true if *a* primary faulted at *t* but the fault handler must wait	56
$end\_wait(a, t)$	Returns true if *a* completes/faults at *t* allowing a waiting fault handler to run	56
$V(a')$	Map from a BPEL activity *a'* into parts of *P*	67
$A_{V(a')}$	Set of activities in *P* from mapping BPEL activity *a'*	67
$E_{V(a')}$	Set of edges in *P* from mapping BPEL activity *a'*	67
$\alpha V(a')$	Start activity created when mapping BPEL activity *a'* to parts of *P*	68
$\beta V(a')$	End activity created when mapping BPEL activity *a'* to parts of *P*	68
L	Map from BPEL links to *P* links	68
COG	Compensation order graph in *P*	77
ScopeLoopQ	Queue of scope and loop instances, contains *ChildQ*s for child instances	80
P_a	Process partition	93
$L_c(n)$	Map from a split loop name *n* with the loop's responsible participant name	94
RT	Scope and loop relationship tree	98
S_{rt}	The set of split scope-nodes in *RT*	98
S_{ns}	The set of non-split compensation-relevant scopes in *RT*	98
L_{rt}	The set of split loop-nodes in *RT*	99
F_{rt}	The set of fragment-nodes in *RT*	99
A_{rt}	The set of fragment-activity edges in *RT*	99
B_{rt}	The set of loop-scope edges in *RT*	99
C_{rt}	The set of child-parent edges.	99
$[\![p]\!]$	Boolean result of evaluating a predicate *p*	102
$f(x)\downarrow$	Denotes that a partial function *f(x)* is defined for the value of *x*	103
DCO	Default compensation order graph of a BPEL scope	105
$Q_s(a, x)$	Set of query and writer set tuples resulting from data analysis on a BPEL process.	119
$Q_s^{postloop}(l, x)$	Set of query and writer set tuples for data needed after a loop.	120
$Q_s^{preloop}(l, x)$	Set of query and writer set tuples for data needed before a loop	120
$Q_s^{intraloop}(l, x)$	Set of query and writer set tuples for data needed from a previous loop iteration	120
$Q_s^{prehandler}(h, x)$	Set of query and writer set tuples for data needed for a compensation handler	120
$A_d(a, x)$	The set of all writers in $Q_s(a, x)$	120
fh_{rt}	Returns the matching fault handler from *RT* for a fault named *n*	147

Chapter 1 Introduction

The Internet has networked the world's enterprises with a speed and ease that have enabled major changes in the way they communicate and, more importantly, do business. This networked environment natively exhibits several key features that make the way to do business drastically different from the past, where enterprises operated as isolated silos: It is very dynamic, distributed, loosely coupled, and heterogeneous. Software supporting business interactions between enterprises in this new environment is often based on business process models. This thesis is focused on solving some of the core challenges resulting from the need to dynamically restructure enterprise interactions. Restructuring such interactions corresponds to the fragmentation of intra and inter enterprise business process models. This thesis describes how to identify, create, and execute process fragments without loosing the operational semantics of the original process models.

'Service Oriented Architecture' (SOA) is a relatively new approach to software that natively deals with the key features of the above-mentioned networked environment, offering application functions as networked services. Web services [WCLS05] is one instantiation of an SOA, consisting of a modular, layered stack of XML standards and corresponding implementations that address the different aspects of this environment. These aspects include, among others, service descriptions, means of communication, service composition, and quality of service capabilities. Offering application functions as networked 'services' enables anyone supporting the stack to interact with and consume these services. The set of Web services standards promote interoperability and portability, resulting in platform independence: Neither a Web services user nor a Web services provider is restricted to using a particular implementation language, middleware platform, or hosting environment.

In the SOA world, the core task of creating new applications, especially those involving interactions between enterprises, has become the complex, yet equally vital, task of aggregating services into business processes. However, it requires relinquishing control of the parts, i.e. the services, making up the application [FRWK02] [KHMW03]. The building block in the Web services world is, simply, the service: services can achieve their function directly and/or by aggregating functions provided by other services - possibly remote ones owned by different organizations. In Web services, the area of composition is strongly influenced by work on Workflow and Business Process Modeling. *Please note that in this thesis we use the terms 'workflow' and 'business process' interchangeably as often done in the literature; the same is true for the use of 'process' and 'process model' when it is clear from the context whether the model instance or the model itself is meant.*

The standard covering this area is the Business Process Execution Language for Web Services [AAAB07], referred to either as BPEL4WS, WS-BPEL, or simply BPEL. It enables the creation of business processes by weaving together Web services interactions. Weaving interactions in BPEL consists of providing control flow around a set of activities. Activities may themselves provide operations of new services or presented in SOA-based composition, and embodied in BPEL.

Globalization and the increase of competitive pressures created the need for agility in business processes, i.e. to dynamically reassign different groups of activities to different business partners. While hampered thus far by limited infrastructure capabilities, the increase in bandwidth and connectivity and decrease in communication cost have removed these limits. Thomas Friedman [FRIE05] put it aptly, stating that "the world is flat." As the technology to manage and collaborate with people across geographical boundaries has matured, corporations have spread their boundaries across continents. Outsourcing and off shoring are hot topics for enterprises today, enabling them focus on core competencies. For a software company, for example, non-core parts of the business could include call centers, Customer Relationship Management (CRM), and/or accounting. Third party companies have been created whose sole purpose is to support these trends.

Even aside from companies globalizing their own operations, some are making revenue by selling business process models. The processes are meant to be enacted by independent, distributed partners. The creating company is not even involved in their execution. One example is Oracle and its creation of Process Integration Packs[1]. Another is RosettaNet[2], a consortium of companies from the Electronics industry that defines an open e-business environment using an open–standards approach for the requirements and behavior of business partners.

In order for an organization to be able to outsource, offshore, or otherwise distribute its once-centralized business processes or parts thereof, it needs to be able to separate these parts. Today, this is a manual, design-time endeavor. For example, it may use the concept of subprocesses as parts to be outsourced. However, there is often no way to foresee, in advance, which parts of the process need to be cut-off. Thus, today's technology for outsourcing is static and not dynamic at all.

Therefore, there is a growing need for the ability to fragment one's business processes in an agile manner, and be able to distribute and wire these fragments so that their combined execution recreates the function of the original process. Process modeling languages currently are either focused on the work of one party such as BPEL, or on the global work done between a set of parties such as WS-CDL [KABR04]. BPEL4Chor [DKLW07] has been newly introduced to handle a mix of the two. However, none deals directly with fragmenting processes.

In this thesis, we enable the fragmentation of business processes in a Web services environment. The contributions are a categorization of existing Web services aggregation techniques, a meta-model of Web services business process mechanisms using a graph-based formalism, a solution for the automatic and operational semantics-preserving fragmentation of such processes, and an architecture and implementation for a corresponding build-time and runtime environment.

1.1 Overview

To support understanding of process behavior, we provide a process meta-model that combines calculus with graph-based process modeling. This combination is one of the aspects of BPEL: It is a mixed language containing both structured activities and cross-

[1] http://www.oracle.com/applications/process-integration-packs.html
[2] http://www.rosettanet.org

structural control links. For graph-based modeling, a meta-model of a workflow consisting of data and control flow surrounding a set of activities is presented in [LERO00]. In [CKLW03], we claim that BPEL's key enabler for combining these approaches is its exception handling mechanism using 'scopes'. A scope is a construct that groups a set of activities together that share a common context, similar to [LERO00]'s concept of 'spheres'.

Several approaches have mapped BPEL into lower level formalisms that enable running analysis algorithms. However, these formalisms are based on artifacts whose behavior is largely different from the primitives of BPEL. While this is not a big concern when performing advanced analysis, we aim to model the process language's primitives in a first class manner so we can understand their execution behavior, enable modeling in the formalism directly if so wished, and be able to perform the same behavior in a distributed setting.

The first part of this thesis proposes a formalism for business processes that extends [LERO00] to include the use of scopes, the propagation of fault handling, and the design-time computation of compensation order. This support makes the new formalism able to naturally support mappings from mixed languages like BPEL. The approach consists of using an acyclic graph with conditional control links as edges and activities as nodes. Hypergraphs are used to model scopes, each scope being a hyperedge consisting of activities from the main process graph. Each scope provides an optional set of fault and/or compensation handlers. Scopes may be nested, with the top-most scope containing all activities in the process. The nesting relation between scopes is modeled using a tree structure, with the nodes being the hyperedges of the hypergraph. We define the run-time navigational semantics of this model. Finally, we present a mapping from BPEL processes to this new formalism.

In the second part of this thesis, we focus on the key goal of enabling organizations to fragment business processes. Our solution places strong emphasis on enabling easily changing the fragmentation of a process. This is achieved by separating the definition of a business process from the specification of its fragments. The fragmentation is encoded as a set-theoretic partition, i.e. an assignment of disjoint sets of activities to different participants. In other words, a partition is created by just marking a set of activities with the participant responsible for them.

The fragmentation solution presented consists of an algorithm that performs the split using the business process and its partition. The result is several BPEL processes, one for each participant, as well as the information needed to connect the participants resulting in an overall business process execution having the same semantics as the original process. To achieve fragmentation in the presence of split loops and scopes, we define two new corresponding coordination protocols.

Instead of taking a participant-centric approach which is dependent on a-priori knowledge of the partition, a process-centric approach is suggested that is independent of particular partitions. Subsequently, one can derive an 'abstract process' for each party that encodes only the resulting interactions. Once the local processes have been created, they need to be wired together so that the distributed work reproduces the behavior of the main process model.

In creating the fragmentation design, three main criteria were deemed essential and drove several design decisions: (i) Maintaining interoperability through standardization. The input process, as well as the resulting processes, are specified in standard BPEL. In the case of split

loops/scopes standard-compliant extensions are introduced. (ii) Ensuring transparency: The fragments should be in a format that the designer can understand, the system can audit, and tools can represent graphically. It should be possible to identify the artifacts added to connect the fragments from those that are business-critical. (iii) Not requiring (proprietary) middleware: Using patterns of BPEL itself to recreate the behavior in the unsplit BPEL process enabled us to avoid the use of additional proprietary middleware. In so far as one does not split loops and scopes, any BPEL engine can be used to run the fragments created by our approach. Upon splitting loops and scopes, however, we require the use of existing middleware implementing standards-based agreement protocols, i.e. WS-Coordination [OASI07].

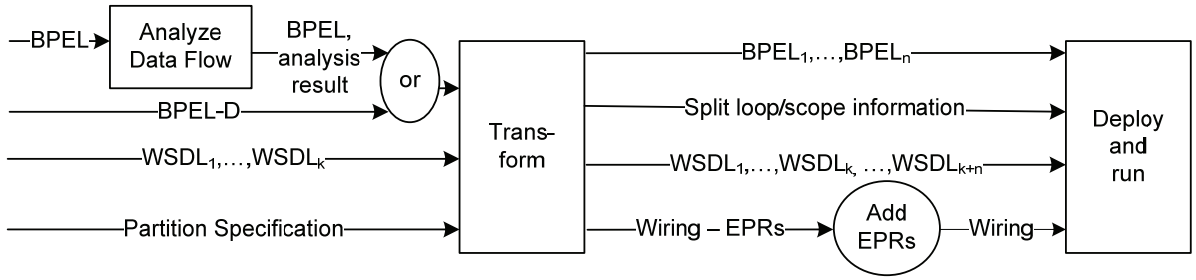


Figure 1: Overview of the approach for splitting processes

Architecturally, the approach presented follows the steps shown in Figure 1. The transformation ('Transform' box) takes as input either a BPEL process or a variant of BPEL we have created that uses explicit data flow ('BPEL-D', see Chapter 4), its corresponding WSDLs, and a specification of the partition. The result is the creation of one BPEL process and one WSDL file per participant, as well as a simple global wiring definition and information regarding split loops/scopes. To enable splitting loops and scopes, coordination protocols are introduced and used at runtime ('Deploy and run' box) to ensure that these tightly coupled constructs can in fact run in a fragmented fashion.

As a first step towards computing fragmentation, we focus on data and control logic. We provide algorithms that create the participant BPEL definitions. Maintaining data dependencies across fragments in the case of BPEL, as opposed to BPEL-D, presents special challenges for which we use patterns of receiving and assigning activities in a flow to resolve race conditions across participants. A pre-requisite for this is to perform data analysis ('Analyze data' box) on BPEL. Several data analysis for BPEL already exist, with one created specifically for the problem in this thesis [KOKL07]. We do not repeat the data analysis in this thesis, as any of these may be used as long as their result can be put into the input format, described in section 5.10.1, of the 'Transform' box.

From there, we move to the more advanced case of breaking apart loops as well as fault handling and compensating scopes. This is a case of agreement between partners: A fragment of a scope cannot complete successfully until the other fragments of that scope have also completed successfully. A fragment of a loop cannot iterate or continue unless it is aware of the loop condition and that its fragments are also ready to iterate, etc. Therefore, we cast this problem as one of 'coordination'. We provide coordination protocols, pluggable into WS-Coordination compliant implementations, which can ensure that the fragments work in-step with each other for these advanced cases. The process meta-model defined in Chapter 3, especially the behavior of faults, loops, and compensation, becomes critical at this stage. It provides us with the steps carried out to perform these advanced functions.

The final part of the thesis describes the architecture and implementation of a complete system supporting the design-time and runtime of business process fragmentation. It consists of a BPEL-D editor that also supports the specification of a partition and the transformation logic, and a runtime that supports running split processes. The editor produces the process fragments and any additional artifacts needed. The runtime extends the open-source ActiveBPEL engine [ACEN07] using a modular approach for plugging in extensions presented in [KHKL07A]. In particular, our runtime supports the coordination protocols we have defined for split loops and scopes.

1.2 Organization

The thesis is organized as follows. Chapter 2 presents the background and related work, including a survey of aggregation and composition techniques and continuing into process modeling, fragmentation, and coordination. Chapter 3 presents a graph-based formalism for a business process meta-model combining graph and calculus approaches, and shows how it can be model the constructs of the BPEL language.

Chapter 4 presents BPEL-D, a variant of BPEL with data-links. The groundwork for fragmenting BPEL processes is in Chapter 5: the notation for designating a partition, the algorithm for creating the individual processes by breaking data and control links, and a notation to encode the information needed to connect the fragments. Splitting loops and scopes requires inter-process coordination, addressed in Chapter 6, including two new coordination protocols. Chapter 7 describes the architecture of the implemented system that provides design and run time support for process fragmentation. Finally, we conclude in Chapter 8 with a summary and a description of open research areas.

Chapter 2 Background and Related Work

We provide an overview of salient technologies used in this thesis in section 2.1. Having provided the basic background information, the context in which this work was done is presented in section 2.2. It lays out the state of the art, and explains the key ideas of existing work, as well as the differences and similarities to the contributions of this thesis. This starts with a classification of aggregation techniques, including business processes, used in the Web services technology. Then, we focus on areas of business process technology that are most relevant to the contributions of this thesis: distributed/inter-organizational business processes, splitting business processes, and the use of coordination in concert with business process systems and models. For an overview of business process techniques and models in general, we point the reader to [DUHA05].

2.1 Background

This section provides a brief overview of relevant technologies, mainly related to Web services, needed as background for understanding the contributions of this thesis. Additional details, where needed, are provided in the chapters that deal with a particular subject matter.

2.1.1 The Web Services Framework

The Web services framework [WCLS05] presents an open, XML-based realization of the Service Oriented Architecture paradigm. It provides a modular, layered stack of standards that enables interoperability and portability of distributed applications, especially in the Internet. These specifications cover the areas of data encoding (XML), message structure (SOAP [W3CS00]), endpoint references (WS-Addressing [W3CW04]), service description (WSDL [W3CW02A]), composition (BPEL [AAAB07]) and quality of service such as reliability, security, and coordination (WS-Coordination [OASI07]). In addition, it provides the WS-Policy framework [HOKA03A] [HOKA03B] for declaratively attaching (QoS) policies to Web service artifacts. In this thesis, we focus most heavily on four of these standards: WSDL, WS-Addressing, BPEL, and WS-Coordination.

WSDL and WS-Addressing are the simplest of these standards. The Web Services Description Language, WSDL, provides a two-part description of a Web service: abstract and concrete. The abstract part consists of portTypes that group operations that in turn have input and output messages typed using XML-Schema. A portType is effectively the interface of the service. The concrete part consists of binding the interface to an explicit transport and data format, such as SOAP over HTTP, and of providing one or more endpoints at which an implementation of the service is made available for external callers. WS-Addressing provides a transport neutral representation of Web service endpoints, as well as additional ‘message information’ headers. One example of such a header is ‘replyTo’, that provides an endpoint where the asynchronous reply to a message should be sent. A WS-Addressing endpoint contains a URL and an optional set of properties that enable the addressee to do further processing on the address while remaining opaque to the caller.

BPEL

The Web Services Business Process Execution Language, known as BPEL, WS-BPEL, or (formerly) BPEL4WS, is the standard for workflow-based composition in the Web services

stack. A BPEL process embodies the main characteristics of SOA based compositions, which we have laid out in [KHMW03]. They include being:

- Recursive: It invokes other services, and provides its own capabilities by exposing itself as one or more Web services.
- Peer-to-peer: The services being composed do not belong to the composition. In fact, they are possibly owned and controlled even by other enterprises altogether and may be compositions themselves.
- Robust: The composition supports error handling using both fault handling similar to that provided in programming languages and compensation [LEYM95] which is rolling back completed behaviors. It composes services at the type, not instance, level.

BPEL Processes come in two flavors, abstract and executable. A BPEL 'abstract process' provides a specification of a service's behavior and may hide private information that is not relevant to the other participants in the interaction. On the other hand, the 'executable' variant provides a process definition with enough information for it to be interpreted and, thus, executed. In this thesis, we focus on the executable variant.

A BPEL process is exposed as one or more Web services with WSDL portTypes. Named 'partnerLinks' define instances of typed connectors that provide one or both of a role that the process implements, and one that it expects from a partner. A process's logic is encoded using activities that define the business actions, or steps, and control dependencies between these activities. Primitive activities have pre–defined behavior: invoking a Web service ('invoke'), receiving and replying to invocations of their exposed operations ('receive/reply'), waiting ('wait'), throwing faults ('throw'), and so on. Structured activities, on the other hand, impose behavior on groups of activities nested within them, such as strict sequencing ('sequence'), or parallelism ('flow'). Conditional control links may impose additional ordering on activities within a 'flow'. The language also provides conditional, directed control links that define 'transitionConditions'. If the 'transitionCondition' of a link from activity A to activity B is true, then A must complete before B can start. An activity also has a 'joinCondition', whose default is the disjunction of the status of its incoming links, which determines when it can run. Common phrases used throughout this thesis when regarding links are: when a link is traversed and its the transition condition is set, we say that the link has 'fired' with that value. We also use the term 'a link has a value true/false' or the 'status of a link is true/false' to mean that the link's transition condition has that value. More will be described about these constructs in Chapter 3, in which we provide a meta-model for the language and for the execution semantics of its elements.

'Scopes' are special structured BPEL activities that provide additional capabilities to their nested activities, such as data scoping, fault handling, compensation handling, and event handling. Compensation handling enables one to provide undo actions for completed units of work, with a default order being the reverse order of the work units themselves. It was introduced for workflow by Leymann in [LEYM95] and is further elaborated on by Eder and Liebhart in [EDLI96]. Fault handling on nested constructs in workflow systems was used in OPERA [HAAL98], in combination with transactional capabilities using a notion of 'spheres of atomicity'. Scopes will be of special interest in Chapter 3 and Chapter 6, which go into more details. The former describes their behavior and defines their execution semantics in the meta-model, while the latter uses them to define how to split a scope while maintaining its operational behavior.

BPEL makes use of the 'Dead Path Elimination' technique (DPE) [LEAL94] to automatically skip or disable activities that are along a path that is determined to be no longer reachable (dead path). For example, if an upstream activity has failed, DPE propagates the disabled state across the process graph to all activities waiting on the disabled one, thereby ensuring process termination. DPE is an accepted capability in graph-oriented workflow languages such as FDL [LERO00] and WSFL [LEYM01]. In BPEL, DPE is simply a special case of BPEL's fault handling: An activity whose 'joinCondition' evaluates to false throws a 'joinFailure'. An optional activity or process level Boolean-valued attribute, 'suppressJoinFailure', may be set to true to provide behavior equivalent to surrounding the activity with a scope having an empty fault handler for joinFailures and forcing the system to perform DPE. Process-wide DPE semantics is achieved simply by setting this attribute to true on the whole process. [CKLW03] provides overall coverage of BPEL fault handling, with a focus on advanced issues on BPEL's joinFailure in particular.

In order to support multiple entry points, a BPEL process provides a first-class correlation mechanism to help route incoming messages to the correct process instance. A correlation set enables one to refer to specific parts of different messages aliased to named properties. The interaction activities of a process can set their correlation values when a message is sent or received. Then, a message from a partner can be checked for these values and matched to the proper running instance. We will make use of correlation in order to ensure that different instances of process fragments corresponding to different instances of the original (unsplit) process do not mistakenly interact together.

Process data is stored in shared, scoped variables. Variables may be read and/or written by any activity in the process that has the ability to read/write data. Therefore, data flow through the process is implicit as opposed to other workflow languages in which it is modeled using explicit data links. Chapter 4 defines a variant of BPEL, BPEL-D, that replaces implicit data flow with explicit data flow. The splitting algorithm in this thesis will first addresses BPEL-D, then go on to reuse and expand the concepts used therein to handle splitting standard BPEL (i.e. with implicit data flow instead of explicit data links).

This thesis focuses on the 1.1 version of BPEL (BPEL 1.1). BPEL 2.0 was released recently, providing some changes. Many of the changes were additive to BPEL 1.1 and did not change the overall navigation approach itself (additional activity types, renaming existing activities, etc.). Other changes included corrections and clarifications in particular for compensation handling and data manipulation. In this thesis, we do use the compensation model in BPEL 2.0 for compensation handling due to the provided corrections. This affects a controlled part of the algorithms in Chapter 6.

The core contributions of BPEL to previous workflow and Web-based composition are described in [KHMW03]. Additionally, our survey in [KHLK06] provides a look at BPEL's core principles and diverse BPEL applications and extensions. For details on the architectures of BPEL engines, see IBM's WebSphere Process Server [KKLP04], ActiveEndpoint's ActiveBPEL [ACEN07], as well as our work on the first public BPEL implementation, BPWS4J [CKNW06] [CDKM04] and usage in the lightweight Colombo server [CDKN05].

WS-Coordination

WS-Coordination [OASI07] is a pluggable framework for coordinating the agreement of the outcome of the execution of a collection of services that jointly perform a 'distributed action'. For that purpose, WS-Coordination specifies two middleware-related services: an Activation

Service and a Registration Service. These services are used to set up the connections between coordinators and participants and to join participants to a coordinated unit of work. Being a framework, WS-Coordination allows one to define one's own agreement protocols. An implementation of new protocols would also need corresponding protocol handlers to perform the actual coordination logic. A protocol handler is the entity in charge of exchanging messages to reach outcome agreement between the services

The Activation Service, which is optional, is used by the initiator to create a unique context identifying the distributed action to be started. Then, this context can be exchanged, for example, in the header of application messages between the services of the distributed action. Upon first receiving a message with a coordination context, such a service registers for participation in the distributed action with the Registration Service. Registration especially encompasses associating a 'protocol handler' with each such service. A distributed action is thus equivalent to an instance of the associated agreement protocol. Combining registration and activation allows services to dynamically join such a running protocol instance.

Specifying a new protocol consists of possibly extending the standardized activation and/or registration messages, defining the port types of both the participants' and coordinator's protocol handlers, the order in which the protocol messages need to be exchanged, and the required underlying behavior as a result of receiving or sending a protocol message. For example, if a coordinator sends a *commit* message in a transaction protocol to a participant, the participant must respond with *committed* - but only after doing the work associated with the *commit* message (i.e.: committing the transaction). Examples of existing protocols include the WS-AtomicTransaction protocol [OASI07A] for an atomic activity, the WS-BusinessActivity protocol [OASI07B] for a long-running activity, and the WS-HumanTask protocol in BPEL4People [AABI07] that ties together tasks and people activities.

The framework also provides some advanced features not used in this thesis. These include creating subordinate coordinators and allowing each application to use its own coordinator. For an overview of the use of coordination in Web services, see [LITT03] and [CUKM08].

In this thesis, we propose two new WS-Coordination protocols for running split loops and split scopes. The details of how they fit into the framework and relate to existing protocols are presented in Chapter 6.

2.2 Related Work

Having covered the necessary background, we now move on to the related work. A lot of research has been undertaken in this area, requiring us to focus on the topics most closely related to the core ideas of this thesis.

2.2.1 A Survey and Classification of Web Services Aggregation

A large number of mechanisms and approaches have been created for aggregating Web services, usually with business processes at the forefront. [JOHA02] notes that agents concentrate on dynamicity while components concentrate on composability. Web services need both, making their aggregation models of particular interest. In [KHLE03], we provide a classification of Web services aggregation approaches, putting them in context with prior art in component composition techniques, and classifying them based on their usability.

The purpose of doing so is two-fold: first, it distills the large amount of work in this space enabling us to group mechanisms based on their design goals and applicability; second, it

identifies a set of primitive aggregation techniques enabling one to consider combinations and higher level compositions. This section clarifies the relationships of these approaches to each other including where they may overlap and why their focus is different, making it easier for designers to pick technologies appropriate to the task at hand.

We identify two top-level categories of aggregation: unconstrained and constrained.

- **Unconstrained aggregation:** An aggregation mechanism that *does not* impose constraints on the functionality that aggregated services must support and/or on the interactions they may have with each other or with the aggregate itself.
- **Constrained aggregation**: An aggregation mechanism that *does* impose constraints on the functionality that aggregated services must support and/or on the interactions they may have with each other or with the aggregate itself.

Unconstrained Aggregation

We identify two forms of unconstrained aggregation: grouping and recursive wiring. Grouping simply groups a set of services together, similar to putting them in the same 'bag'. The common forms of grouping include interface grouping and instance grouping.

Interface grouping has appeared in the form of interface inheritance, proposed for WSDL 2.0. At the WSDL definition level, it is simply the equivalent of copying the definitions of the supertype to the definition of the subtype. An implementation of the subtype provides a binding and an endpoint at which lies the implementation implementing all the operations, including the inherited ones. Interface inheritance allows substitutability at the instance level. An instance of the subtype in the inheritance hierarchy can by used wherever an instance of one of its supertypes may be used.

Bringing interface inheritance over from the world of Object Oriented Programming (OOP) to Web services has caused several new complications due to the change in environment and assumptions. In the latter case, the ownership of the interfaces may belong to different entities (people, organizations, etc). A subtype creator has no control over the definition of the supertype or its evolution. For example, consider a case where one would like to inherit from two different interfaces for performing a particular new function. However, the two interfaces have a method with the same signature. Normally, one could simply rename one of them but this is no longer possible because the subtype's creator does not have write access to them. A second complication comes from the use of any decorators on the interfaces. These decorators could include extensions such as those related to quality of service requirements. An operation may be declared secure in the supertype yet unsecure in the subtype. Another example is a case where a supertype uses a declarative extension for some functionality that the subtype implementation does not understand. The semantics for resolving such clashes in policies and naming are as yet undefined.

Instance grouping is the unstructured grouping of a set of related service instances. Consider the case of the Web Services for Remote Portlets standard [OASI03]. This standard provides mechanisms for incorporating functions from several Web services into a portlet and offering them to a user from a single location. The different services are not connected together and their input is not piped from one to the other.

The next type of unconstrained aggregation is known as *recursive wiring*. This is the wiring of several services together, architecturally. The resulting aggregation allows one to pipe output from one service's operation to the input of another's. Additionally, unconnected

inputs and outputs can be then exposed on the aggregate directly making it a service as well. One example is a chain of filters, where each filter is implemented as a Web service and the data is piped through the wiring. The input of the first filter becomes the input of the aggregate, and the output of the last becomes the output of the aggregate.

This form of aggregation has been used in the area of component based software engineering, where a Lego™-like construction of components is enabled by connecting input and output 'ports'. The 'ports' are interfaces defined on the components [MATK97] [SREE02]. This wiring approach also appears in Java™, through the 'Event' concept. A class may throw an event, which is sent to all registered listeners. An event source and an event listener therefore form a wired output and input pair. In both these cases the wiring is implicit, usually part of the component definition itself. This wiring is made first class for Java in the Bean Markup Language (BML) [WCDE01] [CUWD00], an XML scripting language for creating compositions of Java beans. The connectors are made first class in [DAMT99], where typed connectors are used to connect components together.

For the area of Web services, the concept has reappeared as a 'global model'. This was presented in the Web Services Flow Language, WSFL [LEYM01]. The idea of global models is to use explicitly defined connectors known as 'plug links' that refer to the output operation from the portType (interface) of one service and the input operation from the portType of another service. A plug link also optionally contains a data adaptation map and a 'locator' that provides information for locating a compliant implementation of the target service. Therefore, the connected operations' signatures do not need to mirror each other. Any unconnected operations get exposed in the WSDL definition of the newly created aggregate, making it a Web service itself. The relation here is one of delegation, enabling recursive aggregation and similar to the concepts in the approaches above and in Microsoft's™ Component Object Model (COM): A call to one of the exposed operations gets delegated to the internal service actually implementing it [SUMS99]. The difference is that global models enable a data adaptation step before passing the input over to the internal component, and refer to interfaces and not actual implementations of the wired services/components while still enabling implementation targeting through the 'locator' mechanism. The Service Component Architecture (SCA, [BCII06]) provides a service wiring specification for recursive aggregation of components that include Web services. In all these wiring scenarios, the services themselves are driving the implementation of the aggregate.

Constrained Aggregation

We focus on three subcategories of constrained aggregation: orchestration (renamed from choreography in [KHLE03]), service domains, and agreements.

Orchestration is a constrained, proactive composition mechanism in which the aggregate drives the interactions with the services it aggregates by imposing control and data dependencies on a set of activities that carry out those interactions. Recursive orchestration is orchestration where the aggregate itself is also exposed as a Web service. The most common orchestration approaches are rooted in workflow languages and mechanisms, and an orchestration is alternatively referred to as a process. Web services orchestration languages are based on interface composition, decoupled from the actual implementations and endpoints behind them. Implementations need not be chosen at design time, opening the door for many approaches using dynamic binding to service endpoints based on relevant parameters calculated and optimized at either deployment time or run time - and possibly different for each instance of the same process model. Two examples include meeting quality of service

constraints [ZBDK03] [LEYM03], plugging in necessary adapters in case a compliant service is unavailable [MAMC03], or dynamically finding a service to bind based on those available at runtime [KLNW06] . This allows processes to handle the dynamic nature of the SOA environment in which services may change frequently, and the reuse of the same business logic with multiple providers. BPEL is the standard for Web service orchestration.

In related work, semantic information is used to create orchestrations from partial information, such as by backtracking from a business goal [SHDF03] or starting from a partially specified, annotated BPEL process [PAOS03].

A *Service Domain* is an aggregate containing a set of implementations that complement each other and/or compete with one another to collectively implement a related set of portTypes. The constraint here is that all the service implementations in the domain must implement one or more of the domain's portTypes. Examples of aggregation using service domains include [TTVX02] and what is called 'service communities in the SELF-SERV Project [BEDM02], in which the service communities themselves are exposed as Web services.

Therefore, a service domain, *SD,* is a set of portTypes *pT*. At the instance level, a service domain, *sd*, is a set of Web service ports, *p.* We then have:

$$SD = \{pT^1, ..., pT^n\}$$
$$sd = \{p_1^1, ..., p_{k(1)}^1, ..., p_1^n, ...p_{k(n)}^n\}$$

Where port $p_i^j \in sd$ is or portType $pT^j \in SD$.

Different service providers may contribute service endpoints (ports) to the service domain, and possibly register them with different qualities of service, with each provider implementing a (full or partial) subset of the domain's total portTypes. This enables a piece of middleware managing the domain to route incoming calls to the different service instances based on several factors, whether middleware or business related, such as load on the providers, quality of service requirements from the caller, or preferred customer status. Such requirements may be encoded using Service Level Agreements (SLA's) [LKDK03] or Web Service Policy assertions [HOKA03B]. The owner of the managing 'hub' specifies the rules that influence these capabilities and choices. The implementations residing in the domain are hidden from the requestor, who may not even realize such a selection is going on.

An *Agreement* based aggregation is a temporary grouping of a set of service instances into a distributed application, at the end of which a joint outcome is reached and possibly disseminated. The constraint in this mechanism is that the service instances must follow pre-defined protocol(s) in reaching this agreement. In an agreement based aggregation, the application level portTypes of the involved services are of no importance. What matters is that the service endpoints can participate in the protocol(s), which is usually supported by the middleware in which these endpoints run and not exposed on the portTypes themselves.

One realization of this aggregation mechanism is the WS-Coordination framework described earlier in the Background section, enabling the execution of a distributed activity at the end of which participants should have reached an agreement based on following a coordination protocol. This is the mainstream example of agreements in Web services.

For an example illustrating an agreement based aggregation, consider the sealed bid auction scenario presented in [LEYM03]. An auctioneer and a number of buyers and sellers modeled as Web services follow a coordination protocol throughout the bidding process. Once the winner is decided and announced, the aggregation ceases to exist. A detailed coordination protocol for multiple round sealed bid auctions and a discussion on agreement based aggregation beyond transactions are presented in [LEPO05].

Discussion

The categories presented so far lay out the basic forms of aggregation used in Web services today. There are still several dimensions in each of these categories, such as level of dynamicity (with respect to endpoint binding, semantic information, auto-generation, etc), support for different patterns, style of approach (for example, orchestration can be graph-based or calculus-based), and so on. We have not aimed to cover all the possibilities, but instead to separate out distinct approaches, highlight their differences, and group mechanisms that use them. The categorization allows us to reason about new approaches and where they fit with respect to the prior art. Most interestingly, when faced with a new problem to solve in Web services aggregation, one can look at these categories and their instantiations for input on whether to use prior art, extend a particular instantiation, or combine two or more together based on ones goals. For example, BPEL4Chor [DKLW07] combines orchestration and choreography through BPEL extensions that allow directly specifying the orchestration for each of several partners and directly connecting their sending and receiving activities.

In this thesis, we model business processes that are based on orchestration, decompose them, and then run the fragmented pieces together as a unit that replicates the function of the initial process. Therefore, no single one of these approaches is applicable on its own. In fact, we will eventually combine three: orchestration for defining the process and its fragments, agreement for coordinating between parts of the fragments that do not have enough information to act autonomously such as in the case of a fragmented scope, and recursive wiring for linking process fragments upon deployment.

2.2.2 Inter-organizational and Distributed Business Processes

In this section, we consider work on business processes involving several actors who own different parts of a process. Of these, some take a top-down approach starting from a neutral view of the interactions between the parties and working down to the local process of each partner, while others take a bottom-up approach starting from the local processes and working up to a neutral view.

Conversational approaches such as the state-machine based model in [FUBS03B] and the pi-calculus based WS-CDL [KABR04] define a neutral view of interactions between several processes. Barros et al [BADO05] provide a research agenda for this area based on a critical overview of WS-CDL, in which they highlight several shortcomings including but not limited to lack of a clear tie-in with the rest of the Web services stack and BPEL in particular, as well as support for only binary interactions. Contrast this with BPEL's 'hub and spokes' model where the interactions are described from only one partner's point of view. Such neutral approaches do not cover the private process of each partner or provide derivation rules to produce compliant implementations. They differ from this thesis in several ways of which the most important is that their first class citizen is the definition and order of *messages* exchanged between partners, whereas our first class citizen is the order and role-based assignment of *activities* in a business process.

Some projects take a top-down approach. [FUBS03A] starts from a certain conversation protocol and is focused on verification. The protocol consists of a 'conversation schema' providing the interconnections between the Web services and a 'Guarded Finite State Automaton' representing the message exchanges of the conversation. Another example of a top down approach is presented by Desai et. al [DMCS05] which focuses on flexibility and is based on Milner's pi-calculus [MILN99] . It starts from a 'business protocol' specified in an ontology based language they call OWL-P which provides commitments (rules regarding the effects of the exchanged messages) in addition to the usual constituents of a protocol such as roles and (ordered) messages. They aim for a local business process for each partner created from a local OWL-P process augmented by policies (local decisions, in this work).

A natural choice in Web services for the local implementation of each party's business process as the result of a top-down approach is to use BPEL, motivating Mendling and Hafner [MEHA05] to look into generating a BPEL process from each party out of a WS-CDL description. The result is a semi-automated approach, requiring designer input for information not provided in the WS-CDL description. A more direct top-down approach, requiring little to no model transformation, is to use BPEL4Chor [DKLW07]. This work proposes a BPEL extensions and syntax for defining a partner topology. The extensions enable one to specify which receive activity in one process should get a message sent by a particular invoke activity in another process, and so on. One can think of it as BPEL processes in swim-lanes, where sending and receiving activities are connected across the swim-lane boundaries. This goes beyond connecting only the WSDL interfaces as is the case in BPEL's partnerLink concept today. While relevant for our area of research, it solves a different problem focused on topology.

Next, we consider bottom-up approaches in which one starts with the individual processes and then wires them together to form the global view. A key challenge in this approach is ensuring that the processes can, in fact, work together. Three examples are presented here. The first is Fu et. al [FUBS04], which analyzes compatibility between interacting BPEL processes to aid in validating that the processes are indeed compatible. They do so by mapping BPEL processes to an extended guarded automata model, then mapping that into a language (Promela) that gets used by a model checker (SPIN), and finally checking it against criteria expressed using temporal logic (LTL). The second is the work of Casati and Discenza [CADI00] which defines and implements a framework for coordination and interaction between workflows that can send/receive events, based on a pub/sub mechanism for connecting the workflows together. That work does not focus on the decomposition of global processes. Additionally, it predates BPEL which has event handlers and is already able to interact with other processes by sending/receiving messages (but without pub/sub).

The last example we present of the bottom-up approach is our work on using BPEL for RosettaNet processes in [KHAL05A], and further detailed in [KHAL07A]. RosettaNet defines specific business processes between partners in the form of RosettaNet Partner Interface Processes or PIPs. An overview of RosettaNet standards is provided in [DAMO04]. RosettaNet itself provides these PIPs so that companies can use them to perform e-business transactions with each other, without RosettaNet's involvement. A PIP definition consists of a word document describing the expected message exchanges as Message Sequence Charts [RUGG96], definitions of the messages, and quality of service policies. This is a case where a central party creates a business process that governs the interactions between a number of participants of which the creating party is never a part. It would be advantageous in such cases to be able to deliver to the parties a package with one process for each of them that

governs only the part of the work it needs to care about. We provided templatized abstract BPEL processes for a common class of PIPs, an approach for specializing them to create an abstract BPEL 1.1 processes for each partner, and rules for creating compliant executable BPEL processes that the partners can follow to guarantee compliance. The approach builds on [BHKN04] which details how the RosettaNet infrastructure (RNIF) can be replaced by the Web services framework. In such approaches, an important problem is whether the executable processes are in fact compliant with the abstract processes. Several equivalence notions have been developed for programs [GLAB90]. In [KHAL05A], we validate the rules for creating executables using one such equivalence notion defined in [MART05]. It states that two processes have equivalent behavior if an executable model 'simulates' an abstract model, that is if it could replace the abstract model without requiring changes to the environment in which the process operates. This is done by comparing 'Communication Graphs' containing each process's externally visible behavior. The step of creating compliant executable processes in this RosettaNet based approach is close to the idea of public flows and transformation rules in van der Aalst's P2P work [AALS01], covered in more detail in the next section.

Such approaches are geared towards a different problem than addressed in this thesis: defining a flow involving the work of multiple partners. It is well–suited when one can assume little about the inner workings of each party and needs to focus on maintaining the contract of the interactions between them. For example, a reassignment of tasks in those approaches would require a redefinition of the main business process. On the other hand, this thesis is concerned with cases where one defines the business logic for a particular business goal, and later wishes to partition it, possibly more than once and in different ways, and then distribute and run the fragments at business partner sites.

2.2.3 Splitting Workflows

This section focuses on an area that is closer to our problem domain: projects that focus on taking an existing workflow and breaking it apart.

Dumas et al. [DFMV05] present a framework for flexible process modeling whereby a process, defined as a UML Activity Diagram, is translated into an event based application and run in a space-based computing runtime. While created for flexibility not for distribution, it could be used to run a fragmented process because the execution of the process happens through a coordination space that effectively provides shared memory. The space they use can handle 'active objects' that can perform work in their own threads. The process is executed through the use of 'coordinators', active objects that can respond to events, write data, and do work. An action in an activity diagram is mapped to a controller of type 'connector' that handles the work itself, and a number of controllers of type 'router' that have input sets encoding the ways that one could arrive at this action. Different process artifacts are mapped into controllers.

The SELF-SERV Project [BEDM02] [BEDS05] is a system created to enable rapid composition of Web services: compositions are declarative and are executed in a distributed (peer-to-peer) fashion. Each service is associated with a coordinator that coordinates its work with other services involved in the same compositions. A scheduling algorithm is not needed, as each coordinator get its necessary information (routing tables, location, etc) statically from the representation of the composition, which is itself a state-chart. SELF-SERV's composite service has a special type of coordinator, known as the 'initial coordinator'. When it is invoked, it sends messages to all states that can start. From there, the rest of the work of the

composite service is scheduled through messages between its internal coordinators: A state's coordinator receives notifications from other coordinators and determines when it should in fact enter its state. Every time a state completes, its coordinator sends a completion notice, containing all needed data items, to the coordinators of all other states that may need to execute next. The coordinator may receive external events while it is running, which may cause it to abort its service execution and send the completion notice early. Once the composite service completes, the initial coordinator is notified and can respond to the invocation that started it. The messages between coordinators are called 'control-flow notifications' while the messages between a coordinator and its service are called 'service invocations/completions'. This is similar to our work in this thesis in that it also uses a kind of coordinator, and that we also provide information about the distributed process statically: as a list of name value pairs and a tree containing scope relationships and metadata.

In both these approaches, Dumas et. al. and SELF-SERV, semantics is maintained across split processes. The approaches, however, are not inline with our goals: the use of non-BPEL meta-models (UML Activity diagrams, state charts), the requirement of new middleware (coordination space, SELF-SERV runtime), and lack of transparency because runtime artifacts are specified in different meta-models (controllers, coordinators) than build-time artifacts (the process model itself).

Van der Aalst provides a Petri Net [REIS85] based approach in [AALS01] for creating a multi-party process and then splitting it between different partners. A 'public workflow' encodes both the logic at each party as well as the message exchanges between the parties. The public workflow is defined as a Petri-Net based 'Workflow Net', with interactions between the parties defined using a place between two transitions (one from each). Then, the flow is divided into one public part per party. The work also provides transformation rules used to create the private flow for one partner from its public one that was retrieved from the Workflow Net. Drawing a parallel to BPEL, the transformation rules would be similar to rules regarding abstract to executable process compliance and not for the creation of a fully executable process as a fragment from a larger, also fully executable, process. An earlier version of this work using Message-Sequence Charts is in [AALS99]. Dijkman and Dumas [DIDU04] identify and provide a formalism of four aspects of 'Service Oriented design' to be the core parts of defining multi-party applications. These aspects are interface behavior, provider behavior, choreography and orchestration. Their approach models relations between these aspects to aid in verification. It extends [AALS01] and is also Petri-Net based.

The Mentor project [MWWK98] presents a system for running split processes modeled as a state chart (control flow) and activity chart (data flow). Each partition is executed in one workflow engine in the system. Each engine can directly execute the work to be done by its partition's subset of the activities, and contains a work list manager that manages assigning work to actors as well as a log manager that tracks the work for recovery purposes and a history manager that does so for monitoring. The coordination of the work of the partitions in their respective engines is done through posting local changes of an engine's 'configuration' to other engines for which the change may be relevant. When a change occurs in one engine, the local communication manager is notified. It then determines which engines need the configuration update and sends them the change. The recipients' engines then correspondingly update their local configurations. They start with synchronizing after every step, but then relax this restriction with different mechanisms to reduce the number of messages. As in our work, the partitions are created by the process designer and the resulting fragments are themselves in the same model as the original process. However, it is non-BPEL

(predates it), uses a data flow model which simplifies the synchronization of configurations, and requires specialized middleware to manage the propagation of changes. In our approach we can handle variable-style data in standard BPEL as well as the data link variant. We use activities in the fragments and only require new middleware for splitting highly coupled artifacts (loops and scopes), even then using the standard-based WS-Coordination middleware to maximize reuse.

OSIRIS [SWSS03] is a hyper-database system which includes a peer-to-peer implementation of a workflow system. It enables executing a single process model by distributing it across the peers for optimized use of the hyper-database environment. Distribution means to split a process model apart into the smallest units that can be executed locally. Such splits are not partner-based, which is in contrast to our approach. The decision of where to execute each activity is dynamic and calculated at runtime based on available resources and actual load. The system routes process instances directly from one executing node to the next, using a single join node for every process instance where multiple paths get merged. It does not state which process meta-model is used.

In [CCMG04] [GOCS04], Gowri Nanda et. al. break down a BPEL process into several BPEL processes using program analysis and possibly reordering nodes, with each process deployed and executed on a separate machine, in order to maximize the throughput in cases where multiple instances of a process are running concurrently. They define new Workflow Dependency Graphs as an alternate to Threaded Program Dependency Graphs [SASI93] in order to model the control and data dependencies in the flow. The work is grounded in automatic program parallelization theory and focuses on program rewrite, which conflicts with our goals of transparency and minimizing graph changes. Also, important aspects for generic partitioning are not addressed: instance identification and matching, splitting loops, splitting scopes, and so on. They do not handle exceptions – and in particular cannot handle dead-path elimination. However, there are some similarities with our work in this thesis: data and control dependencies are communicated using BPEL activities and data dependencies are only propagated if they follow the control flow.

Another important difference between both, OSIRIS and the work of Gowri Nanda et. al., and our approach is that their partitioning is computed by the system and cannot be chosen by the designer: the fragmentation has no business semantics at all.

Finally, we mention two newer works on split processes. The first is Baresi's work [BAMM06] on mobile workflow which proposes mechanisms to split BPEL processes using send/receive activities to transmit control and data. It is not clear from the text whether the resulting process model is in fact BPEL. This work is still early stage and stays mainly at the requirements level. It provides a similar intuition to our approach for the simple cases of propagating data and control [KHLE06], but leaves the hard problems for future work therefore not providing a full solution. The other work is by Yildiz [YILD07], which focuses on the theoretical as opposed to the applied aspects of splitting. They define a process meta-model that appears close to (but is not) a subset of BPEL. They then split processes in this meta-model. They do cover Dead-Path Elimination for flat (non-structured) processes. As in our early paper on splitting [KHLE06], which they cite, they assign different activities to different partners, provide sending and receiving activities for propagating control and data, and assume that data dependencies are in the unsplit process model using data edges. They provide a mathematical treatment of sending and receiving control/data in that model and ensure soundness of the resulting processes. Since their model is not BPEL, they do not

provide the mechanisms for determining activity state in order to send it with the messages, and running their approach would require all partners to use a runtime supporting this process model. They do not split scopes or loops (especially fault and compensation handling) nor variable-style standard BPEL as we have done since in [KHKL07B]. Neither Baresi nor Yildiz provides, as we do, patterns of BPEL constructs that enable us to leverage BPEL's built-in behavior itself in order to reproduce the split behavior at the fragments - such as for faults, compensation, dead-path elimination, correlation, and others.

Workflow patterns for control flow and service interactions are studied and categorized in [AWBH00] and [BADH05] respectively. However, these patterns cover a different area than the ones in this thesis: they encode recurring requirements gathered from several, different workflow systems and languages in the first case and between Service Oriented applications in the second. In contrast, the patterns in this thesis are patterns of BPEL activities that reproduce BPEL behavior upon fragmentation.

2.2.4 Coordination in Web Services

Coordination is an active area of study in distributed systems. In this section, we focus our discussion on coordination as it relates to Web services. The interested reader is referred to [PAAR98] for an overview and categorization of several coordination models and languages besides those used in Web services.

The previous versions of the BPEL specification provided a section closely relating the semantics of the Business Activity protocol of WS-Coordination [CGKL03] to how a parent BPEL scope interacts with and controls the lifecycle of its child scopes at runtime. An in-depth comparison between WS-BusinessActivity and the parent-child scope relationships in a single BPEL process is provided in [SAME05]. In the BPEL proposal for Sub-Processes [IBSA05], BPEL language extensions are provided that enable parts of a process to be marked as a 'subprocess'. The subprocess can then be called from or imported into a specific activity in another process, called the 'parent' process. The document also defines a new WS-Coordination protocol to coordinate fault propagation, termination, and compensation between the parent process and the subprocess. BPEL4People's WS-HumanTask protocol enables a BPEL process that has a people-facing activity to coordinate a remote human task. The relationship is again that of parent and child. However, as Chapter 6 will show, there are seminal differences between coordinating nested items and coordinating fragments of one item.

Tai et. al [TAKM04], [TMWD04] enabled adding transactional capabilities to BPEL constructs such as interaction activities in the same scope or related to the same partnerLink. It declared the needed capabilities using a new policy language for transactional requirements, related to WS-BusinessActivity and WS-AtomicTransaction, which users can attach to the BPEL constructs. It also provided a description of how to combine transactional behavior with the BPEL behavior itself, and how the middleware interacts with the workflow processor. In follow-on work [TAIS05], Tai shows that while adding quality of service capabilities to a process may be orthogonal to the programming model, the middleware subsystems providing the quality of service capabilities may not be easily composable: they may need to work tightly together and sometimes with the workflow system itself. In this thesis, we instrument the BPEL engine to enable the coordination protocols for split loops and scopes to affect the navigation.

Having just touched on quality of service capabilities and their use in BPEL, we present a few related projects on this topic before coming back to coordination in particular. Different types of service policies have been called 'features' in GlueQoS [WTMR04]. It provides an approach for dynamically managing the different features required by different parts of a service oriented application. The different parts declare their QoS requirements as policies in the GlueQoS language. A protocol is provided that enables the parts to negotiate their different feature requirements before beginning their application level interactions. Once an acceptable feature set is agreed on, it is chosen and executed. Several projects have aimed at adding capabilities to business processes dynamically at runtime. Some use an Aspect Oriented approach, such as AO4BPEL [CHME04] and the work by Courbis and Finkelstein [COFI05]. We point the interested reader to our survey and outlook in [CUKM08] for the use of policies in Web services.

A comparison between WS-Coordination and BTP and an overview of both are provided in [FUGR02] [LIFR03]. The former expresses doubts that WS-Coordination protocols are needed that are not for two-phase commit or for distributed transactional work. However, the auction scenario in [LEPO05] shows that there are other use cases for WS-Coordination protocols because it is a generic framework for coordinating long running activities. BPEL4People and BPEL for Sub-Processes also present examples of coordination usage besides two-phase commit, as does the use of coordination protocols in this thesis.

None of that work addresses coordinating BPEL scopes or loops fragmented across several processes. WS-Coordination protocols have mostly not involved the coordinator in any application specific data. In this thesis, coordination messages may carry application data (i.e. the value of the fault variable) and the coordinator is provided with a-priori information about the participants (i.e. their location, capabilities, etc.).

2.3 Conclusion

This section has provided background information about the Web services framework, in particular WSDL, WS-Addressing, BPEL, and WS-Coordination. Then, it provided the first contribution of this thesis: a categorization of existing Web services aggregation techniques.

The related work section has shown that the area of splitting business processes has been approached in many different ways: top-down, bottom-up, by mapping to new artifacts, and/or by using dedicated middleware. It has also been done with different goals in mind: decreasing latency, optimizing use of networked resources, injecting behavior, and finally, as we do in this thesis, to support a business need.

The related work section has also identified the similarities and differences between our thesis and existing work. While the concept of using sending and receiving activities to propagate simple, direct control and data dependencies has been used before, the idea of using BPEL in BPEL for fragments is quite novel. The approaches we are aware of either do not deal with distributed fault handling and/or compensation based recovery or require specialized, new middleware for that purpose. Additionally, our approach is also novel in both executing split loops and scopes as well as its use of WS-Coordination for this execution. The resolution of data races resulting from process fragmentation using patterns of business process model artifacts (sending and receiving activities), instead of new middleware such as a shared database, is new as well.

Chapter 3 A Process Meta-Model

Today's process meta-models are either calculus-based or graph-based. Languages like BPEL, however, combine both approaches at the language level. There is no formal process meta-model that directly embodies the concepts introduced by BPEL. In this section, we introduce such a formal meta-model. In addition, this meta-model supports the fault handling capabilities of BPEL; in fact, it turned out that fault handling is a canonical enabler for combining both approaches. This meta-model significantly extends the one in [LERO00], which defines a graph-based model for workflow definition and execution.

The BPEL specification does not have formal semantics. Several proposals have been made in academia. These include the use of Petri Nets [SCST04] which suffers from a scalability problem as it results in very large nets for simple BPEL processes, the use of Abstract State Machines such as [FAGV05], and the use of pi-calculus in [LUMA07]. BPEL combined the graph oriented approach used in workflow for the last two decades and languages such as IBM's WSFL [LEYM01] with the structured, calculus-based approach used in formalizations such as pi-calculus and languages such as Microsoft's XLANG [THAT01]. Historically, major industry players have implemented either pure graph based or pure calculus-based process meta-models and have had to adapt to BPEL's mixed model. In [CKLW03], we showed that BPEL's ability to unifying the graph based and structured programming models is through its special exception handling mechanism which is able to propagate faults both, up the scope hierarchy (vertical) and along control links (horizontal). This is a key difference from pure graph-based models, and thus presented a driving force for some of the directions of this process meta-model. We will show how BPEL process models can be mapped onto the meta-model introduced in this chapter.

In a nutshell, our meta-model allows to specify process models as a set of activities joined by directed edges expressing conditional control flow, as well as loops and purely nested scopes. Our meta-model does not make assumptions about the concrete implementation of an activity, but only represents the abstract piece of work that has to be performed when the activity is determined to be executed. Scopes group activities to provide them with common properties, such as fault and/or compensation handling. The edges, loops, and scopes together provide the ordering semantics for the set of activities in the process.

Dead-Path Elimination in this meta-model is derived from BPEL's approach: it occurs by surrounding each activity on which it is to be enabled with a scope that can handle a special fault known as 'joinFailure'. This fault occurs if the joinCondition of the activity evaluates to false. The addition of scopes enables enforcing behavior on a group of activities. Most importantly, by mirroring BPEL's capability of making the join failure a special fault handled by the fault handling mechanism of a scope, we are able to skip multi-activity process regions in one step. As a result, we are able to map several important algebraic process constructs to our modified graph-based approach.

This is in contrast to how navigation occurs in [LERO00] where navigation simply walks the graph: An activity is scheduled for execution once all its incoming edges have been traversed and its join condition evaluates to true. If that condition evaluates to false, then the activity is

disabled and the status of every link leaving the activity is set to 'false'. Therefore, the meta-model navigates down possible paths until it reaches another activity whose join condition is true.

Our meta-model makes fault handlers a first class entity: they are reached upon the occurrence of a matching fault anywhere inside the scope the handler is associated with. A cursory approach would suggest simply using control links to link from activities that can fault to the activities in the fault handler. However, this has several disadvantages. As the number of activities that can throw a fault increases, so does the number of links to the same activities that form the fault handling. The fault handling behavior would then become mixed in with the core business logic and hard to extract. Finally, fault handling lends itself well to being applied to regions of work. For example, in several programming languages one talks of a block of code to which the fault handling logic is attached. Faults are thus able to propagate up a hierarchy of possible handlers. BPEL exhibits this same behavior, with fault handlers attached to an entire scope and faults able to propagate up the scope hierarchy. Therefore, there is clear benefit in bringing the concept of scopes with fault handlers directly into the meta-model. Additionally, as claimed in [CKLW03] (and shown in the following), fault handling is the only significant mechanism needed to support the mixing of the two process modeling approaches at hand.

Compensation handling is another important feature covered in this meta-model. It involves the undoing of completed work by running associated 'compensation handlers' in reverse order of completion of that work. Compensation in process models that combine calculus and graph based approaches, such as BPEL, present special challenges in computing the reverse order. The reason is two-fold: First, the unit of compensation (scope) can be nested. Second, control links can be from/to the boundary of the units of compensation (scopes) as well as from/to activities inside that unit. In section 3.6, we describe BPEL's compensation model and provide a modification of that in the meta-model that is both simpler and more flexible.

Scopes are modeled using a hypergraph. A scope may contain several activities and is related to other scopes by a parent-child relationship. Where a graph represents one-to-one relations, a hypergraph enables on to model many-to-one relations [GASC99]. An edge in a hypergraph, called a hyperedge, associates a set of nodes to a single entity; therefore each scope becomes such a hyperedge associating a set of activities with the scope itself. A directed graph of the hypergraph is then used to encode relationships between scopes. Currently, we only encode parent-child relations. A natural extension of this approach would be to enable associating other properties of scopes as well and using new types of edges in the graph of the hypergraph to encode resulting scope relations.

3.1 Definition

The process meta-model consists of several artifacts. Note that we use $\pi_i(f)$ to represent the i-th projection map. We define the elements of the process meta-model as follows (details will be defined in the rest of the chapter):

Notation	*Definition*
P	Process in the meta-model, $P = \{N, E, C, H, G, F, M, J, W\}$
N	Set of nodes, where each node is an activity
E	Set of links, i.e. directed edges weighted by Boolean transition conditions. $E = N \times N \times C$
C	Set of all conditions

S	Set of named scopes, where *name(s)* retrieves the scope name
$H = (N, E_H)$	Hypergraph of scopes. Each $e_H \in E_H$ corresponds to a scope
$G = (N_G, E_G)$	Graph of the hypergraph of scopes
F	Set of all fault handlers
M	Set of fault names
J	Set of compensation handlers
W	Set of named loops, where name(w) retrieves the loop name
$\psi(a)$	Implementation of activity *a*
$\phi(a)$	Join condition of activity *a*
$E^{\leftarrow}(a)$	Set of links entering *a*, $E^{\leftarrow}(a) = \{(x, a, p), x \in N, p \in C\}$
$E^{\rightarrow}(a)$	Set of links leaving *a*, $E^{\rightarrow}(a) = \{(a, x, p), x \in N, p \in C\}$
A(s)	The set of activities in a scope *s*
M(s)	Subset of *M* containing the fault names caught by handlers of scope *s*
$F(s) = \{f_1, ..., f_n\}$	Set of fault handlers of scope $s \in S, F(s) \subseteq F$
$f_i = (\mu, p_f, s)$, *where* $\mu \subset M(s), p_f \subset N$	A fault handler $f_i \in F(s)$ that is defined on scope s. μ is the set of the names of the faults that this handler handles. p_f contains the activities in the fault handler
$A(f)$	The set of activities $p_f = \pi_2(f)$
$j(s) = (P_j, s)$	A partial function returning the compensation handler associated with a scope. There is at most one for each scope. Each compensation handler contains a process meta-model.
S_{comp}	The domain of *j(s)*. It is the subset of *S* for which explicit compensation handlers are defined.

Table 1: Notation for the process meta-model

3.1.1 Activities

The activity is the unit of work of the process meta-model. It may be the source and/or target of multiple links. If it is the target of one or more links, it must define a 'join condition', $\phi(a)$, in terms of the status of its incoming links. The status of a link is usually the truth value of its transition condition. An activity may read and/or write data and throw faults during its execution, but is otherwise uninterruptible.

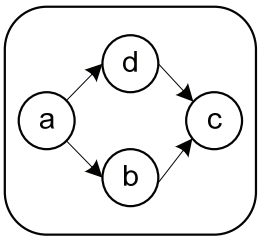
3.1.2 Links

A link is a directed edge between two activities. It is defined as a tuple of a source activity, a target activity, and a transition condition. A transition condition is a Boolean function that is a function of the data visible to an activity that is the source of the link. A link imposes control semantics on the activities such that the target activity cannot begin executing before the source activity completes.

An activity may have zero or more outgoing links. It may also have zero or more incoming links, along with a 'join condition' in terms of the status of each incoming link that determines whether the activity will run or not. Therefore:

$$|E^{\leftarrow}(a)| \geq 0, |E^{\rightarrow}(a)| \geq 0$$

For example, consider the process model illustrated in Figure 2. It has four activities and four links. Assume all the transition conditions are 'true' and the join condition of *c* is the disjunction of the status of the incoming links. We encode this in the meta-model as shown in the right of Figure 2.



$$N = \{a, b, c, d\}$$
$$E = \{(a, b, true), (b, c, true), (a, d, true), (d, c, true)\}$$
$$C = \{\bigcup_{e \in E} \pi_3(e), \phi(c)\}$$
$$\phi(c) = \vee_{e \in E^{\leftarrow}(c)} e$$
$$H = \{N, \{N\}\}$$
$$G = \{\{N\}, \emptyset\}$$
$$F = M = J = W = \emptyset$$

Figure 2: Process example

The joining of paths in this meta-model constitutes synchronizing joins: the status of all incoming links must be known before an activity can evaluate its join condition and determine whether it can run. Therefore, the graph formed by the links is acyclic.

Cycles would cause deadlocks. For example, consider a business process where the designer wishes to loop back to activity *b*. The designer may attempt to create a process similar to the one in Figure 3. However, notice that activity *b* will not be able to execute until the status of both its incoming links are known. In other words, *b* cannot run until after both *a* and *c* have run - which will never happen.

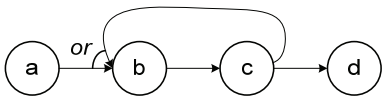


Figure 3: Invalid process showing why cycles are disallowed

This behavior of links and activities is the same as in [LERO00]. No two links share the same source and the same target activities. We add two more restrictions on links: (1) a link must not cross into a scope's fault handler from outside that handler and (2) a link must not cross the boundary of a loop.

3.1.3 Scopes

The purpose of a scope is to define properties that affect navigation on the set of activities enclosed within in the scope. Such properties include collective fault and compensation handling. A scope that defines collective fault or compensation handling capabilities is called a fault handling scope or a compensation handling scope, respectively. Scopes in this meta-model must be strictly nested. Every process has at least one scope containing all the activities in the process, this is the root scope s_r.

Fault handling scopes provide the ability to perform dead-path-elimination (DPE) behavior [LEAL94] because DPE simply reduces to handling a particular kind of fault thrown if an activity's join condition evaluates to false [CKLW03]. In BPEL, such a fault is known as 'joinFailure': Basically, once the links entering an activity have all been traversed, the join condition on that activity gets evaluated. If the result of evaluating this condition is false, then a fault known as 'joinFailure' is thrown to the scope to which the activity belongs. We often say that 'an activity throws a join failure' to mean that the environment will throw the join failure fault on behalf of the activity. The navigation section will show how fault handling will result in dead-path elimination being propagated at runtime.

A grouping mechanism is needed to denote the set of activities in a scope. A scope is therefore modeled as a hyperedge in a hypergraph, where the hyperedge consists of all the

activities in the scope. A scope *s* therefore corresponds to a hyperedge in the hypergraph *H* of the process.

$$s = e_H \subseteq N$$

The activities in a scope include the activities, $a_1, ..., a_n$ in the scope body and the activities in the scope's fault handlers. Therefore, defining *A(s)* to be the set of activities in a scope *s* and where $\bigcup_{f \in F(s)} A(f)$ is the set of all activities in the fault handlers of *s,*

$$A(s) := \{a_1, ..., a_n\} \cup \bigcup_{f \in F(s)} A(f)$$

The activities in *A(s)* define the hyperedges of *H*,

$$e_H(s) := A(s)$$

The graph of the hypergraph, *G*, defines relevant relationships between scopes. The relationship we are most concerned with is that of scope nesting. The edges in the *G* represent this parent-child relationship, with edges leading from a child to its parent.

Lemma: *G* is a rooted, directed tree with the process scope as the root scope.

Proof: By definition, scopes are strictly nested. For *G* to be a rooted tree we must prove that there is exactly one node that has only incoming edges, and all other nodes have exactly one outgoing edge and zero or more incoming edges. As the process scope has no parents by definition, it is the root node. Additionally, any scope may have zero or more children, resulting in zero or more incoming edges for the scope node in *G*.

We still have to prove that there can only be one outgoing edge for all non-root nodes. We do this by contradiction: Assume that a node has more than one outgoing edge. Thus, the scope has multiple parent scopes because an edge represents a child-parent relationship between scopes. A scope with two or more parents means that the scope is nested in all of them. It also means that the parents themselves are siblings, not nested in each other. However, strict nesting requires that sibling scopes have mutually exclusive activity sets. Thus a node cannot have more than one outgoing edge. Finally, since all scopes in the process are nested at some level in the root scope, no other scope has no parent and each node must have at least one outgoing edge. Thus, *G* must be a rooted directed tree rooted at the process's root scope.■

Each scope is represented as a node in the graph of the hypergraph. In other words, the node n_G is equivalent to the hyperedge e_H in the hypergraph corresponding to a scope. The edges, E_G, of the graph of the hypergraph connect scopes to their logical parent scopes, i.e. $N_G := S$.

An activity belonging to a scope *s*, must also belong to the parent scope of *s*. However, the inverse is not true. Consider a scope with two child scopes. The parent scope contains the activities of both children. However, the activities of the sibling scopes are mutually exclusive because of the strict nesting requirement:

$$\forall e = (s, s_p) \in E_G : a \in A(s) \Rightarrow a \in A(s_p)$$

Lemma: Iff $a \in A(s_1) \cap A(s_2)$, then at least one path to the root s_r exists containing both s_1 and s_2.

Proof: '⇒' If $a \in A(s_1)$ and $a \in A(s_2)$, then s_1 and s_2 are overlapping scopes. Since scopes must be strictly nested, then any scopes with overlapping activity sets are nested. Thus, s_1 and s_2 are nested. The edges of G represent the child-parent relationships. Therefore, there must be a path in G between s_1 and s_2. Since G is a tree, one can have a path from any node to the root scope s_r. Therefore, there is path in G containing s_r, s_1, and s_2.

'⇐' If there is a path between two scopes s_1 and s_2, then one scope contains the other scope. Thus, the activity set of one scope is a subset of the activity set of the other. ■

Consider the process model in Figure 4. It has four scopes, s_r, s_1, s_2, and s_3 with activities: $A(s_r) = \{a, b, c, d\}, A(s_1) = \{a\}, A(s_2) = \{c, d\}, A(s_3) = \{c\}$. The graph of the hypergraph is shown on the right. It is the only valid *G* for this process model given the restrictions above. For example, s_3 cannot be the parent of s_2 because it does not contain *d*.

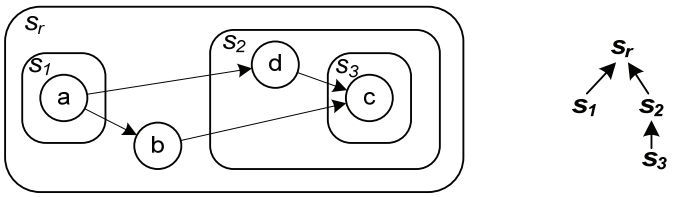


Figure 4: Graphical representation of a process model and its graph of the hypergraph

Fault Handlers

A scope may have zero or more fault handlers. A fault handler is defined on exactly one scope and has a set of fault names μ that are the names of the faults it can catch.

$$\forall f \in F : |\pi_1(f) = \mu| > 0$$

Consider μ_i to be the set of names of the i-th fault handler, f_i, of a scope *s*. So, $\mu_i = \pi_1(f_i)$. No two handlers on the same scope are allowed to handle faults with the same name:

$$\mu_i = \mu_j \Rightarrow f_i = f_j$$

A scope may have a 'catch all' handler that is meant to catch all faults not caught by the other handlers of the scope. This is the only case in which the size of the set of names of a fault handler is allowed to be greater than one. Additionally, there may be only one such handler on a particular scope. Let *I* be the set of indices *i* of all fault handlers of scope *s* where $|\mu_i| > 1$: $I := \{i \in \{1, ..., n\} || \mu_i| > 1\}$. There can be at most one 'catch all' handler on each scope, i.e. $|I| \leq 1$.

Therefore, the set of fault names that the 'catch all' fault handler can catch consists of all the fault names except those used in other fault handlers of this scope.

$$|\mu_i| > 1 \Rightarrow \mu_i = M - \bigcup_{f \in (F(s) - \{f_i\})} \pi_1(f)$$

Each fault handler *f* is defined as a tuple $f = (\mu, p_f, s)$, where p_f is the set of activities constituting the work of the handler. The activities of p_f belong to *A(s)*. We define a set *E(s)* containing all links whose source belongs to *A(s)*. Scopes can group activities that are inside a

fault handler: For example, the fault handler of scope s_2 in Figure 5 has two scopes, one nested in the other.

A scope is said to be in a fault handler if all the scope's activities *A(s)* belong to the activities p_f of the fault handler. A scope s_f that is in a fault handler but whose parent scope is not in the same handler has the scope of the fault handler as its parent in *G*: For example, notice that in the graph of the hypergraph for Figure 5 scope s_4 has s_3 as its parent, while s_3 itself has s_2.

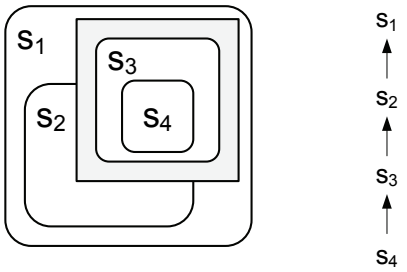


Figure 5: Scopes in fault handlers

The activities in a fault handler cannot be the targets of links whose sources belong outside the handler. However, an activity in a fault handler may be the source of a link whose target is not in the handler. Therefore, we have the following restriction:

$$\forall s \in S, (x, y, p) \in E : y \in \bigcup_{f \in F(s)} p_f \Rightarrow x \in \bigcup_{f \in F(s)} p_f$$

Common Faults

Every process of this meta-model must support two built-in faults, a fault that signals a join failure (*join_failure*) and another that signals activity timeout (*activity_timeout*). The corresponding set of fault names is called M_0. Each process may define additional faults.

$$M_0 := \{join\_failure, activity\_timeout\} \subseteq M$$

3.1.4 Loops

The purpose of a loop is to group a set of activities so that they can be executed zero or more times until a certain condition evaluates to false. A loop is modeled as a special type of activity. It contains a set of activities and a condition. A loop is handled more simply than a scope because links cannot cross the boundary of a loop.

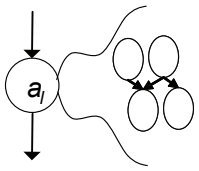


Figure 6: A loop, a_l, containing four activities

Therefore, a loop $w \in W$ is defined as follows:

$$w := (c, A_l) \text{ where } c \in C, A_l \subseteq N$$

Define a map, K, that associates with every loop an activity in the process. This activity represents the place of the loop in the control flow:

$$K : W \rightarrow N$$

The immediately enclosing scope of a loop w is the immediately enclosing scope of $K(w)$. We say a loop $w \in W$ is nested in a scope *s* if $K(w)$ belongs to *A(s)*. Additionally, the set of activities of an enclosing scope of a loop contain the set of activities A_l of the loop: $K(w) \in A(s) \Rightarrow A_l \subset A(s)$.

We say *a scope is nested in a loop* if all the scope's activities *A(s)* are in the set of activities of the loop A_l and the scope activities do not include $K(w)$.

3.2 Navigation

We have so far presented the definitions of the constructs of the process meta-model. In this section, we define the navigation semantics of the meta-model for a given process instance.

3.2.1 Reflecting Time

We use the set of natural numbers to represent time, starting from 0 for when an instance is created. The current time is incremented by 1 whenever an activity completes or an activity faults. The time is also incremented when a loop iteration completes and a new iteration starts, which occurs in three consecutive time steps as will be explained in section 3.2.8. Activity completion and fault notification are always processed sequentially even if they actually happened in parallel. At any point in time $t \in \mathbb{N}$, the states of the elements of the process instance are computed using the time-sensitive maps defined in this section. Navigation consists of stepping through a process instance from the time of its creation (*t=0*) until its completion.

3.2.2 Activity Lifecycle

The intuition for navigation is that any activity that is a start activity of the process or whose join condition evaluates to true will activate. A process start activity is one with no incoming links and that is not in a fault handler or a loop. All process start activities activate at instance creation time, *t=0*.

Once an activity activates it performs the work prescribed in its implementation. When that work is complete, the implementation of the activity returns. An activity then completes and the links leaving it fire with the value of evaluating their transition conditions. However, as noted earlier, an activity may also throw a fault. If a fault occurs, the scope hierarchy is searched for a compatible fault handler. Once a fault handler is reached, then all activities in the scope of the handler stop execution and all links whose sources are in that scope and whose targets belong outside that scope will fire with a false value. The first activity of a discovered fault handler will activate and navigation will continue as above. If no handler is found, the process instance is terminated. The join condition of an activity is evaluated once all the activity's incoming links have fired.

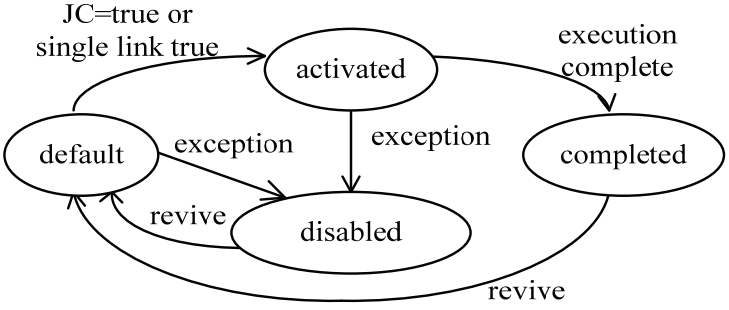


Figure 7: Activity lifecycle

The activity lifecycle definition is needed for navigation. At any point in time, *t*, each activity in the process instance is in one of the four states illustrated in Figure 7, and where JC stands for join condition. When an activity enters each state will be formally defined in the rest of this section and specifically in section 3.2.10.

default: All activities are in this state when the process is instantiated. This state may be entered again to allow an activity in a loop to be restarted.

activated: An activity enters this state when the conditions for it to run are satisfied.

completed: An activity enters this state after its implementation returns.

disabled: An activity that has been disabled enters this state. This occurs in two cases:
- A fault occurred in its scope when it was either in the 'default' or 'activated' state. This may happen because it threw a fault itself, including the join failure before activation or any other fault during execution. It may also happen if another activity nested in its scope threw a fault that could not be handled at a lower level of scope nesting.
- Its scope completed successfully and it belongs to one of the fault handlers of that scope.

We define $t_0 := 0$ to be time the process is instantiated, $t_j(a)$ the time when the join condition of activity a is evaluated, and $t_a(a)$ the time when the activity is activated, $t_c(a)$ the time the activity completes, and t_{max} the timeout interval for activity completion. If this timeout is exceeded, then the activity faults with an *activity_timeout* fault. An activity *a* may throw a fault at any time, $t_f(a)$, such that:

$$t_j(a) \leq t_f(a) \leq (t_j(a) + t_{max})$$

Additionally, an activity must complete before the timeout interval:

$$t_j(a) < t_a(a) < t_c(a) \leq (t_j + t_{max})$$

No two activities may complete or fault at the same time, therefore:

$$t_c(a) = t_c(b) \iff a = b$$

$$t_f(a) = t_f(b) \iff a = b$$

3.2.3 Activity State Map

We define Υ to be the set of activity and predicate states:

$$\Upsilon = \{default, activated, completed, disabled, evaluated, not\text{-}evaluated\}$$

The predicate states 'evaluated' and 'not-evaluated' will be defined in section 3.2.7. The activity state map associates the state of an activity with a point in time, $t \in \mathbb{N}$. We define the activity state map ω as:

$$\omega : \mathbb{N} \times N \rightarrow \Upsilon$$

Therefore, $\omega(t, a)$ provides the state of *a* at time t. We will often refer to activities that are completed or disabled at a particular time. Thus, we define the following maps:

The completed activities map λ, where

$$\lambda(t) \subseteq N \text{ and } \forall a \in \lambda(t) : \omega(t, a) = completed$$

The disabled activities map δ, where

$$\delta(t) \subseteq N \text{ and } \forall a \in \delta(t) : \omega(t, a) = disabled.$$

Two more maps, used for faults, are defined. The faulted activities map, ρ, where $\rho(t) \subseteq N$:

$$\rho(t) := \begin{cases} \emptyset & t = 0 \\ \rho(t-1) \cup \{a\} & t \geq 1 \wedge a \text{ faulted at time } t \\ \rho(t-1) - \{a\} & a \in \rho(t-1) \wedge \omega(a, t) = default \\ \rho(t-1) & otherwise \end{cases}$$

Note that an activity that faults enters the disabled state, so we also have $\rho(t) \subseteq \delta(t)$. The third condition removes from $\rho(t)$ an activity that had faulted previously but whose state is set back to default (i.e. it is in a loop).

The waiting fault handlers map $z(t) \subseteq F$. This contains the set of handlers that have been determined as able to catch a fault thrown by an activity at time *t*, when that activity faulted, but for which the associated scope needs to wait for running activities to complete before the fault handler can start. It will be fully defined in section 3.2.9.

3.2.4 Process Start Activities

The start activities of a process in this meta-model are the first activities that can run. A process may have more than one start activity. These activities have no incoming links and do not belong to any of the fault handlers or any of the loops. Therefore, we define the set of process start activities N':

$$a \in N' :\Leftrightarrow (E^{\leftarrow}(a) = \emptyset) \wedge a \notin \bigcup_{f \in F} A(f) \cup \bigcup_{w \in W} \pi_2(w)$$

3.2.5 Fault Handler Start Activities

The start activities of a fault handler are the first activities that can run when (and if) the handler catches a fault. These activities have no incoming links and belong inside a fault handler but not in a loop contained in that fault handler. Therefore, we define the set of fault handler start activities, N'':

$$a \in N'' :\Leftrightarrow (E^{\leftarrow}(a) = \emptyset) \wedge a \in \bigcup_{f \in F} (A(f) - \bigcup_{w \in W, K(w) \in A(f)} \pi_2(w))$$

We denote by $N''(f)$ of a specific fault handler *f* the subset of N'' whose elements are activities of the fault handler but not in a fault handler nested in *f*.

$$N''(f) := N'' \cap (A(f) - \bigcup_{f_i \in F - \{f\}, A(f_i) \subset A(f)} A(f_i))$$

3.2.6 Loop Start Activities

The start activities of a loop are the activities in the loop that have no incoming links:

$$a \in N''' :\Leftrightarrow (E^{\leftarrow}(a) = \emptyset) \wedge a \notin N' \cup N''$$

We denote by $N'''(w)$ for a specific loop, w, the subset of N''' minus the activities of all other loops in the process:

$$N'''(w) := N''' - \bigcup_{w_i \in W - \{w\}} \pi_2(w_i)$$

3.2.7 Predicate States

The states of the predicates in the process play a major part in the navigation. The predicate states are similar to [LERO00]. At any point in time, *t*, the state of each predicate is either evaluated or not-evaluated depending on whether its truth value has already been determined. The truth value of a predicate that has been evaluated is either true or false. Therefore we need one more value to refer to it when it is not evaluated. For that, we use the 'unknown' value denoted by '?'.

We use the following notation to represent the conditions of an activity's incoming and outgoing links:

$$C^{\leftarrow}(a) := \{\pi_3(e) | e \in E^{\leftarrow}(a)\}$$

$$C^{\rightarrow}(a) := \{\pi_3(e) | e \in E^{\rightarrow}(a)\}$$

We define the predicate state map that associates a state to a predicate at any point in time *t*:

$$\xi : \mathbb{N} \times C \rightarrow \Upsilon$$

- $\forall p \in C : \xi(0, p) = not\text{-}evaluated$. That is, all predicates are 'not-evaluated' when the process instance is created.
- $\forall a \in \lambda(t), \forall p \in C^{\rightarrow}(a) : \xi(t, p) = evaluated$. That is, when an activity completes normally, states of the predicates of the links leaving *a* become 'evaluated'. Their value is the value of evaluating the transition condition.
- $\forall a \in \delta(t)$ the status of links leaving a disabled activity is set to 'false' if they cross the boundary of a scope that has completed. The time this occurs is detailed in section 3.2.9 (scope navigation). The time *a* disables is denoted as $t_d(a)$.
- $\omega(i, a) = default \wedge (\forall p \in C^{\leftarrow}(a), \xi(i, p) = evaluated) \Rightarrow \xi(i, \phi(a)) = evaluated$. That is, the join condition gets evaluated once all of its constituting predicates are in state 'evaluated' and the activity is in the default state. For an activity *a*, we refer to this time as $t_j(a)$.
- $\forall w \in W, \omega(t, K(w)) = activated \Rightarrow \xi(t, \pi_1(w)) = evaluated$. The condition of a loop is evaluated once the activity corresponding to the loop in the control flow is activated. This condition will get re-evaluated at the time when an activity completes an iteration of the loop, as will be defined in section 3.2.8.
- If an activity is revived (due to a loop) then the state of related activities and predicates will be reset as will be described in section 3.2.8.
- In all other situations, a predicate is in state 'not-evaluated'.

The truth value of a predicate is 'unknown' until the predicate has been evaluated. The truth value is set to either true or false at the point in time that the predicate state becomes evaluated. We use $[\![p]\!]_t$ to refer to the truth value of a predicate *p* at time *t*. Therefore:

$$\xi(t,p) = \textit{not-evaluated} :\Leftrightarrow [\![p]\!]_t = ?$$

Additionally, even if all the values of incoming links of a disabled activity have been evaluated, its join condition is not evaluated: Its truth value is '?'.

$$\omega(t,a) = \textit{disabled} \wedge \{\forall e \in E^{\leftarrow}(a), \xi(t,e) = \textit{evaluated}\} \Rightarrow \xi(t,\phi(a)) = \textit{not-evaluated}$$

We use $[\![e]\!]_t$ to refer to the status of a link *e* at time *t*. The value will be either the result of evaluating the link's transition condition at time *t*, $[\![\pi_3(e)]\!]_t$, or false if the activity that is the source of the link is disabled.

$$[\![e]\!]_t := \begin{cases} ? & [\![\pi_3(e)]\!]_t = ? \wedge (\omega(t,\pi_1(e)) \neq \textit{disabled}) \\ \textit{false} & \omega(t,\pi_1(e)) = \textit{disabled} \\ [\![\pi_3(e)]\!]_t & \textit{otherwise} \end{cases}$$

We define the truth value of the join condition at a time *t* to be the result of evaluating its Boolean value after replacing each reference to a link by the status of the link at time *t*. For each activity *a* there is a natural number n_a such that:

$$\phi(a) = \bigvee_{1 \leq i \leq n_a} \bigwedge_{e_i \in E^{\leftarrow}(a)} e :\Rightarrow [\![\phi(a)]\!]_t = \bigvee_{1 \leq i \leq n_a} \bigwedge_{e_i \in E^{\leftarrow}(a)} [\![e]\!]_t$$

3.2.8 Navigation in Loops

The loop condition is evaluated when the activity $K(w)$, representing a loop w, activates. If the loop condition is true, the set of start activities of the corresponding loop are activated. If the condition is false, $K(w)$ completes, and all nested activities are set to state 'disabled' and the state of the predicates of these activities (join conditions and loop conditions) and the links (transition conditions) between them are set to 'evaluated'. The truth values of these predicates are set as follows: to '?', unknown, for transition and loop conditions, and to 'false' for link transition conditions. Recall from section 3.2.7 that a disabled activity's join condition is 'not-evaluated', regardless of the status of its incoming links. Navigation continues normally from $K(w)$.

An activity *a completes an iteration of a loop* w at time *t* if *a* is an activity in the loop and it is the last activity to complete in the loop, i.e. if the following conditions are met:

1. $a \in \pi_2(w)$
2. $a \in \lambda_t - \lambda_{t-1}$
3. $\forall b \in \pi_2(\omega), b \in \lambda_t \cup \delta_t$

When an activity completes an iteration of a loop, and the loop condition is true, then navigation will proceed from the activity that completed the loop to the loop start activities. This takes places in three time steps that we define to be consecutive:

- Consider *a* completes an iteration of the loop at time *t*.
- At *t+1* the state of all activities in the loop is set to 'default' and they are removed from the set of faulted activities $\rho(t)$ if they had been members of it. Also in this time step, the state of the predicates of these activities (join conditions and loop conditions) and the links

(transition conditions) between them are set to 'not-evaluated'. The truth values of these predicates returns to '?', unknown.
- At time *t+2*, the loop start activities get activated.

An activity *a completes a loop* w *at time t* if *a* completes an iteration of the loop and the condition of the loop evaluates to false, i.e. if the following conditions are met*:*
1. *a* completes an iteration of w *at time t*
2. $[\![\pi_1(w)]\!]_t = false$

The work of the loop, i.e. the implementation of $K(w)$, returns either when the loop is activated and the loop condition is false or when an activity completes the loop:
$\psi(K(w))$ returns at time $t :\Leftrightarrow$

$(\omega(K(w), t-1) = disabled \wedge \omega(K(w), t) = activated \wedge [\![\pi_1(w)]\!]_t = false) \vee$

$\exists a \in \pi_2(w) : a$ completes w at time t

3.2.9 Navigation in Scopes

Navigation in scopes is affected by the completion or faulting of the scope's activities. This section defines scope completion, when and whether to search for a fault handler, and how this search is performed.

Consider $N_G(a)$, the set of nodes in the graph of the hypergraph containing the activity, *a*. It contains, in other words, all the scopes containing *a*: $N_G(a) := \{n \in N_G | a \in A(n)\}$.

The immediate scope of an activity is the scope that is furthest from the root scope in *G* and still contains *a*. With *depth(G,n)* as the distance of *n* from the root of *G*, the immediately enclosing scope of *a*, $s'_0(a)$ is defined as:

$$s'_0(a) := \{n \in N_G(a) | \forall n_i \in N_G(a) - \{n\} : depth(G, n) > depth(G, n_i)\}$$

$s'_0(a)$ is well defined, i.e. $|s'_0(a)| = 1$: Since scopes are strictly nested, each activity has exactly one immediately enclosing scope.

An activity completes a scope *s* in one of two cases: Either it is the last activity in the body of the scope, or it is the last activity in the fault handler of the scope. Additionally, an activity that completes a scope *s* may also be completing one or more of *s*'s parent scopes.

Let *body(s)* be a function that returns the activities in the body of a scope, i.e. those not in the scope's fault handlers:

$$body(s) := A(s) - \bigcup_{f \in F(s)} A(f)$$

Let *scope_part(a,s)* be the partial function (where $\wp(N)$ denotes the power set of N):

$$scope\_part : N \times S \rightarrow \wp(N)$$

This function is defined for all pairs *(a,s)* such that $s \in N_G(a)$. It returns either all activities of the scope or only those in the body of the scope, depending on whether *a* belongs to the fault handlers or the body:

$$scope\_part(a,s) = \begin{cases} A(s) & a \in \bigcup_{f \in F(s)} A(f) \\ body(s)) & a \in body(s) \end{cases}$$

We say that *a completes* a scope *s* at time *t* iff all the following three conditions are met*:*

1. $a \in \lambda_t - \lambda_{t-1}$
2. $a \notin \{body(s) | s \in N_G(a), \exists f \in F(s) : f \in z(t-1)\}$
3. $\forall b \in scope\_part(a,s) : b \in \lambda_t \cup \delta_t$

The states of certain activities must be disabled and status of links set to false when a scope completes or an activity faults and a matching fault handler is found. The activity *a* that completes a scope may have completed several nested scopes at once. Therefore, we need to work from the largest scope (closest to the root) that *a* completes. The links to be set to false are collected in $\delta E(s,t)$, the dead-link list of a scope *s* at time *t.* The activities to be disabled are collected for each scope in a corresponding set $A_{disable}$. This section (including its subsections) explain the reasoning and steps for creating these sets, leading to the complete definition $A_{disable}$ in the subsection on fault handling.

We define the function *c(a,s,t*) which is true if *a* completes *s* at time *t*:

$$c(a,s,t) := \begin{cases} true & a \text{ completes } s \text{ at time } t \\ false & \text{otherwise} \end{cases}$$

Denote by t_{c_s} the time that the scope completed: $\exists a \in N : c(a,s,t) = true \Rightarrow t_{c_s} = t$.

Consider the following algorithm for a function *u(s,t)* that takes a scope and a time *t* as input and returns the set of all links that are unevaluated at time *t*, whose source has not completed and is in the scope, and whose target is outside the scope.

```
function u(s,t)
 U ← ∅
 for each a ∈ A(s) − λ_t
  for each e ∈ E→(a)
    if π2(e) ∉ A(s) ∧ ξ(t_cs, π3(e)) = not-evaluated
      U ← U ∪ {e}
 return U
```

We define $\delta E(s,t)$ as a set containing the links whose status will be set to false upon scope completion. This set contains all links unevaluated at scope completion time whose sources belong to the scope and whose targets do not. At any other time, this set is empty:

$$\delta E(s,t) := \begin{cases} u(s,t) & t = t_{c_s} \\ \emptyset & otherwise \end{cases}$$

Links whose sources have completed are expressly excluded so as not to set to false the status of the link of the activity that completed the scope.

Successful Scope Completion

A scope, *s*, is said to have *completed normally* if it completed without running any of its fault handlers.

$$s \text{ completed normally } :\Leftrightarrow \exists b \in s : c(b, s, t_{c_s}) = true \wedge b \notin \bigcup_{f \in F(s)} A(f)$$

Define a helper function *dflt_activities(A,t)* that takes a set of activities *A* and a time *t* as input and returns the set of all activities in the set that were in the default state at time *t-1*.

$$dflt\_activities(A, t) := \{a \in A | \omega(t-1, a) = default\}$$

$$s \text{ completed normally } \Rightarrow A_{1\_disable} := dflt\_activities(A(s), t_{c_s})$$

$$s \text{ completed normally} \Rightarrow \omega(t_{c_s}, a) = disabled, \text{ for } a \in A_{1\_disable}$$

If a scope completes because one of its fault handlers completed, we say it *completed abnormally*. In that case, all the activities in it have already been disabled or completed. If a scope does not complete normally or abnormally, i.e.: it is exited due to a fault handled by an ancestor scope or not at all, then we say the scope has been *exited*.

Throwing Faults and Finding Fault Handlers

An activity may throw a fault at any time after it starts and before the activity time-out limit. A faulted activity's state becomes disabled when the fault is thrown.

$$a \text{ faults at time } t \Rightarrow \omega(t, a) = disabled, a \in \rho(t)$$

An activity faults by returning from execution with a fault with name *n* instead of returning normally. If a fault occurs, a matching fault handler is searched for by walking iteratively up the scope tree looking for the first handler that matches. Activities that are already executing when a fault occurs are not interrupted. The navigation must wait until they return before it can run the found fault handler. The activities return either by completing or themselves throwing a fault. Such a fault, called a secondary fault, disables the activity that caused it but should not result in any fault handling because the scope that it is thrown in is already scheduled to handle a prior fault. A primary fault, on the other hand, is a fault that causes a search for a fault handler.

In order to start the search, we first define a function *parent* that retrieves the parent of a given scope. Every scope in a process has a parent, except for the root scope, therefore

$$parent : S - \{s_r\} \rightarrow S$$

The parent is found using the graph of the hypergraph:

$$parent(s) = \pi_2(e) \text{ where } e \in E_G : e = (s, s')$$

We define a function *m* that determines whether a fault handler matches a fault of a particular name, *n*:

$$m(n, f) := \begin{cases} true & n \in \pi_1(f) \\ false & \text{otherwise} \end{cases}$$

Next, we define a function *X* that returns the set containing the matching handler for a fault of a particular name in a particular scope.

$$X(n, s) := \{f \in F(s) | m(n, f) = true\}$$

There is always only zero or one matching fault handler, so $|X(n,s)| \in \{0,1\}$.

Next, we define the search for a matching fault handler. We define '$\perp$' to represent 'not found'. A helper function *g(s,a)* is defined that takes as input a current scope *s* and faulting activity *a* and returns the next scope to search for a fault handler. This scope is normally the parent of the current scope because the fault gets thrown up the scope hierarchy. If a fault occurred in a fault handler and was not caught in the scope hierarchy inside the fault handler, however, the search must skip the scope associated with the fault handler. To skip such scopes, we define the recursive function *g(s,a)*:

$$g : S \times N \rightarrow S \cup \{\perp\}$$

$$g(s,a) = \begin{cases} parent(s) & s \neq s_r \wedge a \notin \bigcup_{f \in F(parent(s))} A(f) \\ g(parent(s),a) & s \neq s_r \wedge a \in \bigcup_{f \in F(parent(s))} A(f) \\ \perp & otherwise \end{cases}$$

The first case occurs when a fault comes from an activity in the body of the current scope's parent, thus *g* returns the parent itself. In the second case the fault comes from the fault handler of the current scope's parent, thus *g* must skip the parent. The parent's parent should be returned by *g* instead; however, the parent's parent might also be in the fault handler of its parent and thus need to be skipped too. Thus, *g* is a recursive function. If *s* is the root scope, then there is no scope to search next.

The scope where the search handler lookup starts is defined by *s_0 (a)*. It is usually *s'$_0$ (a),* the immediately enclosing scope of *a*. However, *a* may be in a fault handler of *s'$_0$ (a),* so it and possibly some of its ancestor scopes as well, must be skipped. The scope *s_0 (a)* where this search starts is therefore defined as follows:

$$s_0(a) := \begin{cases} s'_0(a) & a \notin \bigcup_{f \in F(s'_0(a))} A(f) \\ g(s'_0(a)) & a \in \bigcup_{f \in F(s'_0(a))} A(f) \wedge s'_0(a) \neq s_r \\ \perp & \text{otherwise} \end{cases}$$

Note that *s_0 (a)* returns 'not found' only if *a* is in a fault handler of the root scope or if it is in a fault handler of *s'$_0$ (a)* and $g(s'_0(a))$ returns 'not found'.

We are now ready to define the map *y* that returns the matching fault handler of a fault named *n* thrown by an activity *a*. We define a map *y* such that:

$$y : M \times (S \cup \{\perp\}) \times N \rightarrow F \cup \{\perp\}$$

$$y(n,s,a) = \begin{cases} x \in X(n,s) & |X(n,s)| = 1 \wedge s \neq \perp \\ y(n,g(s,a),a) & |X(n,s)| = 0 \wedge s \notin \{s_r, \perp\} \\ \perp & otherwise \end{cases}$$

Observe that in *y, g* is used if $y(n,s,a)$ does not find the handler in *s* itself. Finally, define a function *fh(a,n)* that takes an activity *a* and a fault name *n* and returns a fault handler that can catch a fault named *n* if it is thrown by *a*:

$$fh(a,n) := y(n, s_0(a), a)$$

The behavior of the process upon the occurrence of a fault depends on whether or not a fault handler is found.

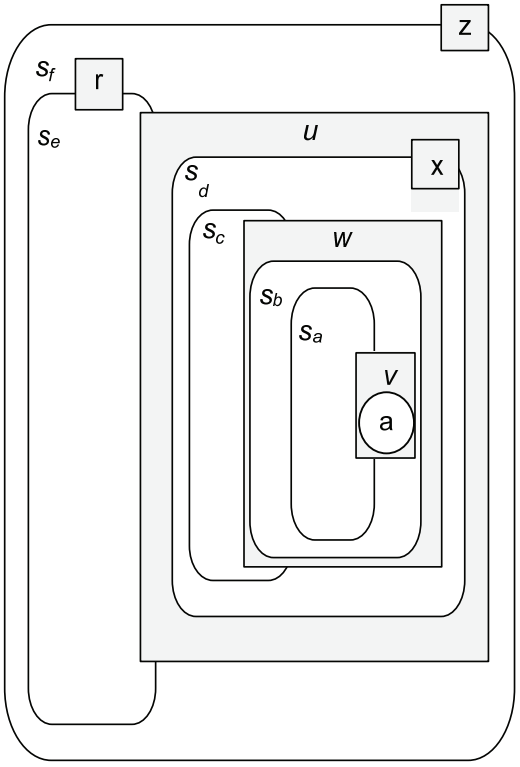
- If no fault handler is found, the process will be terminated.
- Otherwise, the following takes place at time t_f when the fault was thrown:
  - Disable all activities that are in the *default* state, that belong to $s_f = \pi_3(fh(a,n))$ but *not* to the activities of $fh(a,n)$ itself. In other words, disable at time t_f all activities in the set $A_{2\_disable}$ where:

$$A_{2\_disable} := dflt\_activities(A(s_f) - A(fh(a,n)), t_f)$$

$$\omega(t_f, a) = disabled, \text{ for } a \in A_{2\_disable}$$

  - Consider the time that s_f completes, i.e. when its fault handler completes. Then the dead-link list of the scope at this time, $\delta E(s_f, t_{c_s})$, will contain all links whose sources are these disabled activities and whose targets do not belong to s_f.

Figure 8 illustrates a fault handler search going up several levels of scope nesting. The rounded rectangles are scopes. The circle is the faulting activity *a*. The normal rectangles are fault handlers labeled with the names of the faults they can catch. Two examples are shown for searching for a matching fault handler. Notice how the search skips a scope when crossing a fault handler boundary.



Example: *a* throws fault *x*:
$s'_0(a) = s_a$ //immediately enclosing scope
$s_0(a) = s_b$ //*a* in fault handler of $s'_0(a)$, so
//start search from an ancestor scope
$y(x, s_0(a), a) = y(x, s_b, a)$
//s_b has no matching handler, so use *g*
//to find the next scope to search
$= y(x, g(s_b)), a) = y(x, g(parent(s_b)), a)$
//s_b is in s_c's fault handler, so *g* skips s_c
$= y(x, s_d, a)$
//s_d has a matching handler, f_x, so return it
$= f_x$

Example: *a* throws fault *z*:
$y(z, s_b, a) = y(z, s_d, a)$
//same as Example 1, but s_d has no matching
//handler,so use *g* to find the next scope
$= y(z, g(parent(s_d)), a)$
//s_d is in s_e's fault handler, so *g* skips s_e
$= y(x, s_f, a)$
//s_f has a matching handler, f_z, so return it
$= f_z$

Figure 8: Searching for a fault handler for a fault thrown by activity *a*

The function *p_faulted* takes an activity *a* and a time *t* and returns whether or not *a* threw a primary fault:

$$p\_faulted(a,t) := \begin{cases} true & a \in \rho_t - \rho_{t-1} \wedge a \notin \bigcup_{f \in z(t-1)} body(\pi_3(f)) \\ false & otherwise \end{cases}$$

The conditions for which this is true are summarized as: (1) the activity faulted at time *t*, and (2) the activity is not in the body of a scope that is waiting to handle a fault. In this case, a fault handler must be searched for.

The definition of $A_{disable}(s,t)$ is therefore as follows, combining $A_{1\_disable}$ in the first case and $A_{2\_disable}$ in the second where *n* is the name of the fault thrown by *b:*

$$A_{disable}(s,t) := \begin{cases} dflt\_activities(A(s),t) & \exists b \in A(s) : c(s,b,t) = true \\ dflt\_activities(A(s) - A(fh(b,n),t) & \exists b \in A(s) : \\ & \quad p\_faulted(b,t) = true \wedge s = \pi_3(fh(b,n)) \\ \emptyset & \text{otherwise} \end{cases}$$

In summary, if a scope completes then any activities still in the default state need to be disabled. If an activity faults and a matching fault handler is found, then the activities to be disabled are those in the default state in the fault handler's scope that are not in the fault handler itself. $A_{disable}(s,t)$ provides these activities. Note that $A_{disable}(s,t)$ is well defined because the cases are mutually exclusive: by definition, only one activity is allowed to complete or fault at time *t*.

Having defined the search for a fault handler, the disabled activity set and the dead-link list, it remains to define the waiting fault handlers *z(t)* and several helper functions that will be necessary in section 3.2.13 for defining the navigation step of this meta-model.

Consider another function $wait\_fh(a,t)$ that takes an activity *a* and a time *t* and returns true if *a* threw a primary fault at time *t* but there were still activities in the body of the scope that were running. In such a case, the scope must wait for these running activities to complete (or fault) before it can start a fault handler. Consider in the first case of the definition that *n* is the name of the fault thrown by *a*.

$$wait\_fh(a,t) := \begin{cases} true & p\_faulted(a,t) \wedge fh(b,n) \neq \perp \wedge \\ & \quad \exists b \in A(\pi_3(fh(b,n))) : \omega(b,t) = activated \\ false & otherwise \end{cases}$$

In order to determine whether the wait has ended and the fault handler can run, we define a function *end_wait(a,t)* that takes an activity *a* and a time *t* and returns true if the activity *a* completes or faults at time *t* such that it is the last activity to do so in the body of a scope that is waiting to handle a fault.

$$end\_wait : N \times \mathbb{N} \rightarrow \{true, false\}$$

$end\_wait(a,t)$ returns true if the following two conditions are met and false otherwise:

1. $a \in \rho_t - \rho_{t-1} \cup \lambda_t - \lambda_{t-1}$, and
2. $\exists s \in S : a \in body(s) \wedge \forall b \in body(s) : b \in \delta_t \cup \lambda_t \wedge \exists f \in F(s) : f \in z(t-1)$

We are now ready to formally define the map of waiting fault handlers, $z : \mathbb{N} \rightarrow \wp(F)$. Consider in the second and third cases below that $f_h = fh(a, n)$ where *n* is the name of the fault thrown by *a* at time *t* and that $F_{nested} = \{f \in F | A(f) \subset body(\pi_3(f_h)\}$.

$$z(t) := \begin{cases} \emptyset & t = 0 \\ z(t-1) \cup \{f_h\} - F_{nested} & \exists a \in N : wait\_fh(a,t) = true \\ z(t-1) - F_{nested} & \exists a \in N : \\ & \quad p\_faulted = true \wedge wait\_fh(a,t) = false \\ z(t-1) - \{f\} & (\nexists a \in N : p\_faulted = true) \wedge \\ & \quad \exists f \in F, a \in N''(f) : \\ & \quad\quad \omega(a, t-1) = default \wedge \omega(a,t) = activated \\ z(t-1) & \text{otherwise} \end{cases}$$

The set returned by *z(t)* is null when the process instance starts (case 1). If a primary fault occurred at time t and f_h is associated to a scope that needs to wait for activities in its body to complete, then f_h is added to $z(t-1)$ (case 2). Whether or not the scope needs to wait, the occurrence of a primary fault results in any handlers nested in the body of the scope of f_h to be removed (case 2 and case 3). The reason is that they can no longer run due a fault in an ancestor scope. Finally, a fault handler is removed from $z(t-1)$ if its activities activate, because it is no longer waiting to run (case 4). For all other cases, $z(t)$ remains as it was in the previous time step (case 5).

Illustrating Scope Completion

The process in Figure 9 is used to illustrate scope completion. Consider that activity *c* belongs to the body of the scope, and activity *x* belongs to a fault handler of s_3. In this process fragment, *c* will either complete or throw a fault of the type the scope's fault handler can catch.

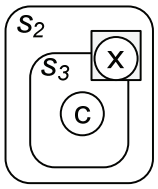


Figure 9: Illustrating scope completion possibilities

We get the following cases:

1. If *c* completes, it causes its immediate scope, s_3, and the parent scope, s_2, to complete normally.
2. If *c* faults, *x* will activate. When *x* completes, it will cause s_3 to complete abnormally and s_2 to complete normally.
3. If *c* and then *x* fault, then both scopes are exited.

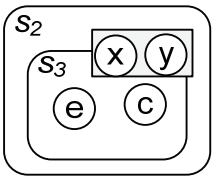


Figure 10: Additional scope completion possibilities

Next, consider the more complex cases in Figure 10:

- Consider *c* faulted at time *t*: $\omega(t,e) = activated, c \in \rho_t - \rho_{t-1}$. Assume the fault handler of s_3 can catch this fault, so s_3 waits for *e* to complete and then *x* and *y* activate.
- Now modify that case by assuming that *e* faults at a later time than *c*. The fault from *e* disables the activity but must not cause any fault handler lookup. That is, beyond disabling the faulting activity *e*, this secondary fault is ignored.
- Finally, modify the first case so instead of completing, *y* throws another fault while *x* is still active. Then, one would get another fault handler lookup starting from s_2.

3.2.10 Activated Activities

An activity that is a start activity of the process *becomes activated* at the time of process instantiation, t_0. Therefore,

$$a \in N' \Leftrightarrow \omega(0,a) = activated$$

$$a \in N - N' \Leftrightarrow \omega(0,a) = default$$

From this point on, navigation happens as a result of a 'navigation step' defined in section 3.2.13.

An activity, $a \in N - (N' \cup N'' \cup N''')$, that is not a start activity of the process, of a fault handler, or of a loop is activated when the predicates on its incoming links have been evaluated and its join condition is true. Therefore,

$a \in N - (N' \cup N'' \cup N''')$ becomes activated at time $t \Leftrightarrow$
1. $\omega(a, t-1) \neq disabled$, and
2. $\exists (b,a,p) \in E : \xi(t,p) = evaluated \wedge \xi(t-1,p) = not\text{-}evaluated$, and
3. $[\![\phi(a)]\!]_t = true$

An activity, $a \in N''$, that is a fault handler start activity becomes activated when the handler catches a matching fault and all activities in the scope have completed or when the handler is waiting to handle a fault and the last activity in the body of the associated completes or faults. Then,

$a \in N''$ becomes activated at time $t \Leftrightarrow$
1. $\exists b \in N : p\_faulted(b,t) = true \wedge wait\_fh = false \wedge fh(b,n) \neq \perp \wedge a \in A(fh(b,n))$, or
2. $\exists c \in N, f_z \in z(t-1) : a \in A(f_z) \wedge end\_wait(c,t) = true, c \in body(\pi_3(f_z))$

The activity state map must satisfy the following property for each $t \in \mathbb{N}$. However, the reverse does not hold. An activity stays activated for some time - until it either completes, or faults (becomes disabled).

$$a \in N \text{ becomes activated at time } t \Rightarrow \omega(t,a) = activated$$

An activity $a \in N'''$, which is a loop start activity becomes activated either when the loop condition evaluates to true and the activity corresponding to the loop gets activated or a two time steps after a loop iteration completes and it is ready to iterate again:

$a \in N'''$ becomes activated at time $t \Leftrightarrow$
$\omega(k(w), t) = activated \wedge$
$((\omega(k(w), t-1) = disabled \wedge [\![\pi_1(w)]\!]_t = true) \vee$
$(\exists a \in \pi_2(w) : a \text{ completes an iteration of } w \text{ at time } t-2 \wedge [\![\pi_1(w)]\!]_{t-2} = true))$

3.2.11 Completed Activities

An activity completes when its implementation returns. Therefore, an activity completes at time t,
$a \in \lambda_t - \lambda_{t-1} :\Leftrightarrow$
1. $\exists j \in \mathbb{N} : j < t \wedge \omega(j, a) = activated$, and
2. $\exists k \in \mathbb{N} : j < k \leq t \wedge \psi(a)$ *returned at time* k

The reason that $k \leq t$ and not $k = t$ is that we do not allow more than one activity to complete at the same time, as noted earlier. If the implementations of more than one activity return at the same, the corresponding activities will be completed one at a time.

3.2.12 Disabled Activities

An activity gets disabled either if it faulted itself or if an enclosing scope disabled it. The latter case occurs if *a* is in a scope that ended before *a* could run either because a fault was raised by another activity in the scope or because *a* is in a fault handler of a scope that completed successfully, or *a* is in a loop whose condition in the first iteration has evaluated to false. Therefore,

$a \in N$ becomes disabled at time $t :\Leftrightarrow$
1. $\exists j \in \mathbb{N} : j \leq t \wedge \omega(j, a) = default \wedge a \in \rho_t - \rho_{t-1}$, or
2. $a \in A_{disable}(s, t)$, where $s \in N_G(s)$, or
3. $\exists w \in W : a \in \pi_2(w) \wedge$
$\omega(K(w), t-1) = disabled \wedge \omega(K(w), t) = activated \wedge [\![\pi_1(w)]\!]_t = false$

3.2.13 Computing Actual Successors

As in [LERO00], we define the navigation by defining the 'actual successors' of an activity. These are the actual successors of an activity at the point in time, *t*, when the activity completes or faults. The actual manner in which the successors are determined differs here from [LERO00]. The actual successors of *a* at time *t* is the set of successor activities that become activated at or after time *t*. The only case when they will become active after time *t* is if they are the start activities of a loop that is on an iteration after its first. Then, they will activate at *t+2*. Recall that any activity *a* whose join condition evaluates to false throws a join fault. A navigation step only occurs when an activity either completes or faults. Therefore, we focus on these two cases.

Before providing the definitions of actual successors, we first define $s''(a, t)$, the scope that *a* completes that is closest to the root scope s_r:

$$s''(a, t) := s \in S \text{ where } c(a, s, t) = true \wedge (s = s_r \vee c(a, parent(s), t) = false)$$

Note that an activity that completes one or more scopes must complete at least its immediate scope $s'_0(a)$.

Now let $a \in N$ be an activity whose state becomes completed at time t, i.e. $a \in \lambda_t - \lambda_{t-1}$. Let:

$$M_1(a) := \{b \in E^{\rightarrow}(a) | \phi(b) = true\}$$

$M_1(a)$ is the set of all successors of *a* having a join condition that evaluates to true. Recall that if a target activity has been disabled due to a fault, the join condition will evaluate to '?' regardless of the incoming links.

$$M_2(a) := \begin{cases} \{\pi_2(e) | e \in \delta E(s''(a,t)) \wedge \phi(\pi_2(e)) = true\} & c(a, s'_0(a), t) = true \\ \emptyset & otherwise \end{cases}$$

$M_2(a)$ is the set of the targets of all the links in the dead-link-list of a scope whose join condition is true, for *a* being the last activity in either the normal execution of a scope, or one of its fault handlers. Recall that by the time *M₂(a)* is determined, the value of the predicates of all the dead-links in $\delta E(s''(a,t))$ are false.

Now, let $a \in N$ be an activity that faults or whose state becomes completed at time t, i.e. $a \in \lambda_t - \lambda_{t-1} \cup \rho_t - \rho_{t-1}$.

$$M_3(a) := \begin{cases} N''(f_z) & end\_wait(a,t) = true \wedge \exists f_z \in z(t-1) : a \in body(\pi_3(f_z)) \\ \emptyset & otherwise \end{cases}$$

$M_3(a)$ is the set of start activities of a fault handler that is waiting to handle a fault, and for which *a* is the last activity in the fault handler's body to complete or fault thus signaling the end of the wait. There will never be more than one such fault handler, because only one handler in a scope matches a fault and due to the definition of *z(t)* whereby handlers that can no longer run are removed.

Now, let *a* be an activity that faults at time *t* for which $p\_faulted(a,t)$ is true. That is, $a \in \rho_t - \rho_{t-1}$ and *a* is not in a scope that is already waiting to handle a fault. Let *n* be the name of the fault *a* threw at time *t* and $f_h = y(n, s'_0(a), a))$, then

$$M_4(a) := \begin{cases} N''(f_h) & p\_faulted(a,t) = true \wedge wait\_fh(a,t) = false \wedge f_h \neq \perp \\ \emptyset & otherwise \end{cases}$$

which is the set of start activities of the matching fault handler.

Next, consider the start activities of a loop, whose activation was described in section 3.2.10. The first time a loop iterates, the implementation of $K(w)$ activates the loop start activities if the loop condition is true. The activities are starting not because they are actual successors of a previous activity but due to the activation of $K(w)$. These start activities may be activated again after a loop iteration completed. Thus, the successors for an activity that completes an iteration of the loop at time *t* and with a true loop condition are again the loop's start activities:

$$M_5(a) := N'''(w)$$

The set $\sum_t(a)$ of *t-actual successors* of an activity *a* is therefore defined as follows:

$$\sum_t(a) := \begin{cases} M_1(a) \cup M_2(a) \cup M_3(a) & a \in \lambda_t - \lambda_{t-1} \wedge \\ & \quad \nexists w \in W : a \text{ completes an iteration of } w \text{ at } t \\ M_3(a) \cup M_4(a) & a \in \rho_t - \rho_{t-1} \\ M_5(a) & \exists w \in W : a \text{ completes an iteration of } w \text{ at } t \wedge \\ & \quad [\![\pi_1(w)]\!]_t = true \wedge \omega(K(w), t) = activated \\ \emptyset & \text{otherwise} \end{cases}$$

In the second case, either *M3(a)* or *M4(a)* will always be empty depending on whether the fault is a primary fault and whether a scope is waiting on *a* before it can run its fault handler. Also, the third case does not need the activities of the first case because although it has an activity that is completing, an activity that completes a loop iteration cannot have outgoing links, cannot be the last activity in a scope that has unfollowed outgoing links, and cannot be the last activity to complete before a fault handler can run.

The process ends when all its activities have been completed or disabled. Note that activities in a compensation handler belong to their own process and as such are not included in this completion check. Compensation will be covered in detail in section 3.6.

3.3 Illustrating the Navigation Algorithm

A sample process is shown in Figure 11. It is used to illustrate the navigation.

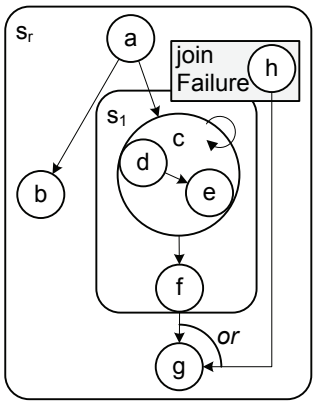


Figure 11: A sample process

The root scope in Figure 11 contains several activities including another scope s_1. Scope s_1 contains a loop *c* and a fault handler for the joinFailure fault containing one activity, *h*.

First, consider the case that the process does not fault, and the loop iterates once:

t=0	*a* becomes active because it belongs to *N'*
t=1	*a* completes so $\lambda(1) = \{a\}$. Assuming that the links to *b* and *c* evaluate to true, then the actual successors of *a* are: $\sum_1(a) = M_1(a) \cup M_2(a) \cup M_3(a) = M_1(a) = \{b, c\}$ *b* and *c* become activated. The loop condition is evaluated. Assume it is true. *d* becomes activated, because $N'''(c) = \{d\}$
t=2	*b* completes so $\lambda(2) = \{a, b\}$, and

	$$\sum_{2}(b) = M_1(b) \cup M_2(b) \cup M_3(b) = \emptyset$$
t=3	*d* completes so $\lambda(3) = \{a, b, d\}$. Assuming the link to *e* evaluates to true, then the actual successors are: $$\sum_{3}(d) = M_1(d) \cup M_2(d) \cup M_3(d) = M_1(d) = \{e\}$$ *e* becomes active
t=4	*e* completes so $\lambda(4) = \{a, b, d, e\}$ *e* completes an iteration of c. Assuming that the condition of the loop evaluates to true, then the actual successors are: $$\sum_{4}(e) = M_5(e) = N'''(e) = \{d\}$$
t=5	The states of activities and predicates inside the loop are reset. Among other things, $\omega(d, 5) = \omega(e, 5) = default$ and thus $\lambda(5) = \{a, b\}$
t=6	*d* becomes active
t=7	The same as t=3
t=8	*e* completes so $\lambda(8) = \{a, b, d, e\}$ Assuming the condition of the loop evaluates to false, then the completion of *e* causes *c* to return. As two activities cannot complete at the same time, *c* completes in the next time step
t=9	*c* completes, so $\lambda(9) = \{a, b, d, e, c\}$ Assuming that the status of the link to *f* is true, then $$\sum_{8}(c) = M_1(c) \cup M_2(c) \cup M_3(c) = M_1(c) = \{f\}$$ *f* becomes active
t=10	*f* completes, which in turn completes the scope s_1. $$A_{disable}(s_1, 10) = \{h\}$$ $$\delta E(s_1, 10) = \{(h, g, p_{hg})\}$$ so : $$\delta(10) = \{h\}$$ $$[\![(h, g, p_{hg}]\!]_{t=10} = false$$ Assuming the status of the link between *f* and *g* evaluates to true, then $$\sum_{9}(f) = M_1(f) \cup M_2(f) \cup M_3(f) = M_1(f) = \{g\}$$
t=11	*g* completes. This is turn completes the root scope.

	$$A_{disable}(s_r, 11) = \emptyset$$ $$\delta E(s_r, 11) = \emptyset$$ $$\delta(11) = \{h\}$$ $$\lambda(11) = \{a, b, d, e, c, f, g\}$$ The process completes because all activities are either completed or disabled.

Now, consider a case where *c* throws a join failure fault. This would occur if the link from *a* to *c* evaluated to false:

t=1	*a* completes: $\lambda(1) = \{a\}$. Assume that the status of the link to *b* evaluates to true, but that of the link to *c* is false. Then the actual successors of a are: $$\sum_1(a) = M_1(a) \cup M_2(a) \cup M_3(a) = M_1(a) = \{b\}$$ *b* becomes active.
t=2	*c* throws a joinFailure and gets disabled. The fault handler look-up finds the fault handler of *s1*. $$\rho(2) = \{c\}$$ $$A_{disable}(s_1, 2) = \{d, e, f\}$$ $$\delta E(s_1, 2) = \{(f, g, p_{fg})\}$$ Thus, the activities in $A_{disable}$ are disabled and the status of links in δE set to false. $$\delta_{t=2} = \{c, d, e, f\}$$ $$[\![(f, g, p_{fg})]\!]_{t=2} = false$$ This is a primary fault and no other activities in the scope are running, therefore $p\_faulted(c, 2) = true, wait\_fh(c, 2) = false$. Thus the waiting fault handlers are unchanged and the successors are the start activities of the fault handler of s_1. $$z(2) = z(1) = z(0) = \emptyset$$ $$\sum_2(c) = M_3(c) \cup M_4(c) = M_4(c) = \{h\}$$ *h* becomes active
t=3	The same as t=2 in Case 1 (complete *b*)
t=4	*h* completes. Assuming the link from *h* to *g* evaluates to true, then $$\sum_4(h) = M_1(h) \cup M_2(h) \cup M_3(h) = M_1(h) = \{g\}$$

t=5	*g* completes. This is turn completes the root scope. $$A_{disable}(s_r, 5) = \emptyset$$ $$\delta E(s_r, 5) = \emptyset$$ $$\delta(5) = \{c, d, e, f\}$$ $$\lambda(5) = \{a, b, h, g\}$$ The process completes because all activities are either completed or disabled.

This concludes the description of navigation in the meta-model.

3.3.1 Pattern for Modeling Links to and from a Scope

It is often necessary to model the scope itself as the target or source of control links. To accommodate this requirement, which appears in BPEL, we provide a pattern of activities and links in the meta-model. It is illustrated in Figure 12: links to the scope are shown entering the activity α and links leaving the scope are shown leaving the activity β.

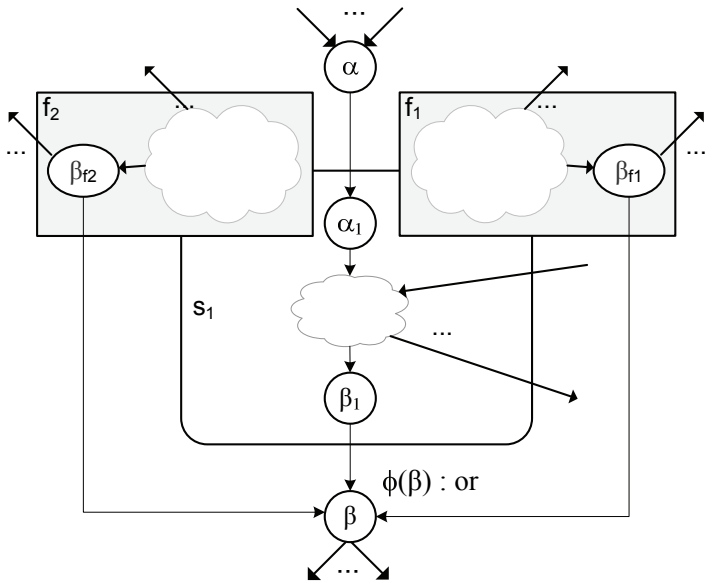


Figure 12: Pattern for linking to and from a scope

A start activity α is created for each scope that needs to be the target of links: α will be the actual target in the meta-model of any links to the scope. An end activity β is created for each scope that needs to be the source of links: β will be the actual source in the meta-model of any links from the scope. If *s* is not the root scope, α and β belong to the set of activities of the parent scope of *s*. Otherwise, they belong to the set of activities of *s*.

Additionally for this pattern, the activities in the body of the scope must start at a single activity α_1 and end at a single activity β_1, such that for each activity *a* that belongs to the scope but not to the scope's fault handlers there exists at least one path from α_1 to β_1 that contains *a*. These two activities, α_1 and β_1, belong to the activities of *s*. It is allowed that $\alpha_1 = \beta_1$. Additionally, they are connected to α and β:

$$\{\alpha_1, \beta_1\} \subset A(s)$$

$$\{(\alpha, \alpha_1, true), (\beta, \beta_1, true)\} \subset E$$

The activities α and β are used to isolate the activities in the body of the scope from the scope's actual beginning and end in the presence of links. Consider the join condition: if the join condition of an activity evaluates to false, it must throw a join failure to its parent scope. Now consider a join failure thrown due to the join condition on the links for which the scope itself is the target. To behave correctly, it follows that the failure must be thrown to the parent scope of *s*. In this pattern, the join condition is therefore placed on the activity α and α is placed in the parent scope of *s*.

Navigation out of a scope occurs either because a fault handler completed successfully, or because the activities in the body of the scope completed. Therefore, β_1 denotes the end of the scope body, while β denotes the real end of the scope: It synchronizes links coming from either the handler or the body.

The activity β is in the parent scope of *s* and not in *s* itself because when the scope completes, links that have not yet been fired and that cross from inside the scope to its outside fire with a false value. However, the links having the scope itself as the source fire by evaluating their transition conditions. Therefore,

$$\{\alpha, \beta\} \subset A(parent(s))$$

If a fault handler is defined on a scope *s* that is the source of links, there also needs to be a path from the activities in the handler to β. An end activity β_{f_i} is defined for each of *s*'s fault handlers. For each activity *a* belonging to the activities of the fault handler $A(f)$, there must exist a path from *a*, through β_{f_i}. All links leaving β_{f_i} must have targets outside the scope. Therefore:

$$a \in \{\pi_2(e) | e \in E^{\rightarrow}(\beta_{f_i})\} :\Rightarrow a \notin A(\pi_3(f_i))$$

The following link is added for each fault handler on such a scope:

$$(\beta_{f_i}, \beta, true) \in E$$

The implementations for all these added start and end activities are no-operations.

$$\psi(\alpha) = \psi(\alpha_1) = \psi(\beta) = \psi(\beta_1) = \psi(\beta_{f_i}) = no\_op$$

Finally, the join conditions of the end activities are the inclusive disjunction of the status of their incoming links:

$$\phi(\beta) = \bigvee_{e \in E^{\leftarrow}(\beta)} e$$

$$\phi(\beta_{f_i}) = \bigvee_{e \in E^{\leftarrow}(\beta_{f_i})} e$$

3.4 Representing BPEL Processes Using the Process Meta-model.

We have claimed that the fault handling mechanism is what enables BPEL processes to combine flat-graph and structured process modeling techniques. We have provided a graph-based meta-model that incorporates fault handling as a basic navigational technique. Therefore, we can now map BPEL constructs into this meta-model. The mapping is faithful except for default compensation order, for which we define a new and simplified approach in section 3.6.

3.4.1 Preprocessing the BPEL Process

A process using the meta-model defined in this chapter is created from a BPEL process. In order to do so, any syntactic shortcuts in the process must be replaced with their full version. BPEL defines two such syntactic shortcuts, both involving scopes: (1) An activity with the attribute suppressJoinFailure=”yes” is defined to be a shortcut for wrapping that activity in a scope with a fault handler that has an empty activity (no-op) and catches the join failure. (2) An invoke activity may directly define fault and compensation handlers, in which case there is an implicit scope surrounding the invoke activity on which these handlers are defined. Before mapping a BPEL process, these shortcuts are removed by making the implicit scopes in both these cases explicit.

Additionally, some switch activities may need to be modified. This modification results in a new switch activity we call ‘$switch_{mod}$’. ‘

Defining $switch_{mod}$

A switch activity is an activity containing ordered, conditional branches, each containing exactly one (primitive or structured) activity. The modification, $switch_{mod}$, is nearly identical to the original switch except for modifying any branches where the activity in a branch in the switch is not a scope that has a fault handler for the join failure fault. Note that this check is performed after removing scope shortcuts as just described above. For such branches, two cases are considered:

- Case 1: There is no link whose source is outside the branch and whose target is in the branch and the join failure is not suppressed by any activity between the target and the activity of the branch.
- Case 2: There is such a link.

For case 1, $switch_{mod}$ adds a new scope whose activity is the activity of the branch and a fault handler for the join failure containing an empty activity. This scope becomes the activity of the branch of $switch_{mod.}$

For case 2, two new scopes are added as shown in Figure 13. The switch on the right is changed in to the switch on the left. Consider that the top-most case’s activity does not suppress the join failure, but the other two do so. The labels on the far right will be used in the mapping description below.

The semantics of a switch are that the branch conditions are evaluated in order. The activity in the first branch to evaluate to true is activated and the others are disabled. The reason for creating the extra scopes is that the mapping will create links to the activities in each branch and use a join failure handler to disable activities in the failing branches. If the change for case 1 was done for case 2 as well, then the result would suppress join failures that the designer meant to surface: if *case a* in Figure 13 is true but the activity in the branch throws a

join failure due to the two incoming links, then the join failure would be caught in the branch - which changes the intended behavior.

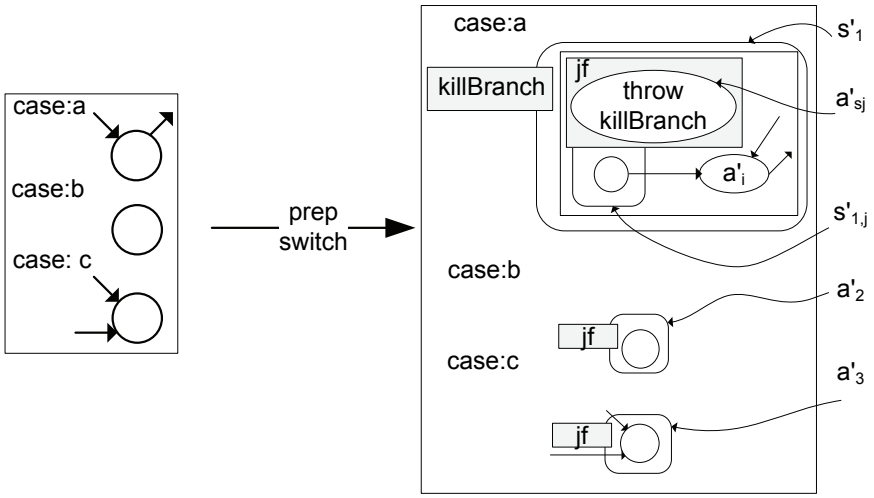


Figure 13: Preparing a switch, where the uppermost case's activity does not catch join failure

The change, in detail is as follows: Consider *b'i* to be the activity of the i[th] branch of 'switch$_{mod}$'. Consider *a'i* to be the activity of the i-th branch in the original switch. Thus, for case 2, $b'_i = s'_i$ where:

- *s'i* is a scope with a fault handler for a new fault named, '*killBranch*' and whose parent scope is the scope of *a'i*
  - The fault handler contains an empty activity.
  - The body of *s'i* contains a flow activity that itself contains:
    - The activity *a'i* of the original switch's branch
    - Another scope *s'j*
- The scope *s'j* contains:
  - An empty activity, *a'sj*, in the body of *s'j.*
  - A fault handler that handles the join failure fault. The activity in the fault handler is a throw activity that throws the 'killBranch' fault.
  - A link, l_i=(*a'sj* ,*a'i*,,true) .
- The join condition $\phi'_0(a'_i)$ of *a'i* is changed to depend on l_i, so $\phi'(a'_i) := \phi'_0(a'_i) \wedge l_i$.

3.4.2 Mapping Framework

We define a map *V* that maps from a BPEL Process, *P',* to a process *P* in the meta-model:

$$V : P' \mapsto P$$

Let *A'* be the set of activities, whether primitive or structured, of the BPEL process being mapped. Where,

Notation	*Definition*
V(a')	mapping of a BPEL activity *a'* into the corresponding parts of the process *P*
$A_{V(a')}$	set of activities created in *P* from mapping the activity *a'*
$E_{V(a')}$	the links created in *P* from mapping the activity *a'*

Table 2: Notation for mapping from a BPEL process to the process meta-model

The mapping of an activity will depend on the type of the activity, therefore we define:

$$A_V(a') := \begin{cases} \{a\} & a' \text{ is a primitive activity} \\ A_{V\,scope}(a') & a' \text{ is a scope} \\ A_{V\,sequence}(a') & a' \text{ is a sequence} \\ A_{V\,flow}(a') & a' \text{ is a flow} \\ A_{V\,switch}(a') & a' \text{ is a switch} \\ A_{V\,pick}(a') & a' \text{ is a pick} \\ A_{V\,while}(a') & a' \text{ is a while} \end{cases}$$

The implicit links in BPEL that result in explicit links in P will also depend on the type of activity, therefore we also have:

$$E_V(a') := \begin{cases} \emptyset & a' \text{ is a primitive activity} \\ E_{V\,scope}(a') & a' \text{ is a scope} \\ E_{V\,sequence}(a') & a' \text{ is a sequence} \\ E_{V\,flow}(a') & a' \text{ is a flow} \\ E_{V\,switch}(a') & a' \text{ is a switch} \\ E_{V\,pick}(a') & a' \text{ is a pick} \\ E_{V\,while}(a') & a' \text{ is a while} \end{cases}$$

The BPEL process itself behaves like a scope, therefore we map that first to the root scope, s_r. We initialize the following sets, where a'_1 is the process itself treated as a scope:

$$N \leftarrow A_{V(a'_1)}$$

$$E \leftarrow E_{V(a'_1)}$$

$$F \leftarrow \emptyset$$

$$S \leftarrow \{s_r\}$$

$$H \leftarrow (N, \{A(s_r))\}$$

$$G \leftarrow (\{s_r\}, \emptyset)$$

$$W \leftarrow \emptyset$$

$$C \leftarrow \emptyset$$

Every condition created in the mapping is added to *C*. This addition will not be explicitly shown to ease readability. Mapping one BPEL activity may result in several activities *P*, therefore even though a BPEL link has the same structure as a link in *P*, the source and target activities may be different. A BPEL activity will result in a start and end activity when mapped, as shown in Figure 14.

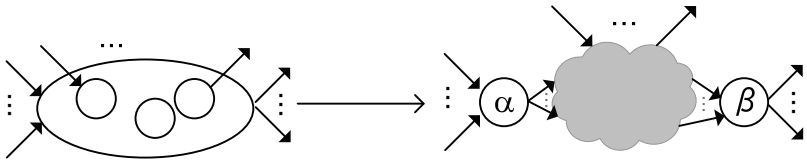


Figure 14: Mapping a BPEL compound activity

The start activity becomes the target of the original activity's incoming links. The end activity becomes the source of the original activity's outgoing links. We call these $\alpha V(a')$ and $\beta V(a')$, respectively. Unless specified otherwise:

$$\psi(\alpha V(a')) := \psi(\beta V(a')) := no\text{-}op.$$

After all the BPEL activities have been mapped, all the explicit links of the BPEL process are mapped. We define a map $L : L' \rightarrow E$ where *L'* are the explicit links of *P'*:

$$L((x, y, p_i)) := (\beta V(x), \alpha V(y), p_i)$$

L may create multiple links with the same source activity and target activity, which must therefore be merged as follows: When adding $e_j = (\beta V(x), \alpha V(y), p_j))$ to E, if another link is found such that $\exists e_j, e_k \in E, j \neq k \wedge \pi_1(e_j) = \pi_1(e_k) \wedge \pi_2(e_j) = \pi_2(e_k)$, then replace both with $e_h = (\beta V(x), \alpha V(y), p_i \wedge p_j)$. The edges thus created are then added to E.

3.4.3 Primitive Activities

A primitive BPEL activity maps directly into a single activity $a \in N$. Additionally,

$$a' \text{ is primitive} :\Rightarrow \alpha V(a') = \beta V(a') = a$$

3.4.4 Structured Activities

Structured activities are deconstructed to populate the meta-model. BPEL 1.1 defines six kinds of structured activities: *scope, flow, sequence, pick, switch,* and *while.* BPEL structured activities introduce additional implicit control dependencies that result in links in *P*. The mapping of each structured activity focuses on the activities and the control dependencies introduced by the structured activity itself: The mapping of all explicit links in a BPEL process is handled by the link mapping *L*. Additionally, any links leaving or entering the structured activity are handled by the link mapping and/or the mapping of its parent structured activity (if any).

Scope

A BPEL scope's function and structure are similar to those of a scope in the meta-model. However, a BPEL scope behaves as an activity: it only allows activities within it to start when it is reached in the control flow and may be the source and target of links. Consider a BPEL scope *a'*. By definition, it has exactly one activity in its body. We use the following notation in this section to refer to different parts of the BPEL scope:

Notation	*Definition*
$a'_{1\text{-}scope}$	The main activity in the body of a BPEL scope *a'*
F'(a')	The set of fault handlers on a BPEL scope *a'*
n'	The fault name of a BPEL fault handler
a'_{fi}	A BPEL activity defining a fault handler's work

Table 3: Notation used when mapping a BPEL scope

We use the pattern for modeling scopes that can be the sources and targets of links when mapping BPEL scopes. Consider $\alpha, \alpha_1, \beta_1, \beta$, as defined in that pattern. The start and end activities are created and connected to the mapping of the main activity in the scope:

$$A_{V\,scope(a')} := \{\alpha, \alpha_1, \beta_1, \beta\} \cup A_{V(a'_{1\_scope})} \cup \bigcup_{f_i \in F'(a')} (A_{V(a'_{f_i})} \cup \{\beta_{f_i}\})$$

$$E_{1\_scope}(a') := \{(\alpha, \alpha_1, true), (\alpha_1, \alpha V(a'_{1\_scope}), true), (\beta V(a'_{1\_scope}), \beta_1, true), (\beta_1, \beta, true)\}$$

The join condition of β_1 is simply the status of its incoming link:

$$\phi(\beta_1) := (\beta V(a'_{1\_scope}), \beta_1, true)$$

We create a scope *s* in *P* with the following activities:

$$A(s) = \begin{cases} A_{V(a')} - \{\alpha, \beta\} & a' = process \\ A_{V(a')} & otherwise \end{cases}$$

We add a hyperedge to the hypergraph to represent the new scope, and link it to the other scopes in P by adding an edge in the graph of the hypergraph.

$$\pi_2(H) \leftarrow \pi_2(H) \cup \{A(s)\}$$

$$\pi_1(G) \leftarrow \pi_1(G) \cup \{s\}$$

$$S \leftarrow S \cup \{s\}$$

Next, the scope is linked to its parent in the graph of the hypergraph by adding a corresponding edge to *G*. Consider *scope(a')* to be a function that returns the BPEL parent scope of the scope *a'*. There is exactly one parent for each scope other than the process itself, because BPEL scopes are strictly nested. Let p_s be the scope in *P* created from mapping the BPEL scope returned by *scope(a')*. Consider $e_G(s)$ to be the edge in *G* whose source is the newly added scope *s*, then:

$$\pi_2(G) \leftarrow \pi_2(G) \cup e_G(s)$$

$$e_G(s) = \begin{cases} \{(n_G(s), n_G(p\_s)\} & a' \neq process \\ \emptyset & otherwise \end{cases}$$

If the *a'* is not the process itself, then α and β are added to the activities of its parent scope in *P*. Therefore $e_H(p\_s)$, the hyperedge in the hypergraph corresponding to the scope of *a'* is modified as follows:

$$a' \neq process \Rightarrow e_H(p\_s) \leftarrow e_H(p\_s) \cup \{\alpha, \beta\}$$

For each fault handler f'_i, with a named fault *n'* and an activity *a'*$_{fi}$, a corresponding fault handler, f_i, is created. If the fault handler is a BPEL *catchall*, the value of *n'* is set to '*':

$$f_i = (n', (A_{V(a'_{f_i})} \cup \{\beta_{f_i}\}), s)$$

$$F \leftarrow F \cup \{f_i\}$$

$$n' \notin M \Rightarrow M \leftarrow M \cup \{n'\}$$

For each fault handler, the edge linking it to the end activity of the scope is collected in a set $E_{2\_scope}(a')$:

$$E_{2\_scope(a')} \leftarrow E_{2\_scope}(a') \cup \{(\beta V(\alpha'_{f_i}), \beta_{f_i}, true), (\beta_{f_i}, \beta, true)\}$$

The scope completes if either its body or one of its fault handlers completes. Therefore, the join condition of the last activity of the scope is the disjunction of the incoming links:

$$\phi(\beta) = \bigvee_{e \in E^{\leftarrow}(\beta)} e$$

$$\phi(\beta_{f_i}) = \bigvee_{e \in E^{\leftarrow}(\beta_{f_i})} e$$

The set of links created from mapping a BPEL scope is therefore:

$$E_{V\,scope(a')} := E_{1\_scope}(a') \cup E_{2\_scope}(a') \cup E_{V(a'_{1\_scope})} \cup \bigcup_{f_i \in F(a')} E_{V(a'_{f_i})}$$

Consider the 'modify-catch-all-faultname' function:

```
function modify-catchall-faultname(P)
 M ← M − {∗}
 for each s ∈ S
   for each f ∈ F(s)
     if π1(f) = {∗}
       π1(f) = M − M(s)
```

After all the activities are mapped, it is called to remove the '*' added by mapping BPEL's catch-all handler from the set of fault names *M*. It also modifies the set of fault handler names for such a handler to be *M* minus all faults names explicitly named by fault handlers in this scope:

Flow

A BPEL flow activity contains :

- A set of other BPEL activities, $A'_{flow} = \{a'_1, ..., a'_n\}$, and
- A set of explicit links $L' = \{l_1, ..., l_n\}$.

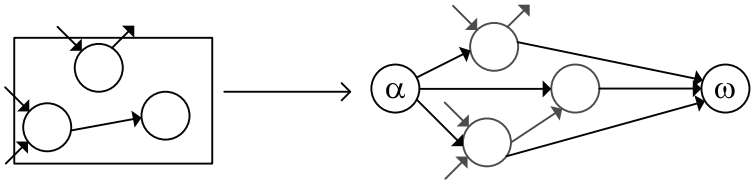


Figure 15: Mapping a BPEL flow

The activities are simply the activities created from mapping the activities in A'_{flow}, and the start and end activities created when mapping a structured activity.

$$A_{V\,flow(a')} = \{\alpha, \beta\} \cup \bigcup_{i=1}^{n} A_{V(a'_i)}$$

The creation of links depends on the runtime semantics of the activity. A BPEL flow's nested activities run in parallel unless they are connected by links. An activity nested in a flow cannot activate before the flow itself activates. This behavior is mapped to *P* by adding links

from the start activity α of the flow to each $\alpha A_{V(a'_i)}$, the start activities of the mapping of every BPEL activity in the flow.

On the other hand, the flow ends when all its activities have completed. Therefore, links are added from each $\beta A_{V(a'_i)}$, the end activities of the mapping of every BPEL activity in the flow, to β the end activity of the flow. These created links are:

$$E_{1\_flow}(a') = \bigcup_{i=1}^{n} \{(\alpha, \alpha V(a'_i), true)\} \cup \bigcup_{i=1}^{n} \{(\beta V(a'_i), \beta, true)\}$$

The set of links created from mapping a flow are then the links from $E_{1\_flow}(a')$, in addition to any links added by mapping the activities in the flow:

$$E_{V_{flow}(a')} = E_{1\_flow}(a') \cup \bigcup_{a_i \in A'_{flow}} E_{V(a_i)}$$

The join condition of each of these start activities is modified to ignore the status of the added links. The join condition of the end activity is simply true, because it needs to wait for the incoming links to fire but is not dependent on their status.

$$\forall a'_i \in A'_{flow} : \phi(\alpha V(a'_i)) = \phi(a'_i) \wedge \bigwedge_{e \in E_{1\_flow}(a') \cap E^{\leftarrow}(\alpha V(a'_i))} e \vee \neg e$$

$$\phi(\beta) = \bigwedge_{e \in E^{\leftarrow}(\beta)} e \vee \neg e$$

Sequence

A BPEL sequence activity contains:

- A tuple of other BPEL activities, $A'_{sequence} = (a'_1, ..., a'_n)$.

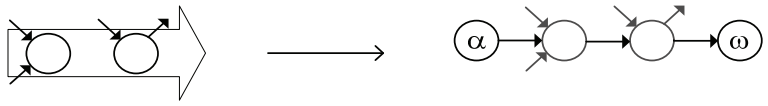


Figure 16: Mapping a BPEL sequence

Similarly to flows, the activities from the mapping are a start activity, an end activity, and all activities created from mapping its activities:

$$A_{Vsequence(a')} = \{\alpha, \beta\} \cup \bigcup_{i=1}^{n} A_{V(a'_i)}$$

Next, links are created to connect the sequence's activities together. The runtime semantics of a BPEL sequence are that the first activity in a BPEL sequence, a_1, may not start until the sequence itself has started. All activities in the sequence must occur one after the other: a'_n may not start until a'_{n-1} has completed. A sequence ends when the last activity in the sequence has ended. Therefore as shown in Figure 16, a link is added from the sequence's start activity α to $\alpha A_{V(a'_1)}$. Links are added between the nested activities: $\alpha A_{V(a'_i)}$ to $\alpha A_{V(a'_{i+1})}$, where *i* is the index of the activity in the tuple. Finally, a link is created from $\beta A_{V(a'_n)}$ to β.

The set of links added above for handle the sequencing of the activities are defined in the set $E_{1\_sequence}(a')$:

$$E_{1\_sequence}(a') = \{(\alpha, \alpha V(a'_1), true), (\beta V(a'_n), \beta, true)\} \cup \bigcup_{i=1}^{n-1} \{(\beta V(a'_i), \alpha V(a'_{i+1}), true)\}$$

The set of links created from mapping a sequence are therefore the links in $E_{1\_sequence}(a')$ that and those created from mapping sequence's activities:

$$E_{V\,sequence(a')} = E_{1\_sequence}(a') \cup \bigcup_{a_i \in A'_{sequence}} E_{V(a_i)}$$

The corresponding updates to the join conditions are:

$$\forall a_i \in A'_{sequence} : \phi(\alpha V(a'_i)) = \phi(a'_i) \wedge \bigwedge_{e \in E_1(a') \cap E^{\leftarrow}(\alpha V(a'_i))} e \vee \neg e$$

Notice that if an activity in a sequence suppresses the join condition, then *a'* is actually the scope that suppresses the join failure: the link added in $E_{1\_sequence}(a')$ from this activity has that scope as its source. Therefore, a join failure from such an activity still results in this link firing with a value of true.

The join condition of β is always true.

$$\phi(\beta) = \bigwedge_{e \in E^{\leftarrow}(\beta)} e \vee \neg e$$

While

A BPEL while contains:

- A condition, c'_{while}
- An activity, a'_{1_while}

A BPEL while activity is mapped to a loop w in *W*, such that:

$$w = (c'_{while}, A_V(a'_{1\_while}))$$

An activity is created to represent the loop in the process. That is, $K(w)$. It is the start and end activity for a BPEL while, because links of a loop are in fact links to/from $K(w)$:

$$a' \text{ is a while} \Rightarrow \alpha V(a') = K(w) = \beta V(a')$$

The activities created from mapping the while activity are $K(w)$ and the activities created from mapping the activity of the while activity itself:

$$A_{V\,while(a')} = \{K(w)\} \cup A_{V(a'_{1\_while})}$$

The links created from mapping a while activity are those from the activity within it:

$$E_{V\,while(a')} = E_{V(a_i)}$$

Switch

A BPEL switch consists of

- A tuple of *n* branches. Each branch contains
  - An activity, *a'*$_i$
  - A condition, *p'*$_i$

As for flows and sequences, we add a start activity and an end activity when mapping a switch. The set of activities created upon mapping a switch activity also contains the activities resulting from mapping the activity of each branch:

$$A_{V\,switch(a')} = \{\alpha, \beta\} \cup \bigcup_{i=1}^{n} A_{V(a'_i)}$$

Figure 17 shows an example of mapping a switch, where the activity of each branch is a primitive activity with suppress join failure set to 'yes'. The figure shows links leaving a scope directly, which is in fact realized using the pattern for linking from a scope.

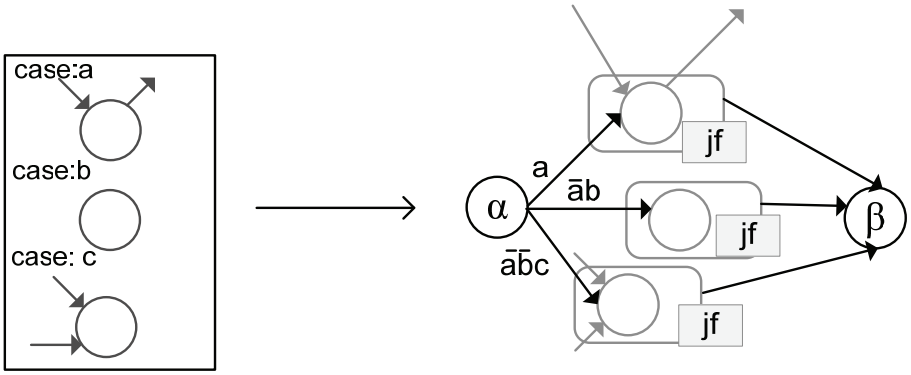


Figure 17: Mapping a BPEL switch

The runtime semantics of a switch activity were defined in 3.4.1 and requires evaluating the branch conditions in order. The meta-model does not encode link ordering. Recreating the ordering of condition evaluation is done by creating links from the start activity to each branch and defining the conditions on the links to account for the BPEL branch order.

Consider *p'*$_i$ to be the original condition on the i-th branch of the switch. For each *p'*$_i$, we create a corresponding $q_i \in C$ in the resulting meta-model

$$q_i = \begin{cases} p'_i & i = 1 \\ p_i \wedge (\bigwedge_{j=1}^{i-1} \neg p'_j) & i > 1 \end{cases}$$

Recall that all the activities in the mapping occur after having mapped the switch activity to switch$_{mod}$. Therefore, a'_i is either a scope with a fault handler for the join failure or a scope with a fault handler for 'killBranch'. We designate an activity c'_i of branch *i.* For the former case, c'_i is defined to be the main activity of a'_i. For the latter case, a'_i contains a single activity which itself is a scope, s'_j, and c'_i is defined to be the main activity of s'_j. Thus the links added to connect the activities of the switch to the start and end activities are:

$$E_{1\_switch}(a') := \bigcup_{i=1}^{n} \{(\alpha, \alpha V(c'_i), q_i))\} \cup \bigcup_{i=1}^{n} \{(\beta V(a'_i), \beta, true)\}$$

The links created from mapping a switch are therefore those in $E_{1\_switch}(a')$ and the links added from mapping the activities in the branches:

$$E_{Vswitch(a')} := E_{1\_switch}(a') \cup \bigcup_{i=1}^{n} E_{V(a'_i)}$$

The join conditions of the branches' start activities are modified to be the conjunction of the original join condition and the status of the added link. For each branch *i*:

$$\phi(\alpha V(c'_i)) := \phi(c'_i) \wedge (\alpha, \alpha V(c'_i), q_i)$$

Consider the links from the last activity $\beta V(a'_i)$ in the mapping of each branch to the end activity β of the mapping of the switch. These links leave from a scope that has a handler for the join failure and no incoming links itself. Thus, they will fire true regardless of whether the branch was taken or disabled. The join condition of β is thus always true.

$$\phi(\beta) = \bigwedge_{e \in E^{\leftarrow}(\beta)} e \vee \neg e$$

Pick

A BPEL pick contains:

- An set of n branches. Each branch contains
  - An activity, *a'*$_i$
  - An event trigger, *t'*$_i$

The pick activity allows a process to perform an exclusive choice from a set of different possible external inputs (messages or alarms). The branch of the event that happens first is taken and the rest are disabled. Another way to think of it is that switch provides *internal* choice while pick provides *external* choice.

The mapping of the pick activity is very similar to that of a switch activity except for: the implementation of the start activity and the values of the conditions on the branches. To avoid repetition, this section focuses on these differences, illustrated in Figure 18.

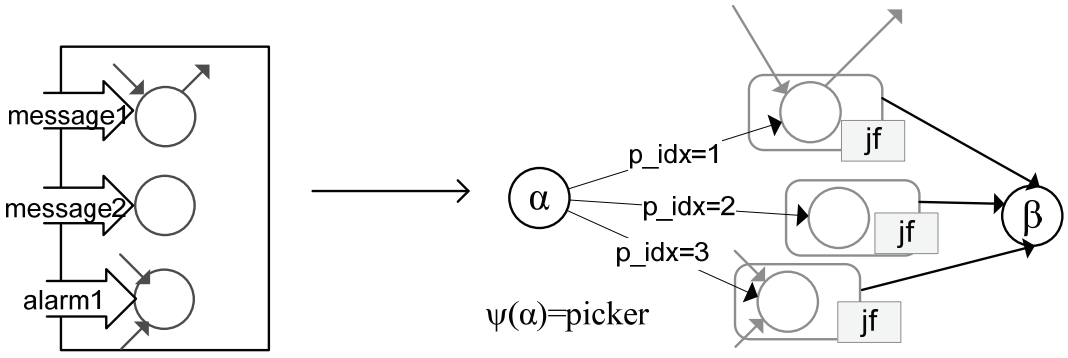


Figure 18: Mapping a BPEL pick

Consider an activity implementation, called *picker*, which takes an ordered set of event triggers as part of its definition. Upon activating, the implementation waits for one of the events to occur, and saves the index of the winning event along with any data that was provided by the event. The implementation then returns.

The implementation of the start of activity of a pick is therefore:

$$\psi(\alpha A_{Vpick}(a')) = picker$$

Using *p_idx* to refer to the index of the winning event written by the start activity, the conditions used on the links to the branches are:

$$q_i = (p\_idx = i)$$

This design for an unstructured pick activity has been used in the Web 2.0 workflow language Bite [CDKL07]: The Bite pick activity is a type of activity similar to *picker*. It contains a list of choices matching external events. One may use the value of the variable it wrote (and hence which branch was taken) on any transition condition in a Bite process.

This concludes the mapping from a BPEL to a process in the meta-model presented in this chapter. The next sections of this chapter address data representation and compensation handling.

3.5 Data in the Process

The data representation of the process meta-model is pluggable. In so far as we have described, we have not introduced a dependency on a particular data meta-model. Two approaches are prevalent in the area of workflow: the use of shared variables and the use of explicit data links. The former is what is used in BPEL today. The latter will be provided as an extension to BPEL in Chapter 4.

This chapter is concerned with the relationship with BPEL, and we therefore focus on shared variables. To use variables, one defines a set of variables on the process. Each condition in the process, except join conditions, may read from any variable. Each activity implementation may read and/or write to any variable during its execution.

3.6 Compensation

BPEL introduces the concept of default handlers for fault, compensation, and termination handling. Each scope therefore has at least one of each of these types of handlers associated with it, i.e. either custom defined or default handler. The resulting behavior is quite complex: We show in the following how using only explicit handlers not only simplifies the meta-model making it more usable, but in fact leads to less surprise behavior and allows one to relax certain BPEL restrictions.

BPEL uses scopes to provide nested activities with several types of additional behavior, like compensation behavior, data visibility behavior, etc. Such mixing of scope behavior is complex. A user may want to create a scope simply to restrict data visibility; however, BPEL scopes always have a default fault, termination and compensation handler associated with them if one of each is not explicitly provided. As a result, when a fault is thrown it gets caught and rethrown at every level of scope nesting between the faulting activity and the scope with the matching explicit handler. The default fault handler first stops immediately nested non-scope activities, calls the (default or explicit) termination handler of any running immediately nested scopes, and then calls (default or explicit) compensation handlers of its immediately nested completed scopes in default compensation order, and then itself rethrowing the fault up the hierarchy. This continues until a matching fault handler is found or no handler is found and the process itself is reached. A default termination handler of a

scope behaves similarly to the default fault handler but does not rethrow the fault. Notably, it itself calls the compensation and termination handlers of its own immediately nested completed and running scopes, respectively. The result is an interleaving of compensation between termination and fault handling as a fault gets thrown up, one scope at a time, from the scope in which it was thrown to the scope where it may be explicitly caught. The resulting compensation order can be calculated; however, there are many possibilities due to the rethrows and the limitation of being able to compensate only one level of scope nesting at a time but still reach all completed scopes from the handling scope down to the leaves. This makes it difficult for the process designer to take all the possibilities into account every time he or she adds a scope to a business process.

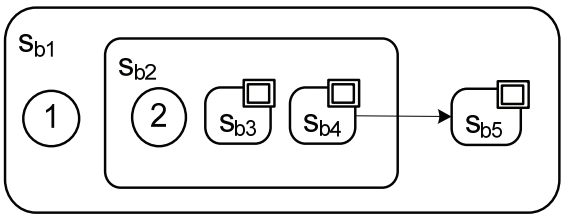


Figure 19: Example of BPEL default compensation order violating explicit control link reversal

Moreover, the BPEL approach results in giving priority to scope nesting over the control order specified by explicit control links, resulting in compensation order that violates the reversal of control dependencies [KOEN06]. For example, consider the process snippet in Figure 19 similar to the example used in [KOEN06]. It has five scopes, three of which have an explicit compensation handler (double lined box). We will show that BPEL will compensate s_{b4} before s_{b5} in spite of the control link. Assume that activity *1* faults and that at the time it does so s_{b3}, s_{b4}, and s_{b5} have already completed successfully and activity *2* is activated. As a result, scope s_{b2} is still activated. The fault is thrown to s_{b1} and caught by its default fault handler. This first performs default termination on s_{b2}, resulting in default compensation on s_{b2}'s completed children, s_{b3} and s_{b4}, in default order. As they are not connected by a control link, they may be compensated in parallel. Once their compensation completes, the termination handler itself completes and the next step of the default fault handler behavior of s_{b1} takes place: compensate its immediately nested completed scopes in default order. Since s_{b2} has been terminated that simply means compensating s_{b5}. Once that completes, s_{b1}'s default fault handler rethrows the fault to its parent scope. In summary, even though the process designer did not define an explicit fault or compensation handler for s_{b2}, its default handlers caused the compensation order to violate the link between s_{b4} and s_{b5}.

Our proposal centers on (1) changing the reachability of a compensation handler in BPEL to allow the compensation handler of a scope to be called from any of its ancestor scopes instead of only its immediately enclosing scope and (2) removing default fault and compensation handlers, as well as default and explicit termination handlers. Default compensation order, however, is kept using the compensate activity. The behavior is that an explicit fault handler first disables all running activities inside its associated scope, at any level of nesting. In the absence of default handlers, compensation is only started if a compensate activity is activated. The default compensation order for a scope is encoded in a graph we call a Compensation Order Graph (COG). The main idea of how this graph is calculated is that starting from the scope in which compensation was called and walking the resulting subtree in the scope hierarchy in a depth-first manner: When a scope with an explicit compensation handler is found, then it is added as a COG node and its children are not visited. Therefore, nested scopes without an explicit compensation handler do not play a

role in compensation order and only those with such a handler are considered thereby avoiding the conflict of the example in Figure 19. The control order between the nodes is then identified and reversed in the COG. Section 3.6.2 provides the details of COG construction, especially in the presence of nested loops.

To illustrate the structures created for compensation handling, and how they are used at runtime to enact the right behavior, we will use the example in Figure 20. It shows two loops and five scopes. Scopes s_1 and s_2 have fault handlers (box), while the three others have compensation handlers. Some examples in this figure of what is allowed here and not in BPEL, making this more flexible, are: s_1 may contain an activity to compensate s_5 even though it is not an immediate child scope, *a* may be a compensate activity even though it is not in a fault handler. The approach is simpler because there are no variations in the COGs: they can be calculated statically and deterministically at design time. Additionally, less scopes are involved in compensation. Finally, BPEL restricts all sibling scopes in a process from having a control cycle whereas we relax this restriction to being only for scopes with explicit compensation handlers, i.e. a COG must not have a cycle.

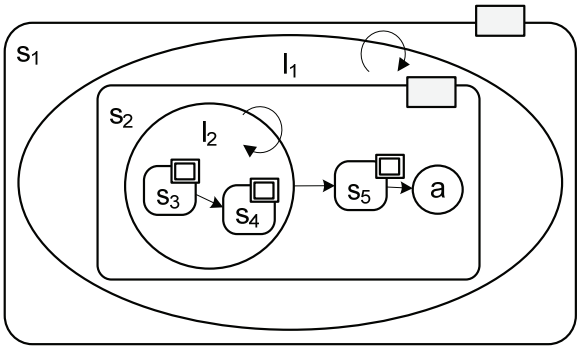


Figure 20: Example of scopes and loops with compensation handlers

3.6.1 Starting and Completing Compensation

Compensation is triggered only by the two kinds of compensation activities: (1) 'compensateScope' that refers to a scope (explicit) and (2) 'compensate' that does not do so. The former runs the compensation handler of the scope it refers to if that scope has an explicit compensation handler. It returns once all activities in the compensation handler of that scope have completed. Note that the handler may be run more than once, depending on whether the scope is in a loop and where compensation was called from, as will be detailed in section 3.6.4.

$$a = compensateScope(s_a) \Leftrightarrow \psi(a) = \begin{cases} \pi_1(j(s_a)) & s_a \in S_{comp} \\ no\_op & \text{otherwise} \end{cases}$$

On the other hand, 'compensate' compensates nested scopes in default compensation order, completing once the Compensation Order Graph (COG) has been traversed.

$$a = compensate \Leftrightarrow \psi(a) = COG(s'_0(a))$$

The compensation handler is treated as a separate process: no links can cross its boundary and its data access is limited. A compensation handler works with a particular instance of its associated scope, which must be kept track of (as will be shown in section 3.6.4). The handler's activities cannot write data visible outside the handler. They can only read data

from a snapshot taken when the particular instance of the associated scope completed. Such data can be then be loaded into the compensation handler process upon starting it.

3.6.2 Creating the Compensation Order Graphs (COGs)

The creation of the compensation order graph for a scope is shown in the definition of the function CREATE_COG shown below and described next.

```
function CREATE_COG(s)
1  loopCOGS = ∅
2  N_c ← ∅
3  GET_COMP_SCOPES(N_c, s)
4  S_n = {n ∈ N_c | ∃w ∈ W : K(w) ∈ A(s) ∧ A(n) ⊆ π2(w)}
5  W_c = {w ∈ W | ∃s_n ∈ S_n : A(s_n) ⊆ π2(w)}
6  W_ctop = {w ∈ W_c | ∄w1 ∈ W_c − {w} : K(w) ∈ π2(w1)}
7  N_c ← (N_c ∪ W_ctop) − S_n
8  E_c ← ∅
9  for each (n1, n2) ∈ N_c : n1 ≠ n2
10    if ∃e ∈ E : π1(e) ∈ linksAct(n1) ∧ π2(e) ∈ linksAct(n2)
11       E_c ← E_c ∪ {(n2, n1)}
12 G_c = (N_c, E_c)
13 for each n ∈ N ∩ W_c
14    N_cl = {s_n ∈ S_n | A(s_n) ⊆ π2(n)}
15    W_cl = {w ∈ W_c | K(w) ∈ π2(n)}
16    W_cltop = {w ∈ W_cl | ∄w1 ∈ W_cl − {w} : K(w) ∈ π2(w1)}
17    N_l = N_cl ∪ W_cltop
18    for each (n1, n2) ∈ N_l : n1 ≠ n2
19       if ∃e ∈ E : π1(e) ∈ linksAct(n1) ∧ π2(e) ∈ linksAct(n2)
20          then E_l ← E_l ∪ {(n2, n1)}
21    G_l = (N_l, E_l)
22    loopCOGs ← {G_l}
23 return (G_c, loopCOGS)
```

Intuitively, the COG's nodes represent all scopes nested in the scope and having an explicit compensation handler and its edges represent the reverse of the control order between these scopes. For the purposes of the COG, links coming from or to an activity nested inside the scope are treated as coming from or to the COG node representing that scope. The nodes of the COG will in fact represent not only scopes but also loops. Loops are included because the compensation of a scope in a loop depends on how many times the loop iterated. Each loop node in the scope's COG, also referred to as a scope's main graph, will have a corresponding COG called a loopCOG.

The CREATE_COG function starts by collecting all scopes nested in *s* that have an explicit compensation handler (line 3). It does so using the helper function GET_COMP_SCOPES that traverses, depth first, the scope subtree rooted at the scope itself. It skips child scopes in the scope's fault handlers. If it finds a scope s_c with an explicit compensation handler, it adds s_c to the N_c and does not visit the children of s_c.

```
function GET_COMP_SCOPES(N_c, s)
  for each s_c : parent(s_c) = s ∧ s_c ∉ A(F(s))
    if s_c ∈ S_comp
       N_c ← N_c ∪ {s_c}
    else GET_COMP_SCOPES(N_c, s_c)
```

If a scope returned by GET_COMP_SCOPES is nested in a loop, the corresponding node created in the COG represents that loop and not the scope. The set S_n is created to contain the set of scopes (line 4) that are nested in a loop where the loop itself is nested in s. The set W_c contains the loops that have elements from S_n nested within them. The set W_{ctop} contains the elements of W_c that will be represented as nodes in the COG of the scope. An element in W_c is included in W_{ctop} if that element is not nested in another member of W_c. The nodes of the COG are created (line 7): compensation relevant scopes not nested in loops and top level loops containing other compensation relevant scopes.

The edges of the COG depend on the paths between the nodes (lines 8-11), as we need to reverse the order imposed by the links in the process itself. If there is a link in the process from an activity in one node, n_1, to an activity, n_2, in another node of the COG, then an edge (n_2, n_1) is placed in the COG. For the case that the node is a loop, w, then we consider the links from and to $K(w)$ of the loop. Therefore, define a helper function:

$$linkActs(a) := \begin{cases} A(a) & a \in S \\ K(a) & a \in W \end{cases}$$

Having created the nodes and edges for the main graph of the scope, a COG is created for each loop node in the main graph (lines 13-21). This is done in similarly as for the scope's main graph. The nodes are scopes and loops nested in the loop itself.

An example of the COG of the scope s_1 from the example in Figure 20 is shown in Figure 21. A dashed line goes from a loop node in a COG to the node's loopCOG.

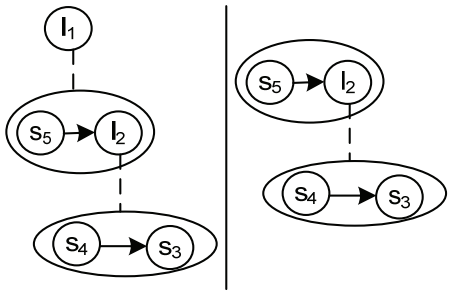


Figure 21: The COG of scope s_1 (left) and s_2 (right) from Figure 20, including loopCOGs of l_1 and l_2

3.6.3 Tracking Instances

In addition creating the COGs, the proper instance of a scope must be used when compensation is called. Loops lead to groups of scope instances. In order to do keep track of these, scope and loop instances are stored in a LIFO queue of instance information called the ScopeLoopQ. A ScopeLoopQ is associated with each scope, and each loop that contains at least one scope (at any depth) with an explicit compensation handler. It contains one entry for each time the scope or loop runs in the lifetime of one instance of the process. Each entry contains: a scope instance identifier, a Boolean that is present only for scopes and is only true if the scope instance completed successfully and has not been compensated, and a (possibly empty) set of name value pairs we call ChildQs.

One member of ChildQs exists for each node in the scope or loop's (loop)COG. The name is that of the scope or loop corresponding to the (loop)COG node, and the value is another LIFO queue. Each entry in it contains an identifier of the loop or scope instance.

The ScopeLoopQs enable the determination at runtime of the group of scope instances (in BPEL, named 'compensation handler groups') related to each instance of a parent scope that needs to be compensated. An example is shown in Figure 22 of the ScopeLoopQs for the scopes and loops in Figure 20. It assumes that both loops iterated twice and *a* faulted in the second iteration of l_1. Section 3.6.4 will explain its usage compensation.

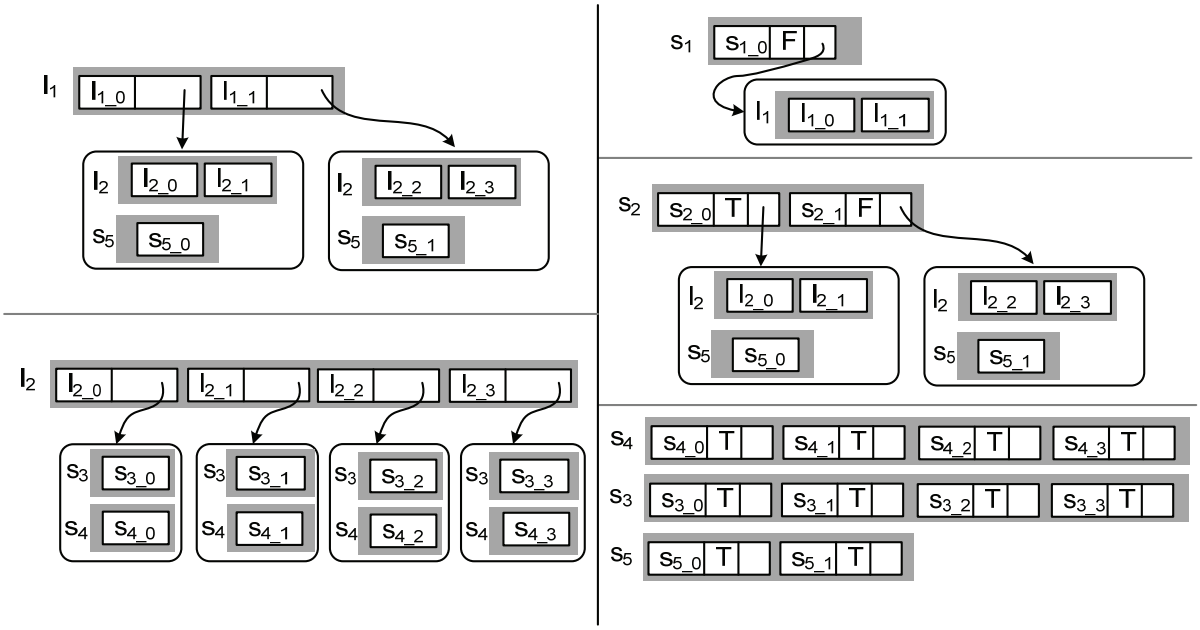


Figure 22: ScoopLoopQs and respective ChildQs for the scopes and loops in Figure 20

3.6.4 Running Compensation: Handlers and COGs

We now move to the runtime aspects of the compensation artifacts explained above. Consider first compensation triggered by the compensateScope activity. When a compensateScope activity *a* with a parent scope s_p that refers to a scope s_{ref} is activated, the instance of s_p in which *a* is activated is looked up in the ScopeLoopQ. Then, s_{ref}'s compensation handler is called once for each instance of s_{ref} that ran in the retrieved instance of s_p. If the Boolean in ScopeLoopQ for an instance of s_{ref} is false, the compensation handler is a *no_op*.

For example, assume that s_2 caught the fault from *a*, and the fault handler had a compensateScope(s_4) activity. Then, one sees in the scopeLoopQs (Figure 22) that the compensation was called from s_{2_1}. From there, one finds two instances of l_2 leading in turn to two instances of s_4 that occurred in s_{2_1} and compensates them in order: s_{4_3} then s_{4_2}. Upon doing so, the Boolean in each compensated scope's scopeLoopQ is changed to false. Notice, in contrast, that if it had been asked to compensate s_5, then it would have only compensated one instance: s_{5_1}.

If the compensate activity is used, compensation must occur in the default order. Thus, the COGs are used. Each COG and loopCOG is in fact a process, with all transition conditions and join conditions set to true. If an activity, whose implementation is a COG, is activated, the relevant COG is run just as any other process (after first loading in any necessary data values from a snapshot of the associated scope instance). Each COG scope node is made into a compensateScope activity, referring to the scope it represents. Each COG loop node, corresponding to a loop is made into a new kind of activity, the *loop-compensate* activity. The implementation of this activity runs the *loopCOG* of that loop. As was done when

compensating a scope, the instance of the loop is found in the ScopeLoopQs and the loopCOG is run as many times as the loop had run for the found instance from which it was called. For example, if compensating s_1 for the instance s_{1_0}, then the COG of s_1 is run. That has one node, for loop l_1. This leads to the loopCOG of l_1, showing to first compensate s_5, then l_2. The ScopeLoopQs show that l_1 ran two instances during s_{1_0}. Therefore, the loopCOG of l_1 is run once for l_{1_1} and again for l_{1_0}. Another example is that of l_2. Notice that eventually, l_2 would be compensated four times once for each of its instances. On the other hand, if default compensation had been called from s_{2_1}, then one would have only compensated l_2 twice: once for l_{2_3} and again for l_{2_2}.

3.7 Conclusion

In this chapter, we have provided a meta-model based on graphs that enables modeling processes with fault and compensation handlers. We showed that enabling fault handling truly does allow modeling calculus-based constructs with links that cross their boundaries. This was done by mapping BPEL, which contains such constructs, to the presented process meta-model. In addition to providing a process meta-model that is very close in the design artifacts to the actual runtime behavior, we also verified our claim from [CKLW03] that the enabler in BPEL for combining calculus and graph-based approaches is the handling of the join failure.

We introduced a new compensation handling approach in the process meta-model that is more flexible and provides more coverage than that of BPEL itself. In addition, our approach is more comprehensible for designers: our meta-model enables one to truly reverse the control order specified by explicit links and enables handling compensation at the level where a fault is caught and only when required by the user. This simplifies the runtime behavior, enabling the designer to easily comprehend the compensation order instead of having it obfuscated by the many possibilities due to faults being thrown, rethrown, and percolating through sets of intermediate scopes as in standard BPEL.

This meta-model, in particular its runtime semantics, its fault handler lookup mechanism, and its treatment of compensation in the presence of loops, will be instrumental in enabling us to define the behavior for split BPEL processes in Chapters 5 and 6.

Chapter 4 Defining BPEL-D

In many business process meta-models, both data flow and control flow are made explicit. BPEL has explicit control flow, but that of data is implicit through the use of shared variables. While shared variables are known to programmers, explicit data flow is attractive for several reasons: data flow is often viewed as a first class modeling construct by process modelers. Additionally, the runtime can be optimized if data flow is known, such as in streaming large data between activities and so forth. Algorithms that deal with process agility, flexibility, and optimization usually require data analysis and produce something similar to data links (data dependency graphs) in order to do so. Additionally, as we will show in Chapters 5 and 6, processes with data links are much more amenable to splitting that those using shared variables.

BPEL provides several advances in process modeling. These advances include fault handling, direct tie-in to Web services, and mind-share due to standardization and high adoption in products. Hence, one would like to re-use BPEL's concepts instead of moving to another process model, yet still enable explicit data modeling.

To address this, we propose a variant of BPEL called BPEL-D that extends the modeling constructs and execution semantics of BPEL by replacing the implicit data flow with explicit data flow. The operational semantics of control flow in BPEL-D is simply that of BPEL. BPEL-D provides coverage for all BPEL 1.1 constructs except for event and termination handlers on scopes and some restrictions on data passing especially to fault handlers. The main reason for leaving these out is that the main mechanisms and concepts for explicit data flow are shown without them and they are not used in the splitting algorithms.

In this thesis, the informal grammar for XML syntax used in the BPEL standard is adopted when defining new XML constructs. It uses the following characters, appended after an element or attribute to indicate whether the element or attribute is optional and how many times it may appear: '?' to mean zero or one, '*' to mean zero or more, '+' to mean one or more. Additionally, items grouped in parenthesis '()' and separated by '|' are syntactic alternatives.

4.1 Data Links

Similar to FDL and WSFL, each activity has a set of containers. In BPEL-D, the elements of this set form two disjoint subsets: an input container set and an output container set. We refer to an element in these subsets as either a container in the input(output) container set or simply as an input(output) container. A container's contents are one tree of data items, whose definition is either an XML Schema Element, an XML Schema Simple Type, or a WSDL Message. A transition condition of a link has access to the containers of the link's source activity. A data link defines the passing of data between two activities by mapping from containers of one activity to containers of the other activity.

4.1.1 Data Links for Loops

A loop, i.e. while activity, has a containers set. Data links to a loop's input containers may

come from other activities. They may also come from its own output containers, in which case we call them 'iteration-datalinks'. An example is shown in Figure 23. For simplicity, the figures in this chapter omit containers of primitive activities. Additionally, they omit a 'flow' activity used as the immediate child on activities that require exactly one child (scope, loop).

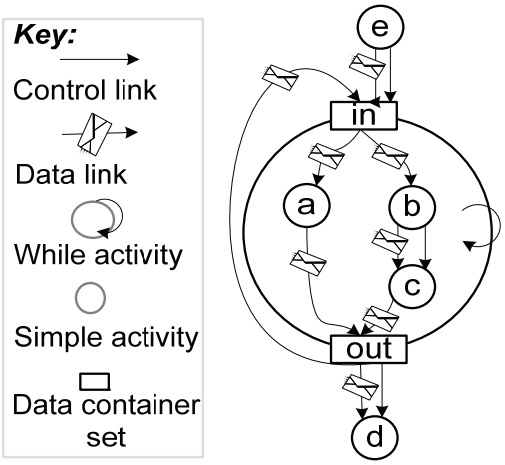


Figure 23: A BPEL-D loop

The input containers are populated from the iteration-datalink if the loop has completed at least one iteration. Otherwise, they are populated from the other data links. The condition of the loop is in terms of the loop's input containers. Data links leaving the output containers include the iteration-datalink and any data links to other activities.

4.1.2 Data Links for Scopes

Data flow for a scope is focused on the scope's fault handlers and its compensation handler. A fault handler has zero or one input container and no output containers. Data links and the input container set related to a fault handler are illustrated in Figure 24. If an activity faults, any data it creates that is related to the fault is placed into the input container of the matching fault handler. Therefore, the input container of a fault handler is not a target of any data link. A data link whose target is inside a fault handler must also have its source in that fault handler.

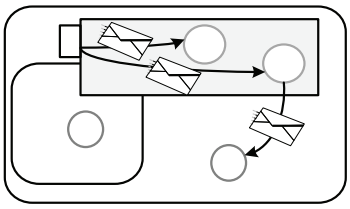


Figure 24: A BPEL-D scope: Data links and fault handlers

Data, however, may flow from activities inside a fault handler to activities outside the handler, i.e. a data link may have its source in a fault handler and its target outside the handler. An example is shown in the center of Figure 24.

Data links and the input container set related to a compensation handler are shown in Figure 25. A compensation handler has input containers and no output containers. Data links may have any activity in the associated scope as their source and the compensation handler's input container as their target. However, data does not flow out of a compensation handler: a data link whose source is in a compensation handler must have its target in that handler. This is

consistent with BPEL 1.1 compensation handler behavior: modifications to data made in the handler are not visible to activities outside it.

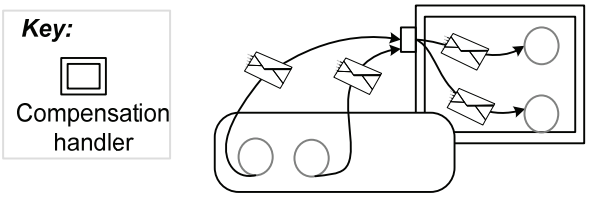


Figure 25: A BPEL-D scope: Data links and compensation handlers

4.1.3 Data Links Meta-Model

An activity, a fault handler, and a compensation handler each have a containers set. Use $i(A_1)$ and $o(A_1)$ to denote, respectively, the output container set and input container subsets of the container set of the activity or handler A_1.

The input containers of an activity or handler are populated by data link(s) mapping from (1) containers in the output container sets of other activities, (2) containers in the input container set of a compound activity, fault handler, or compensation handler that it is nested in. The output containers of a primitive activity are populated by the activity itself. The output containers of a compound activity are populated from containers in the output container sets of activities nested within it. These possibilities are summarized in the *Data_Dep* function defined further on.

A data link $d(A_1, A_2)$ specifies a set of data connector maps where each assigns from a container in A_1's container set to a container in A_2's container set. A data link must not cross the boundary of a loop. Additionally, it is required that A_2 be reachable via control flow from A_1 in order for A_1 and A_2 joined by a data link $d(A_1, A_2)$. For example, a data link cannot exist between two parallel activities or to an activity in a fault handler of a scope from an activity not in that fault handler.

Consider the set *AH* to be the set of activities *A*, union the set of compensation handlers J, union the set of fault handlers F:

$$AH = A \cup J \cup F$$

Define a function $Data\_Dep$ that takes two elements of AH:

$$Data\_Dep(A_1, A_2) = \begin{cases} \wp(i(A_1) \times i(A_2)) & (A_1 \text{ structured} \vee A_1 \in J \cup F) \wedge (A_2 \text{ nested in } A_1) \\ \wp(o(A_1) \times o(A_2)) & (A_1 \text{ nested in } A_2) \wedge (A_2 \text{ structured }) \\ \wp(o(A_1) \times i(A_2)) & \text{none of the above and } A_1 \in A \end{cases}$$

Therefore, we define a data connector map, adapted from [LERO00], as follows:

$$\Delta : AH \times AH \rightarrow \bigcup_{A_1 \in AH, A_2 \in AH} Data\_Dep(A_1, A_2)$$

with the following conditions summarizing the restrictions above:

- $\Delta(A_1, A_2) \in Data\_Dep(A_1, A_2)$
- $\Delta(A_1, A_2) \neq \emptyset \Rightarrow A_2$ is reachable from A_1 via explicit control dependencies, and both A_1 and A_2 are either not in a loop or are in the same loop.

- $A_1 = A_2 \Rightarrow A_1$ is a while activity.

The data connector maps are similar to the BPEL assign activity's list of 'copy' elements using 'from' and 'to' that refer to variable names. Advanced data manipulation should be done within an assign activity or as a service call.

4.1.4 Semantics of Data Links

Once an activity completes, its output containers are materialized and the data maps on its outgoing data links are evaluated and the containers of the input container set of the target activities/handlers are thus populated. If an activity is disabled (i.e. due to dead-path elimination or a fault), then its output containers are not materialized and no data flows down the data links. The restriction of enforcing a control dependency between the activities when a data dependency exists ensures that an activity's data is ready once it is reached in the control flow.

In case of a conflict with writes to the same location for which the data link source activities ran successfully, the winner is chosen at random. The motivation, also in [LERO00], is to support the very common situation where two paths merge at an activity but only one of them will ever run in any instance.

A special case is needed for the containers of a loop. A loop's input containers are reset at the end of an iteration. Its output containers are reset at the beginning of an iteration. Data links from the loop to input containers of other activities that are not the loop itself are only evaluated once the loop itself completes, whereas iteration-datalinks are evaluated every time a loop iteration completes.

4.2 BPEL-D Syntax

The definition of a complete BPEL-D syntax, based on the concepts presented thus far in this chapter, is provided in [VAZQ07]. In this section, we show the syntax of the constructs most relevant for the splitting algorithms presented in Chapters 5 and 6.

4.2.1 Data Links and Containers

BPEL-D activities, fault and compensation handlers must be uniquely named. Handler names are BPEL extensions. Data links (see Table 4), are sub-elements of the process itself. Each defines a source and target activity/handler and a set of maps. Each map creates a mapping from a container of the source to a container of the target.

```
<datalinks>
  <datalink sourceActivity="…" targetActivity="…"> +
      <map sourceContainer="…" targetContainer="…"/> +
  </datalink>
</datalink>
```

Table 4: Data links syntax

The meta-model of an activity's containers is a set of containers with an input container and an output container subset. In the syntax presented in [VAZQ07] and shown in Table 5, the containers set is modeled using a 'containers' element. Each 'containers' element contains one or more 'container' elements, each having a name, an optional type, and a direction. The type is specified similarly to that of a BPEL variable: by one of the attributes 'messageType', 'type' or 'element' depending on whether it refers to a WSDL message type, an XML

Schema simple type or an element respectively. The direction determines whether the container is part of the input or the output subset. It states whether the container is used for data that is received from or sent to external partners (toPartner, fromPartner), whether it is part of the input or of the output (input, output), or whether it is data to be sent with a fault (faultData). Data from a partner for an activity forms part of the activity's output data, while data to a partner forms part of an activity's input data. The relation to the partner is used so that one does need yet another element to state which part of an activity's input to send in a reply or an invoke and which container in the output container set to save data from a partner in a receive or an invoke.

```
<container name="…" messageType="…"? element="…"? type="…"?
      direction="input|output|toPartner|fromPartner|faultData"/>
```

Table 5: Container syntax

Therefore, the input container set of an activity in the meta-model is the set of container elements in the syntax where the direction is either 'input' or 'toPartner'. The output container set of an activity in the meta-model is the set of container elements in the syntax where the direction is either 'output', 'fromPartner' or 'faultData'.

4.2.2 The Primitive Activities

The primitive activities follow a similar pattern, so we will show only invoke and assign fully. A transition condition of an outgoing control link of an activity reads data from the source activity's container. Therefore, the empty activity does not read data to do its actual work, but may need containers in the input container set for its control links' conditions.

Invoke

The invoke activity (see Table 6) sends data to a partner and may receive data from a partner. The latter, if present, forms this activity's output data. The activity may need additional data for its transition conditions, in which case one uses containers whose direction is 'input'.

```
<invoke …>
  …
  <containers>
     <container name="…" … direction="input"/>*
     <container name="…" … direction="toPartner"/>?
     <container name="…" … direction="fromPartner"/>?
  </containers>
</invoke>
```

Table 6: Syntax of containers on an invoke activity

A container element with a direction whose value is fromPartner is optional: it is used if the invoke does not send an empty message. A container element with a direction whose value is toPartner is also optional: it is used if the invoke is request-response.

Assign

The assign activity (see Table 7) creates its output data in-line. This occurs in its 'copy' statements, which refer to different containers of the assign activity's containers element.

A container, with a direction whose value is input, is optional because the assign activity may not need data from other activities, i.e. if it is assigning from a string, number, etc.

```
<assign …>
  …
  <containers>
     <container name="…" … direction="input"/>*
     <container name="…" … direction="output"/>+
  </containers>
  <copy> +
   <from container="…" …/><to container="…" …/>
  </copy>
</assign>
```

Table 7: Syntax of containers on an assign activity

4.2.3 While

The while activity (see Table 8) has optional input/output container elements.

```
<while …>
  …
  <containers>
     <container name="…" … direction="input"/>*
     <container name="…" … direction="output"/>*
  </containers>
</while>
```

Table 8: Syntax of containers on a while activity

4.2.4 Scope

The scope syntax (see Table 9) shows containers for fault and compensation handlers.

```
<scope …>
  …
  <faultHandlers>?
    <catch name="…" …>
     <containers>?
      <container name="…" … direction="input"/>*
     </containers>
    </catch>
    …
  </faultHandlers>
  <compensationHandler name="…">
     <containers>?
      <container name="…" … direction="input"/>*
     </containers>
  </compensationHandler>
   …
</scope>
```

Table 9: Syntax of containers on a scope activity

4.3 Conclusion

Explicit data flow is often preferred over implicit data flow. However, there is no current process meta-model for Web services that uses explicit data links. This chapter has presented BPEL-D, a variant of BPEL with explicit data flow. We have defined the concepts behind it, and provided the syntax for the subset of BPEL-D used in the splitting algorithms in this thesis. Data containers for primitive activities are very similar to those in [LERO00], except for special care because typically BPEL activities represent concrete implementation behavior (e.g. throw, assign, etc.). This resulted in the introduction of a modeling construct to

specify, for each container of an activity, where data needed or written by the activity is located (e.g. toPartner, faultData, etc).

The data links between loop iterations and crossing boundaries of a scope's fault or compensation handlers are a new contribution. Data passing into a fault handler is restricted to originate only from the faulting activity; this is because data flow must follow control flow. However, data passing for compensation handlers is similar to what is defined in BPEL 1.1: a compensation handler reads from a snapshot of the data at the time of scope completion and does not pass data to activities outside the handler's boundary.

In Chapters 5 and 6, we will show how both, a BPEL-D process and a standard BPEL process can be fragmented. Both share the steps for fragmenting their control dependencies. However, fragmenting BPEL-D processes is simpler than fragmenting standard BPEL processes when it comes to fragmenting a process's data dependencies.

Chapter 5 Process Fragmentation

Business processes are defined to encode the steps that enable one to achieve a certain goal. Most process languages today are not designed to enable fragmentation of the process after it has been defined. Sophisticated runtimes such as OSIRIS [SWSS03] have a peer to peer execution model in mind, thus running a process over a distributed set of machines. However, in such approaches, specialized middleware is required and the fragments of the process are determined by the runtime based on infrastructure and quality of service concerns.

In Chapter 1, we motivated the need to enable a designer to create partitions from a business process based on business needs. An illustration of such a partition is shown in Figure 26. This thesis provides the algorithms to create an executable process for each fragment, i.e. a participant in a chosen partition. In this chapter, we present the design principles, the encoding of information needed about the unsplit process model, and how control and data dependencies get split.

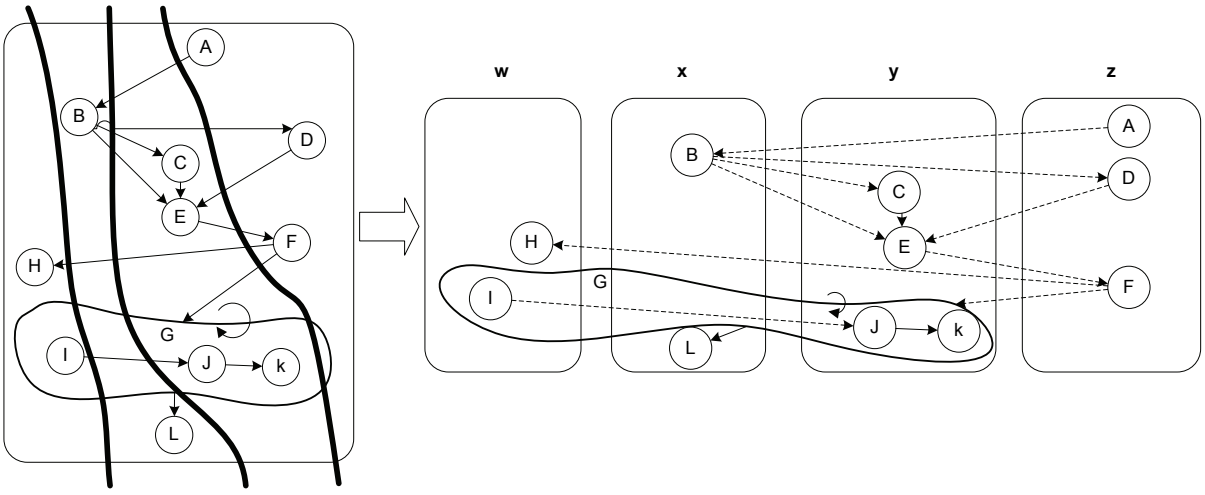


Figure 26: Splitting a process (left) to create the four process fragments (right)

This chapter handles fragmentation for both BPEL-D and for BPEL, based on our work in [KHLE06] and [KHKL07B]. The splitting of control dependencies will be the same for both; however, the algorithms will diverge for splitting data dependencies. The approach for fragmenting BPEL-D provides the basic concepts and structures for splitting a data dependency between two activities. These are then reused as the basic building blocks on which we build different patterns to enable fragmenting the more complex data dependencies resulting from fragmenting BPEL itself, with its use of shared variables. In a nutshell, a data or explicit control dependency that exists in the unsplit process model, between two activities placed in different fragments is flowed via explicitly exchanged messages: sending and receiving activities are added, linked by corresponding partnerLinkTypes. A pattern of BPEL constructs (‘sending block’) consisting of at least one invoke activity in one fragment constructs and sends the needed information to a corresponding pattern of constructs (‘receiving block’) consisting of at least one receive activity in another fragment. The receiving block receives and handles the transmitted information so as to effectively reproduce the behavior present before the fragmentation, when both activities were in the same process.

For cases where control is implicit, e.g. loops and scopes, a coordination framework will be used to handle the fragmentation. In such a case, a coordinator interacts with the fragments to control their execution. The details of using coordination will be presented in the next chapter.

We use the term ‘unsplit process’ to refer to the process model that one aims to fragment. We use the term ‘split process’ to refer to the set of process fragments resulting from a partition of an unsplit process.

5.1 Design Guidelines

There are several concerns that heavily influenced our design decisions. These include increasing portability, increasing interoperability, maintaining transparency between the main process and the resulting fragments, and promoting a balance between control imposed by the main process and the autonomy of a fragment.

Portability of a business process model is the ability to take the same model and use it in different tools and runtimes without having to modify it. In order to address this concern, the fragments we create are BPEL process models. BPEL already enjoys large support in industry and academia with a large number of available runtimes and tools ranging from open source packages to full products by the leading software companies. We maintain compliance to the BPEL specification, adding extensions to both the language and the runtime only in exceptional cases. This occurs in the case of fragmenting loops and fault-handling, compensating scopes. If these activities are not split, the resulting fragments are standard BPEL processes that can run on any engine. If, however, these activities are split, then the extensions required are performed in a manner compliant with BPEL’s extension capabilities and the extensions to the middleware are done such that inter-fragment middleware interactions are performed using the standard WS-Coordination framework. The impact to portability is kept as small as possible in the latter case by using WS-Coordination instead of proprietary middleware.

Interoperability of a business process is its ability to interact with other entities (processes or otherwise) that are written in different languages and executed on different runtimes. Our approach addresses interoperability by using WSDL interfaces for all interactions between fragments, between fragments and external services, and between participants (fragments) and a coordinator when a coordination framework is used.

Transparency is the ability to clearly understand the work done by a fragment itself and ease of relating the fragments to the unsplit process model. Being able to relate constructs in fragments to the unsplit process model makes the fragments more comprehensible to the designer, easier to monitor, etc. In other words, the work performed by a fragment is not hidden under layers of mappings or drastic graph rewrites. This is achieved by minimizing modifications to the process model and using the same process modeling language for both the main process and the fragments. Additionally, the places in the process where the middleware will take over and perform any special handling (i.e.: for fragmented loops and scopes) are clearly labeled in the process model and the behavior is well-defined.

The balance between the control imposed by the unsplit process and the autonomy of its fragments is represented by the amount of freedom that a fragment owner has to modify a fragment without requiring modifications to the unsplit process model. This freedom can be

regulated by the use of the concept of 'abstract processes' defined in BPEL. An abstract process allows one to define certain parts of a process model as modifiable ('opaque'). The relationship of abstract processes to process fragmentation is that one can create abstract processes from the generated fragments. Autonomy given to a fragment is then regulated by the amount of modifiable parts within the corresponding abstract process. At one end of the spectrum, the abstract process contains no modifiable parts, i.e. the fragment has no autonomy. At the other end of the spectrum, only the activities that handle passing control and data dependencies between the fragments are non-modifiable and the rest of the corresponding abstract process (including activities from the main process) is allowed to be changed. I.e. in this case, the fragment owner has full autonomy to change anything in the fragment.

5.2 Coverage

The fragmentation approach presented in this thesis addresses most constructs of BPEL. Some constructs have been left out mainly because we identified the basic mechanisms and concepts for fragmenting business processes by dealing with splitting links, scopes and loops; thus, no substantial new results are expected when dealing with these additional constructs. The following is the subset of BPEL and BPEL-D supported for our approach for the definition of an unsplit process that a user wishes to fragment:

- Exactly one correlation set, which we call the 'global correlation set', must be used between all fragments of the same process.
- The supported structured activities are: 'flow', 'while' and 'scope' activity.
  - Links are allowed, but not from/to the boundary of a 'flow' activity. Therefore, no special handling is needed to split such flow activities. They are only used where one needs more than one activity but BPEL syntax requires exactly one (e.g. in a while activity, a fault handler, etc.).
  - Scopes may have compensation and/or fault handlers but not event or explicit termination handlers.
  - Loops and scopes must be uniquely named within a process.
- The supported primitive BPEL activities are: all except 'terminate', and endpoint reference copying in an 'assign'.

When creating a partition of a process, some restrictions are introduced to ensure proper behavior. These are:

- A 'receive' and its corresponding 'reply' are disallowed from being placed in different participants. This is due to the fact that they refer to the same operation, usually offered over a synchronous protocol (such as HTTP).
- The join condition of a split loop or scope is restricted to a conjunction of the local join conditions of each fragment of a split loop or scope.
- The loop condition is assigned to exactly one loop fragment.
- Compensation is a recovery mechanism, and thus must not fail.

5.3 Defining a partition

A designer fragments a process by defining a partition of the set A of all primitive activities in the process. Consider $P_{n,}$ a set of participant names. Every participant, *p,* belonging to the set of participants, P_a *,* consists of a participant name and a set of one or more activities such that:

$$P_a \subseteq (P_n \times A)$$

A participant must have at least one activity, no two participants share a (primitive) activity or a name, and every primitive activity of the process is assigned to a participant. These restrictions on P_a are expressed as follows:

$$\forall p \in P_a : \pi_2(p) \neq \emptyset$$

$$\forall p_i, p_j : i \neq j \Rightarrow \pi_2(p_i) \cap \pi_2(p_j) = \emptyset$$

$$\bigcup_{p \in P_a} \pi_2(p) = A$$

Scopes and loops are split by assigning the activities in them to different partners. The designer does *not* place the scope or loop itself in a participant.

5.3.1 Loops

Any loop whose nested primitive activities are placed in more than one participant becomes a *split loop*. One *fragment* of the loop is in each participant that contains at least one activity nested in the split loop.

To split a loop the designer assigns the activities inside the loop to participants and designates exactly one fragment of the loop as responsible for evaluating the loop condition. In order to encode the fragment responsible for the loop condition, we define L_n to be the set of names of split loops. Then, we define a map L_c that associates every split loop name, $l \in L_n$ with its responsible participant name:

$$L_c : L_n \rightarrow P_n$$

To illustrate, consider the partition of the process in Figure 26. Four participants are created, and using the identifiers in the figure to refer to the activities, the partition is $P_a = \{p_w = (w, \{I, H\}), p_x = (x, \{A, B, L\}), p_y = (y, \{E, C, J, K\}), p_z = (z, \{D, F\})\}$, $L_c(G) = y$. .

5.3.2 Scopes

Any scope whose nested activities, including activities in fault and compensation handlers, are placed in more than one participant becomes a *split scope*. If the fault handler or compensation handler also has activities in more than one handler, we call it a *split handler*. Similarly, a *fragment* of the scope is in each participant that contains at least one activity nested in the split scope. To split a scope the designer places the activities inside the scope in different participants. This includes activities inside fault and compensation handlers.

Consider Figure 27, showing four ways of splitting a scope with a fault handler between two participants. The scope body (rounded rectangle) has two activities *a* and *b* with a link between them. The fault handler (normal rectangle) has two activities *x* and *y*.
For example, to split the scope above as shown in the bottom right corner (4) of Figure 27, one would have a partition with two participants, $P_a = \{p_1, p_2\}$, and $\{a, y\} \subseteq \pi_2(p_1)$ and $\{b, x\} \subseteq \pi_2(p_2)$.

Having explained the mechanisms by which one splits a process, we now move on to look at the properties of the resulting fragments.

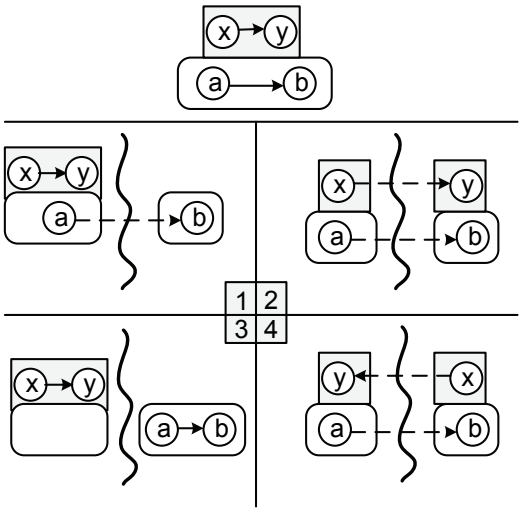


Figure 27: Splitting a scope with a fault handler into two fragments

5.4 The Rubber Band Effect

In BPEL, scopes and loops must be strictly nested. This restriction must be maintained when a process is split. It results in a property we define as the 'rubber band' effect. Note that a process itself is treated as a scope in BPEL.

Corollary: If *x* is a split scope or loop, all the ancestor scopes and loops of *x* (including the process itself) must also be split.

Particularly, if *x* has fragments in participant *p1* and participant *p2*, then all ancestor scopes/loops of *x* also have fragments in *p1* and *p2*. Note that a scope s_c in a fault handler of scope s_p and having no parent scope in that fault handler is treated as a child of s_p for the purposes of the rubber band effect. Therefore, if s_c is split then s_p is as well.

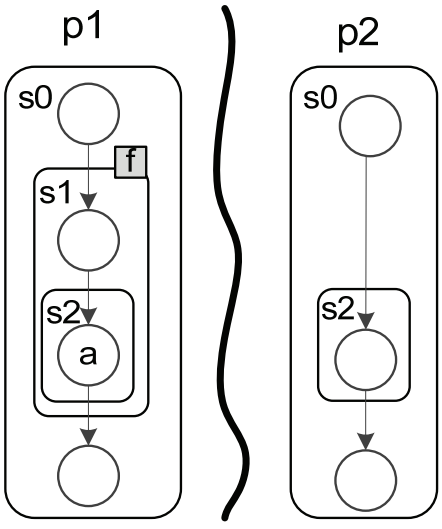


Figure 28: Violating the rubber band effect: *s2* is split, but parent *s1* is not

Violating the rubber band effect would result in inconsistent scope state. Consider the (disallowed) scenario in Figure 28. Each scope in the figure is labeled with its name, with *s0* being the process itself. The process is fragmented such that *s0* and *s2* are split between the participants *p1* and *p2*. However, *s1* is not split in this example: It belongs completely in *p1*,

and is the parent of *s2* in that participant. Furthermore, s1 has a fault handler *f*. If this were allowed, then a fault matching this fault handler thrown from activity *a* would be caught by *s1* in participant *p1*. As a result *p1*'s process would handle the fault to correct the situation with regards to its own work, but *p2*'s process would not. The reasons for the rubber band effect are therefore similar to BPEL's reasoning for disallowing overlapping scopes that are not strictly nested. Thus in our approach, the example below can be split so that the primitive activities are in the same participants, but *p2* would contain a fragment of *s1* as well.

5.5 Encoding common information

The processes created by fragmentation are related to each other because they originated from the same process model. When coordination is used, some information about the unsplit process model and its fragments must be provided to the coordinator so that the operational semantics of the unsplit process model can be preserved at runtime. This section describes this information.

The process fragments created from a process model are grouped together using a new XML element, 'splitProcess' shown in Table 10. It identifies the fragment processes by relating each to the name of the responsible participant. The correlation set element is used for instance matching and will be detailed in section 5.5.1.

```
<xs:splitProcess name="xsd:QName" targetNamespace="xsd:anyURL">
 <xs:participant name="xsd:string"
     processDefinition="xsd:anyURI"/>+
 <xs:globalCorrelationSet>?
  <xs:correlationSetName>…</xs:correlationSetName>
 </xs:globalCorrelationSet>
</xs:splitProcesses>
```

Table 10: <splitProcess> element tying the process fragments together

The process fragments do not contain enough information to relate their partnerLinks to each other's (i.e.: in Figure 26, participants *z* and *x* interact with the same participant *y*), or deterministically pick a connection for a process that offers/requires the same portType over multiple partnerLinks. Therefore, they must be wired together.

Rich global wiring models include SCA [BCII06]. While such models can be used, we focus here on a minimal wiring model that satisfies the simple, basic requirements at hand: partnerLink aliasing and locators to set up initial connections at deployment time. A locator is an item that provides or resolves to the address of a service (WSDL Port, WS-Addressing Endpoint Reference (EPR), etc.). Our wiring model consists of a set of pair-wise connectors where at least one party is a BPEL process. This is shown in the 'splitProcessDeployment' element in Table 11.
The 'splitProcessName' refers to the split process element this deployment descriptor is for. Each connector consists of two 'party' elements. Each party element contains optional 'local-name', 'participant' and 'locator' attributes. The 'local-name' is the local name the party uses for the connector (i.e.: a partnerLink name). The 'participant' is used if a party is implemented by one of the fragment processes. If this is the case, then its value must match the name of a participant in the corresponding 'splitProcess' element. The 'locator' attribute is used if the party is invoked by the other party in the connector. At least one of these three attributes must be present in each party element. At least one party in a connector must have a locator attribute present.

```
<xs:splitProcessDeployment splitProcessName="xsd:QName">
 <xs:import …/>+
 <xs:connector>+
   <xs:party local-name="xsd:string"? participant="xsd:QName"?
             locator="xsd:anyURL"? />
   <xs:party local-name="xsd:string"? participant="xsd:QName"?
             locator=" xsd:anyURL"?/>
 </xs:connector>

 <xs:coordination>?
 <!--  to be defined in the next chapter  -->
 </xs:coordination>
</xs:splitProcessDeployment>
```

Table 11: <splitProcessDeployment> element

A splitProcessDeployment element must contain connectors between the fragments as well as to external partners invoked by the unsplit process model. It will be used at deployment time to wire the fragments together. Note that since deployment in BPEL is by design implementation-dependent, the exact deployment mechanism will depend on the engine in which each fragment is deployed.

In addition to connectors, a splitProcessDeployment element also contains an optional 'coordination' element concerned with connections when coordination is used and therefore defined in the next chapter.

5.5.1 Identifying Instances

The process fragments of a split process must all run together. There may be several instances of the split process running simultaneously and, as a result, several instances of each fragment. Therefore, one must be able to avoid having wrong instances of different fragments interact with each other: One instance of each fragment of a split process constitutes a conceptual instance of that split process. One way to address this problem natively in BPEL is to use the language's correlation mechanism.

A single correlation set will be used in the unsplit process. This is named the 'global correlation set' and is identified in the 'splitProcess' definition. Its value serves as an instance identifier for the split process and is copied into all messages between the fragments. *In order to avoid repetition in the rest of this chapter, it is implicitly assumed that the correlation set value is copied into all messages sent between fragments along with any other data described in the corresponding sections containing inter-fragment messages.*

Once split apart, any interactions with outside parties will be routed properly because BPEL correlation tokens are part of the application data that the partner already knows about. The value is set by any of the starting receive activities, and maintained for the lifecycle of each instance.

Instance creation ability of added receive activities

In converting dependencies into message exchanges due to fragmentation, our approach adds new receive activities in the process fragments. In this section, we present the relation of this addition to process fragment instance creation. The details of creating the receive activities will be described in sections 5.8 through 5.10.

The algorithms must determine whether such a newly added receive activity can create an instance of the fragment, i.e. whether to set the value of its 'createInstance' attribute to 'yes'. A greedy approach is used to make this determination: Unless otherwise stated for a specific case of receiving block creation, the 'createInstance' attribute is set to 'yes' on a subset of the receive activities added to the fragments by the algorithms. The subset consists of those receive activities added for a split dependency whose target: (1) is not in a split loop (or split scope nested in the process itself), and (2) has no incoming link whose source is in the same fragment. For example, this will be the case in Figure 26 for the receive activities in the receiving blocks of activities *B, H, C, D,* and *F*.

This is a harmless overestimate: Only the first of the receive activities, whose createInstance is 'yes', to get a message can actually create an instance. Also, the replication of control dependencies from the unsplit process means that introduced receive activities in receiving blocks of activities that were downstream in the main process but have ended up at the top of a single partner's process will not create an instance. For example, in Figure 26, both *B* and *H* have no incoming links from other original activities in their own fragment. However, *H* will not happen before *B* even though the fragment, in isolation, seems to allow it. Analysis to reduce the number of receives that have this attribute set to 'yes' is possible but has no large effect on performance or execution behavior.

5.5.2 The Scope and Loop Relationship Tree

The coordinator requires two artifacts, created by the algorithms in this chapter, about split loops and scopes. The creation of these artifacts is not a task for the process modeler; they can be automatically derived at deployment time. They are:

- A scope and loop relationship tree, also referred to simply as the relationship tree, encoding the nesting and fragmentation information about fragmented loops and scopes and defined in this section.
- Default compensation order graphs encoding the information needed for the coordinator to determine the order of compensation for scopes in the relationship tree and defined in the next section.

The scope and loop relationship tree, $RT=(N_{rt} , E_{rt})$, consists of a set of nodes N_{rt} and a set of edges E_{rt}. N_{rt} is divided into four pair-wise disjoint sets, each containing a different kind of node. These sets are S_{rt} , S_{ns} , L_{rt} , and F_{rt}.

- S_{rt} , the set of split scope-nodes. Each scope node $s_{rt} \in S_{rt}$ is a tuple consisting of the name of the scope, a set M_s of names of faults that the scope has handlers for, a Boolean stating whether or not this scope is in the fault handler of its immediately enclosing scope, and a Boolean stating whether or not this scope is in the compensation handler of its immediately enclosing scope.

$$s_{rt} = (name, M_s, in\_fault\_handler, in\_comp\_handler)$$

- S_{ns}, the set of unsplit compensation-relevant scopes. An unsplit scope is only relevant for coordination if it is relevant for compensation. In turn, a scope is only relevant for compensation if there are explicit compensation handlers within it. Therefore, a node corresponding to an unsplit scope is *only* included in the tree if the scope's parent is split and the scope or any of its nested scopes have an explicit compensation handler. Each such node, $s_{ns} \in S_{ns}$ is a tuple consisting of the participant name, the scope name, and a Boolean denoting whether it has an explicit compensation handler.

$$s_{ns} = (p\_name, name, has\_compensation)$$

- L_{rt} , the set of split loop-nodes. Each loop node, $l_{rt} \in L_{rt}$, containing the name of the loop.

$$l_{rt} = (name)$$

- F_{rt} , the set of fragment-nodes. A fragment node represents one fragment of a split loop or scope. All fragment nodes are leaf nodes in the relationship tree. Fragment nodes are of two kinds, in the subsets of F_{rt} such that:

$$F_{rt} = F_{rt_s} \cup F_{rt_l}$$

  - F_{rt_l} , the set of loop fragments. Each loop fragment node, $f_{rt_l} \in F_{rt_l}$, is a tuple consisting of the name of the participant the fragment is in (*p_name*), the name of the loop to identify the loop it is part of, and a Boolean stating whether that fragment is responsible for evaluating the condition of the loop.

$$f_{rt_l} = (p\_name, loop\_name, is\_responsible\_for\_condition)$$

  - F_{rt_s} , the set of scope fragments. Each scope fragment-node, $f_{rt_s} \in F_{rt_s}$ is a tuple consisting of the name of the participant the fragment is in, a set O of fault names that this fragment of the scope (regardless of other fragments of the same scope) has handlers for, the name of the scope that this fragment is a part of, and a Boolean stating whether the fragment of the scope has an explicitly defined compensation handler. The fault names must be a subset of the set M_s of the scope-node the fragment is part of.

$$f_{rt_s} = (p\_name, O, scope\_name, has\_compensation) \text{ where } O \subseteq M_s$$

The relationship tree contains three kinds of directed edges: $E_{rt} = A_{rt} \cup B_{rt} \cup C_{rt}$. A_{rt}, B_{rt}, and C_{rt} are pair-wise disjoint.

- A_{rt}, the set of fragment-activity edges. Each fragment-activity edge, *a*, connects either a fragment of a loop to its corresponding loop-node, or a fragment of a scope to its corresponding scope node. Since the fragments of a scope are always 'scope fragments' and the fragments of a loop are always 'loop fragments', scope fragments cannot be linked to a loop-node via a fragment-activity edge and vice versa. We define *A* as:

$$A_{rt} \subseteq (F_{rt_s} \times S_{rt}) \cup (F_{rt_l} \times L_{rt})$$

- B_{rt}, the set of loop-scope edges. Each loop-scope edge, *b*, connects a loop to its parent scope. Each loop node is connected to exactly one scope-node with a loop-scope edge. Therefore,

$$B_{rt} \subseteq L_{rt} \times S_{rt}$$

- C_{rt}, the set of child-parent edges. An edge *c* in C_{rt} directly represents the nesting (child to parent) relations between scopes. BPEL scopes are strictly nested, so each scope node (besides the root) has one child-parent edge to its immediately enclosing scope in the unsplit process. Thus, a scope-node is connected to other scope-nodes by child-parent edges. A scope in a fault handler of another scope is considered the latter's child scope.

$$C_{rt} \subseteq (S_{ns} \cup S_{rt}) \times S_{rt}$$

The nesting of scopes in loops and loops in loops is not encoded in the relationship tree. Chapter 6 will show that the coordination protocols do not need this information.

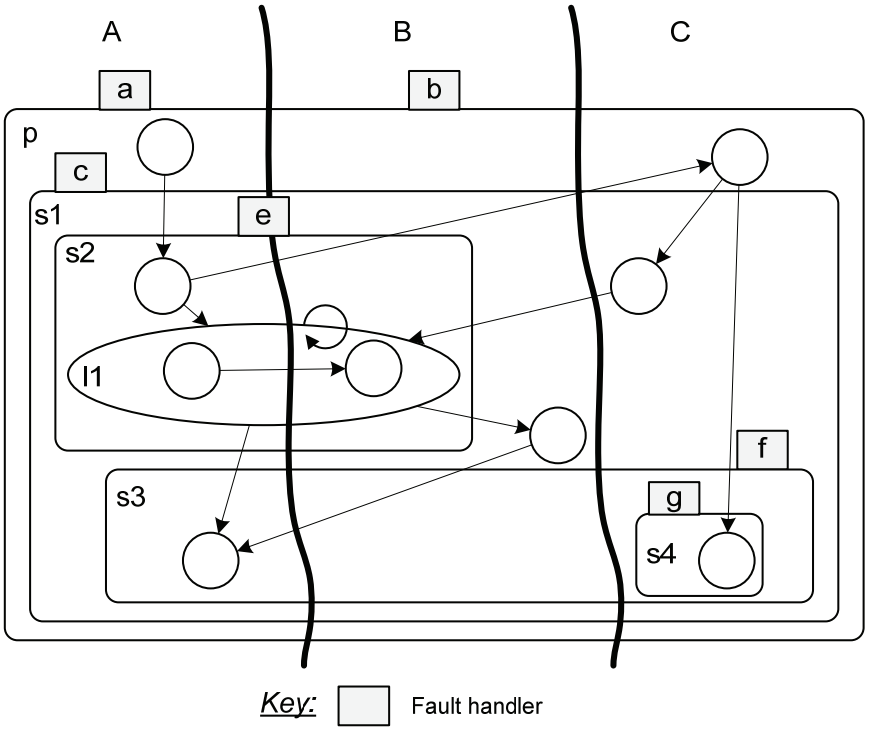


Figure 29: A process with a split loop and several split scopes

Figure 29 shows a sample process with several split scopes (*p,s1,s2,s3*) and a split loop (*l1*). Its corresponding relationship tree, assuming that the participants are named A, B, and C from left to right, is shown in Figure 30. Notice that *s4* does not appear in the tree, because it is an unsplit scope that does not have a compensation handler.

When Coordination is Used

Coordination will only be used if a process has a process level fault handler and/or it contains at least one split loop or scope (apart from the process itself, s_r). Thus, coordination is only used if a process's relationship tree does *not* satisfy the following:

$$S_{rt} = \{s_r\}, \pi_2(s_r) = \emptyset, |S_{ns}| = 0, \text{ and } |L_{rt}| = 0$$

In other words, it has exactly one split-scope node: the node corresponding to the process. The scope node has no explicit fault handlers (the set of fault names is empty). Additionally, the relationship tree has no unsplit scope nodes and no split loop nodes.

Generating the Relationship Tree

The relationship tree can be derived from a process defined using the meta-model in Chapter 3, as will be shown in this section. Assume that upon starting, all the sets of the relationship tree are empty. The scope nodes and child-parent edges are simply a projection of the graph of the hypergraph *G* of the meta-model. Recall that *G*'s nodes represent the scopes of a process and its edges represented parent-child relationships. Therefore, the nodes of the relationship tree's set S are a subset of the nodes of *G*, and the parent-child edges above are derived from the edges of *G*.

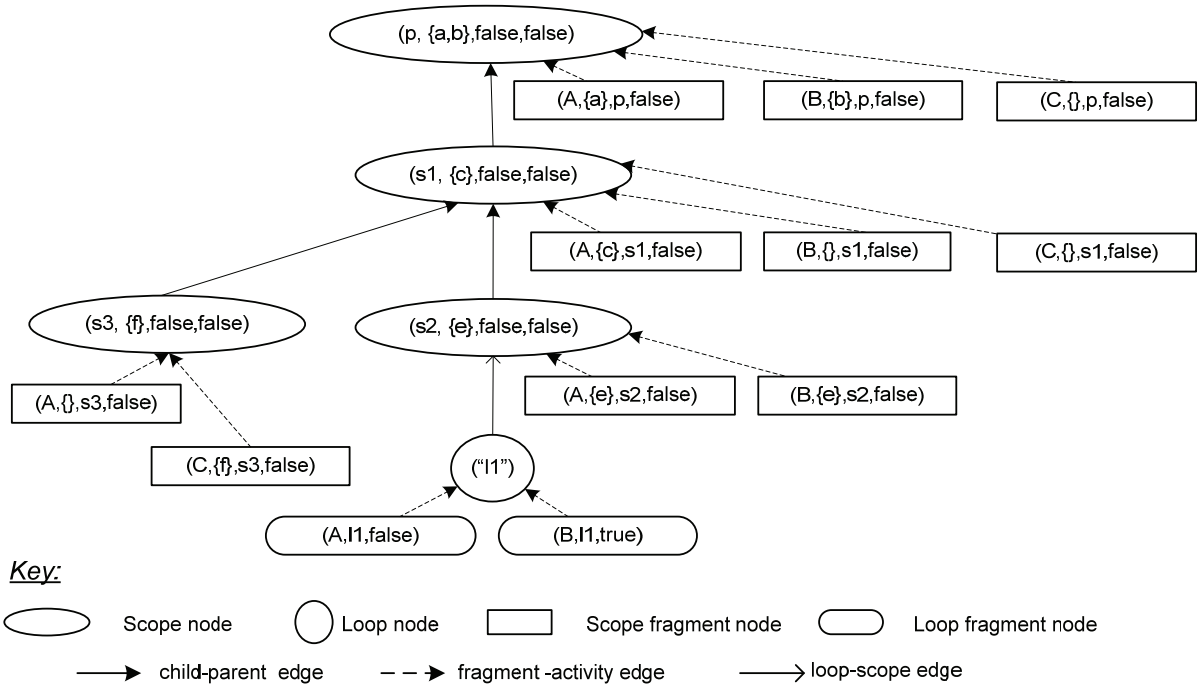


Figure 30: Relationship tree for the process in Figure 29

Chapter 3 showed how a BPEL process can be mapped to the process meta-model. If the meta-model is created from a BPEL process, some scopes created by the mapping to the meta-model should not be included in the relationship tree. These are the scopes added to handle the case where 'suppressJoinFailure' was set to 'yes' in the BPEL process (see 3.4.1). The reason is that the coordination protocols will handle the 'suppressJoinFailure' shortcut, so these scopes are not needed by the coordinator.

To be more precise, consider a process in the meta-model created from a BPEL process. Then for any scope, *s*, is in the process that is not the root scope and whose 'suppressJoinFailure' is set to 'yes' in the BPEL process, then (by construction) its parent in *G* is a scope that has no incoming or outgoing links, has exactly one activity that is itself a loop or a scope and has exactly one fault handler, where the handler is for the join failure fault and contains one empty activity. For every such scope s, the child-parent edge created by the projection will map to an ancestor scope in *G*. Consider the set of these join failure handling scopes to be S_{jf}.

A helper function that skips these scopes, when getting a parent scope in *G*, is defined as:

$$parent_{njf}(s) := \begin{cases} parent(s) & parent(s) \notin S_{jf} \\ parent_{njf}(parent(s)) & otherwise \end{cases}$$

The relationship tree does not include scopes not involved in the coordination or additional information about process details that the meta-model encodes (all included activities in a scope, etc). Therefore, the resulting projection excludes them to avoid adding additional data in the coordinator that it does not need.

Scopes

Every fragmented scope in the set of scopes of the process S has an equivalent element in S_{rt}. Consider S_f to be the set of fragmented scopes in a process model *P*, where:.

$$s \in S_f :\Leftrightarrow (s \in S - S_{jf}) \wedge (\exists a, b \in A(s), \{p_i, p_j\} \subseteq P_a : a \in \pi_2(p_i), b \in \pi_2(p_j), i \neq j)$$

The map S_M takes a fragmented scope and provides the corresponding scope node in the relationship tree.

$$S_M : S_f \rightarrow S_{rt}$$

Similar to Chapter 3, we use $[\![p]\!]$ to refer to the Boolean result of evaluating a predicate *p*. Recall from Chapter 3 that S_{comp} is the set of scopes that have an explicit compensation handler. The scope node corresponding to a scope *s* is thus created as follows:

$$S_M(s) = \left(name(s), M(s), \left[\!\left[A(s) \subseteq \bigcup_{f \in F(s_{p\_nj})} A(f)\right]\!\right], [\![s_{p\_nj} \in S_{comp} \wedge A(s) \subseteq N_{comp}]\!]\right)$$

where $s_{p\_nj} = parent_{njf}(s)$, N_{comp} is the set of activities of the compensation handler of this scope, i.e. the activities in $\pi_1(j(s_{p\_nj}))$.

Thus, the set of scope nodes is constructed by adding each resulting scope node to it:

$$S_{rt} = \bigcup_{s \in S_f} \{S_M(s)\}$$

We now address the unsplit scopes that need to appear in the tree. An unsplit scope is represented as an unsplit scope node in the tree if its parent is split and it or any of its nested scopes has an explicit compensation handler. We define a function that takes the process meta-model and the partition definition and returns a set containing the corresponding unsplit scope nodes for all scope in *S* that should be represented as an unsplit scope node in the relationship tree. Consider a set C_s that contains the scopes nested in *s* along with *s* itself, i.e. the nodes of the subtree of *G* rooted at *s*.

$$S_{ns} = nonsplitScopeNodes(P, P_a)$$

```
function nonsplitScopeNodes(P,Pa)
 S_ns = ∅
 for each s ∈ S − (S_f ∪ S_jf)
  if parent_njf(s) ∈ S_f ∧ ∃s_i ∈ C_s : s_i ∈ S_comp
     S_ns ← S_ns ∪ {(π_1(p), name(s), [[s ∈ S_comp]])}, where p ∈ P_a : A(s) ⊆ π_2(p)
 return S_ns
```

Next, we consider the edges of the relationship tree that connect parent and child scopes. Consider a set S_{o_ns} that contains all scopes in the process that are represented as unsplit scope nodes in the relationship tree, i.e. as a node in S_{ns}. Also consider a function $S_N : S_{o\_ns} \rightarrow S_{ns}$ that returns the corresponding unsplit scope node for a scope in S_{o_ns} .

We define a helper function that provides the scope tree node corresponding to each scope in *S* represented in the relationship tree.

$$S_{MN} : S_f \cup S_{o\_ns} \rightarrow S_{rt} \cup S_{ns}$$

$$S_{MN}(s) := \begin{cases} S_M(s) & s \in S_f \\ S_N(s) & s \in S_{o\_ns} \end{cases}$$

The set C_{rt} of child-parent edges between scopes is derived from the edges G that connect fragmented scopes. Therefore, we define the following map E_M that creates from an edge in G a set containing the corresponding edge in C_{rt}.

$$E_M((a,b)) := \begin{cases} \{(S_{MN}(a), S_{MN}(b))\} & \{a,b\} \subseteq S_f \cup S_{o\_ns} \\ \{(S_{MN}(a), S_{MN}(parent_{njf}(b))\} & a \in S_f \cup S_{o\_ns} \wedge b \in S_{jf} \\ \emptyset & otherwise \end{cases}$$

Due to the rubber-band effect, if an unsplit scope is represented on an edge in C_{rt}, then all children of that scope will be unsplit as well. Therefore, unsplit scopes form subtrees of G, whose root is represented in the relationship tree while they themselves are not. Thus, not representing these nodes in the relationship tree does not result in disconnected nodes and so does not affect its well-formedness. The set C_{rt} is created as follows:

$$C_{rt} = \bigcup_{e \in \pi_2(G)} E_M(e)$$

Fragment edges are derived directly (and only) from the partition definition. Consider a function F_{frag} that retrieves the set of fault handlers that are in a fragment of a scope:

$$F_{frag}(s,p) := \{f \in F(s) | \exists a \in A(s) : a \in \pi_2(p) \wedge a \in A(f)\}$$

Now we are ready to create the fragment nodes. Consider the partial function F_M that, given a fragmented scope and a participant, provides the resulting fragment scope node.

$$F_M : S_f \times P_a \rightarrow F_{rt_s}$$

$$F_M(s,p) = \Big(\pi_1(p), \bigcup_{f \in F_{frag}(s,p))} \pi_1(f), name(s), [\![s \in S_{comp}]\!]\Big)$$

F_M is defined for the subset of scope and participant tuples such that there is an activity in the scope of the tuple that belongs to the corresponding participant's activities. We use the notation from [PITT06] of using a '↓' to denote that a partial function is defined for a particular input:

$$F_M(s,p)\downarrow \;:= \begin{cases} true & (s,p) \subseteq S_f \times P_a, \exists a \in A(s) : a \in \pi_2(p)\} \\ false & otherwise \end{cases}$$

Thus, the set of scope fragment nodes is created as follows:

$$F_{rt_s} = \bigcup_{\{(s,p) | s \in S_f, p \in P_a, F_M(s,p)\downarrow = true\}} \{(F_M(s,p)\}$$

The edges joining a scope fragment node are determined as follows. If there is a fragment of s in a participant p, then A_{rt} contains an edge from that fragment to the scope node in the relationship tree corresponding to *s*.

$$A_{rt} = \bigcup_{\{(s,p) | s \in S_f, p \in P_a, F_M(s,p)\downarrow = true\}} \{(F_M(s,p), S_M(s)))\}$$

Loops

Consider W_f to be the set of fragmented loops in *P*. Thus, $W_f \subseteq W$, recalling that W is the set of loops in *P*.

$$w \in W_f :\Leftrightarrow w \in W \wedge \exists a, b \in \pi_2(w) : a \in p_i, b \in p_j, i \neq j$$

The map W_M creates for every fragmented loop a corresponding loop node in the relationship tree.

$$W_M : W_f \rightarrow L_{rt}$$

Consider *w* to be a loop in W_f. Then, the corresponding loop node is created as follows:

$$W_M(w) = (name(w))$$

The set of loop fragment nodes contains a node for each fragmented loop:

$$L_{rt} = \bigcup_{w \in W_f} \{W_M(w)\}$$

Next, we consider the edges of the relationship tree. We define a helper function E_{WM} that returns a set containing an edge connecting the loop node to its parent scope node, skipping any suppress join failure scopes. Recall from Chapter 3 that $K(w)$ returns the activity that represents the loop and that $s'_0(a)$ returns the immediate scope of *a*. Let:

$$s_w(w) = s'_0(K(w))$$

We define E_{WM} as follows:

$$E_{WM}(w) = \begin{cases} \{(W_M(w), S_M(s_w(w))\} & s_w(w) \subseteq S_f \\ \{(W_M(w), S_M(parent_{sjf}(s_w(w))))\} & s_w(w) \in S_{jf} \\ \emptyset & otherwise \end{cases}$$

The set B_{rt} of loop-scope edges is then created as follows:

$$B_{rt} = \bigcup_{w \in W_f} E_{WM}(w)$$

The loop fragment nodes are the last type of node to add to the tree. One such element is created for every split loop and every participant such that at least one activity from the loop belongs to the activities of the participant.

$$F_{rt_l} = \bigcup_{p \in P_a, w \in W_f, \pi_2(w) \cap \pi_2(p) \neq \emptyset} \{(\pi_1(p), name(w), [\![L_c(n_l) = p]\!])\}$$

The loop fragment nodes need to be connected to the loop nodes they are fragments of:

$$A_{rt} = \bigcup_{l \in F_{rt_l}, l_{rt} \in L_{rt}, \pi_1(l) = \pi_1(l_{rt})} \{(l, l_{rt})\}$$

Having defined the relationship tree, we move to the default compensation order graphs.

5.5.3 Default Compensation Order (DCO) Graphs

The coordinator needs to know the default compensation order of the scopes for each level of scope nesting so it can coordinate compensation in the presence of scope fragments. This order is encoded in what we call 'Default Compensation Order' graphs, or DCOs. BPEL's default compensation order involves only one level of scope nesting. In other words, the default compensation order is needed for peer scopes at each level. Scopes that have no explicit compensation handler and no nested compensation handlers can be ignored for this because compensating them is by definition a no-op.

DCOs are not directly derived from the COGs in the meta-model (section 3.6.2) because the compensation handling model of the meta-model differs from BPEL: the former is not limited to one level of nesting at a time. However, there are similarities in handling compensation in the presence of loops and keeping track of instances.

A few definitions are used for creating a DCO. We define:

- A function *type(a)* that returns the type of the activity *a*
- A function *isNested(l,s)* returns true if the scope *s* is nested in the loop *l* and false otherwise.
- A function *hasDirectPath(a,b,N)* that takes a scope or loop *a*, a scope or loop *b*, and a set of nodes *N*. It returns whether there exists at least one path of explicit BPEL links from the scope/loop corresponding to *a* (or any of its nested activities) to the scope/loop corresponding to *b*, (or any of its nested activities), that does not include any other activity *c* (or its nested activities), where $c \in N - \{a, b\}$.
- A function *getImmediateChildren(a)* returns the immediate child activities of *a* while treating flow activities as transparent, i.e. if a child is a flow then the flow's children are returned instead.
- A function *getChildren(a)* returns all the children, at any level of nesting, of *a.*
- A function *getHandlerActivities(s)* returns all activities in a fault or compensation handler of scope *s.*
- A function *hasCompHandler(s)* returns true if *s* is a scope that has an explicit compensation handler and false otherwise.

The default compensation order at each level is encoded using a set of graphs of child scopes, calculated as follows:

```
function CREATE_DCOS(ProcessModel proc, Participants Pa)
1   DCOS = ∅
2   loopDCOs = ∅
3   for each s where s ∈ S_rt ∧ ∃c_rt ∈ C_rt : s = π2(c_rt)
4      S1 = {a ∈ getImmediateChildren(s)|type(a) = SCOPE}
5      S2 = S1 − getHandlerActivities(s)
6      Sc = S2 − {a ∈ S2|hasCompHndlr(a) ∨ (∃b ∈ getChildren(a) : hasCompHandler(b))}
7      Lc = {l_rt ∈ L_rt|∃b ∈ B_rt : b = (l_rt, s)}
8      N_D ← (Sc ∪ Lc) − Sn
9      E_D ← ∅
10     for each (n1, n2) ∈ N_D : n1 ≠ n2 ∧ hasDirectPath(n1, n2, N_D)
11        E_D ← E_D ∪ {(n2, n1)}
12     G_D = (N_D, E_D)
13     DCOS ← DCOS ∪ {G_D}
14     for each n ∈ N_D : type(n) = LOOP
15        n_c = getImmediateChildren(n)
```

```
16      S_l = n_c ∩ S_c
17      C_l = {a ∈ n_c | type(a) = LOOP ∧ ∃s : s ∈ S_c ∩ getChildren(a)}
18      N_l = C_l ∪ S_l
19      for each (n_1, n_2) ∈ N_l : n_1 ≠ n_2 ∧_l hasDirectPath(n_1, n_2, N_l)
20          E_l ← E_l ∪ {(n_1, n_2)}
21      G_l = (N_l, E_l)
22      loopDCOS ← loopDCOS ∪ {G_l}
23  return {DCOS, loopDCOS}
```

Iterate over each non-leaf scope node in the scope tree (line 3). Consider the set of scopes S_c consisting of (lines 4-6) every child of *s* that is a scope (1) not nested in another scope, (2) not in a fault handler (3) not in the compensation handler of *s,* and (4) either it or any of its nested scopes has an explicit compensation handler.

Create the main compensation graph G_D of *s* with nodes N_D and edges E_D (lines 6-12). The elements of N_D are created so that only compensation-relevant nodes are kept(line 8-9): the scopes in S_c and every loop (line 7) that is an immediate child of *s*, contains scopes in the scope tree, and is not nested in another loop in S_c. However, the scopes in S_n, which are scopes nested in newly added loops, are removed (line 8).

The elements of E_D are created so that they reverse the control dependencies (lines 8-9) in the process and project them to only be concerned with the nodes of N_D.

Next, loops are handled: For every loop node in N_D, we create another graph, $G_l=(N_l, E_l)$. N_l contains immediate children of the loop that are members of S_c (line 16) and immediate loop children containing members of S_c (line 17). E_l contains the edges between the elements of N_l (line 19-20) in the same manner as the edges of G_D.

The result is a main compensation graph for each scope, as well as a compensation graph for each relevant loop.

Consider the sizes of the tree and the DCO graphs. These constructs are created at design time, once for each partition, and transmitted once to the coordinator for all instances of a split process. The number of edges of the relationship tree is the number of nodes minus one, because it is a tree. The number of nodes has an upper bound of $(num_{scopes} + num_{loops}) \times |P_a|$ number of scopes plus the number of loops in the process, multiplied by the number of participants. Its lower bound is zero, which is the case if the process is not split, i.e. $|P_a|$ is zero.

The Default Compensation Order graph is a directed acyclic graph (DAG). The number of nodes of this graph for each scope *s* is at most the number of immediate loop children plus immediate scope children. Each loop child in the DCO (or in a loopDCO) has a corresponding loopDCO whose number of nodes is at most at most the number of immediate scope children of the loop plus immediate loop children.

5.6 BPEL Language Extensions

Several language extensions are needed to indicate to the engine that a loop or scope is split and must therefore allow its lifecycle to be externally controlled by a controller that interacts with a coordinator.

These language extensions are used on the fragments and are in the 'iaas' namespace (http://www.iaas.uni-stuttgart.de). They are defined as follows.:

- On the process element:
  - The optional attribute *belongs-to="QName"* that specifies the name of the overall process. The name of the process (fragment) itself will be that of the participant. If a process has this attribute, it also indicates that the process is fragmented.
- On the scope element:
  - The attribute *fragmented="yes|no"* that denotes whether the scope is a fragment or not. The default value is 'no'.
- On the while element:
  - The attribute *fragmented="yes|no"* that denotes whether the loop is a fragment or not. The default value is 'no'.
  - The attribute *is-responsible="yes|no"* that specifies whether this fragment of the loop is responsible for the loop condition. The default value is 'no'.

Additional details of using these extensions are in [PALU07].

5.7 Creating Participant WSDL and BPEL definitions

In this section, we present the mechanisms by which the algorithm creates the WSDL definition and the BPEL process for each participant. We consider here the creation of the process structure and the placement of the original activities in these new processes. Additional activities and BPEL constructs that are added to transmit control and data dependencies between the fragments are presented in sections 5.8 and 5.9.

5.7.1 Creating the WSDL Definitions

A new WSDL definition is generated for each participant. Communication between fragments occurs using uniquely named operations as presented in sections 5.8 and 5.9. These operations, which we will refer to as 'connecting-operations', are the ones that will be used to transmit data and control dependencies between the fragments.

One new portType gets added to the WSDL definition of a process fragment for each other process fragment that it needs to receive communications from. One new partnerLinkType for each of these portTypes is added to the WSDL definition. This is a portType of a service offered by the process, i.e.: it is referred to in the 'myRole' attribute of a partnerLink element of the process.

So far, the WSDLs only reflect communications between the *newly* created processes. Next, we consider WSDL artifacts exposed to or exposed by existing partners of the unsplit process.

When a process is fragmented, one needs to consider the effects on its existing external clients. We will call the portTypes that the process offered to these partners (before the split) as its 'original portTypes'. A user may want to put the receive activity (or receive/reply activity pair) corresponding to different operations from the same portType in different fragments. If this is the case, then one would like to minimize changes on the clients while maximizing flexibility in the splitting:

- Minimizing the effect on the client involves restricting splitting original portTypes. That is, all receive and reply activities that refer to the same partnerLink must appear in the same fragment. We believe this will not be a hardship because portTypes are interfaces, grouping

closely related operations together. It is thus reasonable to assume that one fragment will get all the activities related to all the operations of one portType.

- Maximizing flexibility involves allowing splitting original portTypes. Repeat the partnerLink referring to this portType in every fragment that has a receive (or receive/reply activity pair) related to at least one of the portType's operations. A major drawback is that not all the operations will be supported by one fragment, violating the client contract provided by a portType. In other words, different fragments will offer subsets of the portType's operations. Once these fragments are deployed, then these subsets are offered at different endpoints. Thus, clients would have to be made aware of which endpoint corresponds to which subset of a portType's operations.

We currently support the first choice due to a trade-off between utility and complexity. Splitting a portType by separating its operations is an edge case. Therefore, to reflect the interactions with external clients on the fragments of the process, one simply needs to move the corresponding partnerLink from the main process to that fragment's BPEL process. The corresponding WSDL artifacts remain unchanged.

However, it is still possible to support splitting portTypes by following a pattern that increases communication but reduces impact on the clients: allow the designer to put the receive/reply activities referring to operations of the same original portType in different fragments but specify one fragment that will act as a gateway to all the operations of this portType. This fragment forwards a message from a client to the fragment where the original receive (and possible reply) activity has been placed and forward any matching reply back to the client. This is done by adding a sequence of a receive activity followed by an invoke activity (and possible reply) in the responsible fragment, and a receive (and possible reply) activity at the fragment where the designer placed the original ones.

5.7.2 Creating the Process Fragments

A new BPEL process is generated for each participant. The process has the same name as the participant and contains an 'iaas:belongs-to' attribute containing the name of the unsplit process. Then, the relevant partnerLinks, correlation set and variables are added to the process as shown in lines 1-10 of the CREATE_PROCESS_FRAGMENT:

```
function CREATE_PROCESS_FRAGMENT(Participant p, ProcessModel process)
1  proc = new Process(),setName(proc,π1(p))
2  setAttribute(proc, "iaas:belongs-to", process.getName())
3  for each partnerlink, pl, used by an activity in π2(p)
4    addPartnerLink(proc, pl)
5  for each participant pi ≠ p, and where pi and p interact
7    addPartnerLink(proc, new PartnerLink(pi,p))
8  for each variable v, used by an activity (incl. transition
                        conditions of its outgoing links) in π2(p)
9     addVariable(proc, v)
10 addCorrelationSet(proc, globalCorrelationSet)
11 for each child of getRootScope(process)
12   PROCESS_CHILD(child, proc, p)
13 ADD_EXTRA_FLOWS_AND_EMPTYS(proc)
14 return proc
```

PROCESS_CHILD (line 12) is detailed next. ADD_EXTRA_FLOWS_AND_EMPTYS (line 13) adds an empty anywhere an activity is needed by the syntax but was not provided, such as

in the body of a scope, and adds a flow anywhere a single activity is needed but multiple are provided, such as in the body of a scope or a handler or loop.

```
function PROCESS_CHILD(Activity child,ProcessModel proc,Participant p)
1  prnt = PARENT_LOOP_OR_SCOPE(child, proc)
2  prntFrg = GET_BY_NAME(proc, prnt.getName())
3  if isSimpleActivity(child) ∧ child ∈ π2(p)
4    anew= child
5  else if (∃a ∈ child : a ∈ π2(p)) ∧ type(child) ≠ FLOW
6    if type(child) = SCOPE
7      anew= new Scope()
8    else if type(child) = LOOP
9      anew= new Loop()
10   setName(anew , child.getName())
11   if ∃b ∈ child : b ∉ π2(p)
12     setAttribute(anew , "iaas:isFragmented", "yes")
13 if (type(prnt) = SCOPE) ∧ ∃h ∈ handlers(prnt) : child ∈ h)
14   GET_OR_CREATE_HANDLER(prtFrg, h).addActivity(anew)
15 else addActivity(prtFrag, anew)
16 if !isPrimitiveActivity(child)
17   for each activity ∈ (child ∩ π2(p))
18     PROCESS_CHILD(activity, child, proc, p)
```

The PROCESS_CHILD function determines for each child activity whether and how to map it to the target fragment's process. If the child is in or has activities that are in the fragment, then it must be placed in that fragment's process - more specifically, in its first loop or scope parent, found by the helper function PARENT_LOOP_OR_SCOPE. The GET_BY_NAME helper function takes an activity from the main process and returns the equivalent activity in the fragment's process if any is found.

If the child is a primitive activity and in the fragment, then add it to the parent fragment (lines 4-5). Upon encountering a scope or loop (lines 7-14), it needs to create a new scope or loop in the fragment's process, name it the same as the original. If it also has activities in another participant then it is fragmented. Therefore, set its fragmented attribute to yes.

The placement of the activity will be either in the body or the handler of its parent (lines 13-14). Finally, recurse and process the children of the current node (lines 16-18). Note that the activity must be added into the process before the recursion can happen so that *prntFrg* is not null on the next iteration. Hence, lines 13-15 precede the recursion.

If coordination is *not* used, a basic fault handling pattern (not shown in the pseudocode above), similar to [GOCS04] and illustrated in [KHAL07A], ensures that instances do not hang if a fragment process fails: a catch-all fault handler is defined on each fragment process. This handler has a flow of invoke activities that notify all the other fragment processes in case of a fault. Correspondingly, event handlers are defined on each process fragment to terminate the instance upon receipt of such a fault message.

Links will be added/modified and activities for inter-process communication will be added to the fragment processes, as described in the next sections of this chapter.

Placement of Links and Inter-fragment Communication Activities

Upon fragmenting a compound activity, i.e. a scope or loop, the algorithms must determine which fragments become the sources and targets of any control links targeted to or leaving

from that compound activity. Additionally, the sending and receiving blocks created to transmit dependencies between fragments must be placed in the appropriate parent BPEL construct for the fragments to behave correctly.

Links from/to split activities

The algorithms must choose which fragment of a loop or scope to place the target/source of a link that has the loop or scope itself as its source/target. To describe how the choice is made, we consider first the outgoing links and then the incoming links. Consider Figure 31: The top part of the figure shows an unsplit scope, and the bottom part shows four ways of splitting it. The outgoing link in question is the one from the scope to *d*.

For outgoing links, the fragmentation algorithms determine the fragment on which to actually place the link so as to reduce inter-fragment communication:

- If the target of a link whose source is a scope/loop belongs to a participant A, where A also contains a fragment of the scope/loop, then the source of the link is the fragment of A. An example is in cases 1 and 3 of Figure 31.
- If the target of a link whose source is a scope/loop does not belong to any participant containing a fragment of the scope/loop, then the fragment that is the source of the link will be the one that needs the least amount of data from other fragments in order to evaluate the link's transition condition. If there is more than one such fragment, then the choice between them is arbitrary. Cases 2 and 4 of Figure 31 show an example.

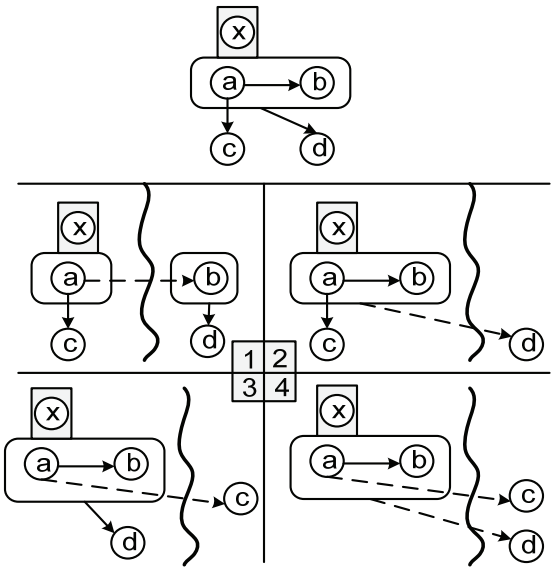


Figure 31: Options for outgoing links

Next, consider incoming links. Figure 32 summarizes the options for their placement: activities *a* and *b* link to a scope, with join condition *jc*, containing activities *c* and *d*.

One option is to have only one fragment as the target of all the links, as shown in cases 1 and 3 of Figure 32. The algorithm would choose the fragment containing the most activities that are the sources of these links. If there is a tie or the sources all belong to different fragments, then the owner of the link's target is chosen arbitrarily from the tied participants for the former case and from all participants owning a fragment of the loop/scope in the latter case. Here, it is clear when and where the join condition gets evaluated. However, a drawback is that it may require more links to be broken amongst participants.

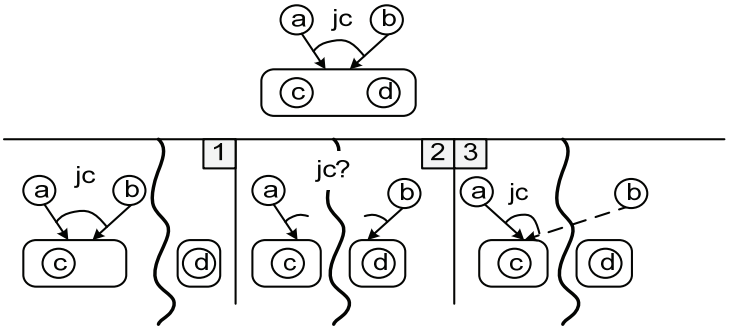


Figure 32: Options for incoming links

Another option is to choose, as the link's target fragment, the fragment that is in the same participant as the link's source activity (case 2 of Figure 32). If the split link's source does not belong to any of the participants that have fragments of the target loop/scope, then one of the participants with a fragment is chosen arbitrarily to be the link's target.

The problem with the second option is two-fold: the evaluation of the join condition would have to happen out of band (i.e. by the coordinator), including starting the join failure mechanism. We therefore favor the first option: one participant has the join condition and is the target of the incoming links. The coordination protocols and system handle this option for incoming links, but they can also handle the second option provided that the original condition can be (and is) refactored as the *'and'* of each local join condition of each fragment.

'Receiving block' and 'sending block' placement

Special attention needs to be given for split dependencies that cross a split scope: The placement of the sending block is critical. Consider the link *(a,c)* in Figure 31. If *a* faults and the handler *x* catches the fault and handles it, the link *(a,c)* must fire negatively and the link from the scope to *d* fires with the value of evaluating its transition condition. Now consider specifically cases 3 and 4 where the source *a* and target *c* are in different participants: if the sending block was placed in the scope on the left side (that also contains *a*) and *a* faults, then any invoke activities in the sending block would be disabled. As a result *c* would hang because its receiving block would never get its incoming message. The following property must thus be satisfied to ensure that sending blocks are not incorrectly disabled in the presence of dead-path elimination and faults.

Property: a cross-scope dependency in the main process has to stay a cross-scope dependency across the partition.

Due to the rubber-band effect and the fact that the process is itself treated as a scope, there will always be at least one common ancestor scope between any two activities in a process. A sending block as therefore placed as follows:

- If the participant containing the source of the control/data dependency also contains a fragment of the target's immediately enclosing scope, place the sending block in the immediately enclosing scope of the target;
- Otherwise, place it in the first common ancestor scope between the source and the target.

The receiving block is placed in the scope of the target activity, in the fragment in which that activity was placed in the partition.

5.8 Fragmenting Explicit Control Dependencies

Control links constitute the explicit control dependencies in a process. Consider the control link, *l(a,b,q)* in the unsplit process model in Figure 33. The link is between activities *a* and *b* and its transition condition is *q*. Partition the process such that *a* is in Participant 1 and *b* is in Participant 2. The transformation for sending control, conceptually, is to transform the link into a message exchange: a sending block linked from *a* in Partition 1 and a receiving block linked to *b* in Participant 2 such that the data sent is the status of the original link from *a* to *b*. The status will be either the value of evaluating *q* or simply false due to a fault or DPE. This status is then set as the transition condition on the link in Participant 2 from the receiving block to *b*.

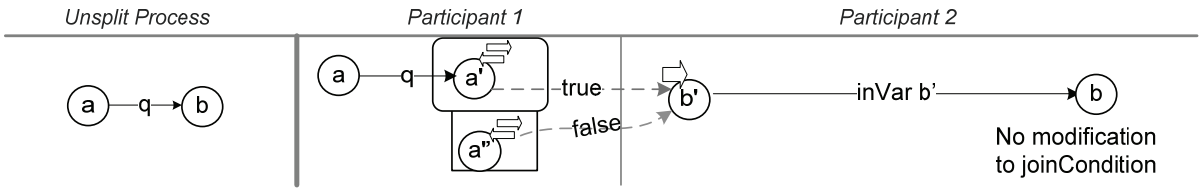


Figure 33: Splitting a control link across BPEL processes

Consider the specific issues of doing this in BPEL. The relevant BPEL activities are created (Figure 33, Participants 1 and 2), to ensure that that the link status is properly propagated. The sending activity is a BPEL invoke and the receiving one a BPEL receive. However, one can only read the status of a BPEL link in the 'joinCondition' of the link's target activity which leads to the pattern for the sending block described next.

The 'explicit control sending block', whose pattern is shown in Participant 1, is explained first. To propagate the link's status, a scope *s* is used. Scope *s* contains an invoke activity *a'* which is the target of the link from *a*, *l(a,a',q)*. Additionally, *a'* has *suppressJoinFailure='no',* and its input variable has the value 'true'. Scope *s* has a fault handler *f('joinFailure',{a''},s)*, where 'joinFailure' is the name of the fault caught by *f,* and *a''* is an invoke activity whose input variable has the value 'false'. Both invoke activities call the same operation.

The 'explicit control receiving block', shown in Participant 2, receives the message from Participant 1 in the receive activity *b'*. Consider *inVar b'* to be the variable of *b'*, i.e. the variable whose value is the message received from Participant 1. A link, *l(b',b, inVar b')* with the same name as the link in the unsplit process between *a* and *b*, connects the receive activity to *b*. The join condition of *b* is not modified.

Consider what happens at runtime: If *a* fails, then the link *l(a,a',q)* will fire with false causing the join condition of *a'* to be false. Therefore, the status of the original link *l(a, b, q*) is whether or not there was a join failure at activity *a'*. To pass this information to Participant 2, we use a fault handler on *s* and set 'suppressJoinFailure' attribute on *a'* to 'no'. At runtime, if the join condition of *a'* is true then *a'* propagates a message containing 'true' to *b'*. On the other hand, if it is false, then the fault handler catches the join failure fault and *a''* propagates a message containing 'false' to *b'*. The corresponding behavior reaches *b*, due to the condition of the control link *l(b', b, inVar b').*

If an activity *b* is the target of multiple links, the links between the receive activities and *b* in the receiving block(s) of *b*'s fragment partition are created so that they have the same name

as the corresponding links in the unsplit process model. As a result, the join condition of *b* is not affected in this approach.

An operation, with a unique name within each local process, will be added for every link in the unsplit process whose source and target activities are assigned to different participants. For *l(a,b,q)* above, the operation is added to the definition of the portType used by Participant 2's role in the partnerLink L between Participant 1 and Participant 2. The portType of Partition 1 is not affected.

We have shown how to split explicit control dependencies. In the rest of this chapter, we address splitting data dependencies.

5.9 Fragmenting Explicit Data Dependencies: BPEL-D

Explicit data dependencies exist when there is explicit data flow, i.e. BPEL-D data links. The patterns of constructs created to propagate the dependency represented by a split data link are presented in this section

An operation, with a unique name within the fragments of the operations, will be added for every data link in the main process where the source and target activities are assigned to different partners.

5.9.1 Splitting a Data Link Between Two Activities

The splitting of an explicit data link is handled at the source activity's fragment by sending to the target's fragment both the data itself and the source activity's completion status: If the source activity completed successfully, send 'true' and the data. Otherwise, send 'false' and 'null' (left side Figure 35a).

Recall that in BPEL-D the winner in case of multiple writes to the same location is chosen at random and that if a data link's source activity is disabled then its data does not reach the target activity's input containers. Therefore, the receiving block writes the data to a location visible to the target activity of a split data link only if the status, in the received message, of the source activity is 'true'.

Consider the top row of Figure 34: Assume that *b* and *d* are placed in different participants and that the condition from *a* to *c* evaluates to true. If the condition on the control link from *a* to *b* evaluates to false then *b* is disabled and *c*'s data should be used by *d*. If one did not make provisions for source activities being disabled by DPE, then the process might incorrectly overwrite the valid data sent from a source activity (*c*) that ran successfully with 'null' sent by a data link whose source activity (*b*) was disabled. Additionally, one must use the status of the data link's source activity to ensure that the sending block is not disabled if the source activity is disabled due to DPE.

The middle row in Figure 34 shows why one cannot combine the message for sending data with a message used to send control if the target is shared: While there is a data link and a control path from *b* to *e*, there is no direct control link between them. One might consider making *d* responsible for collecting the data from *b* and *c* and then sending it. However, *d* itself may be disabled by DPE. The bottom row in the figure shows yet another case where sending the data directly from the source of the data link to its target is preferred, as otherwise it is not clear whether one send the data with the control link from *c* or *d* or both.

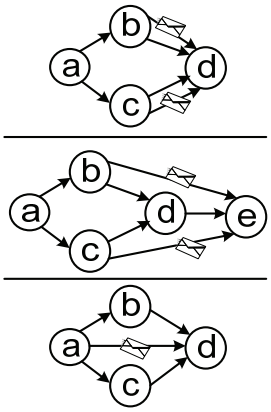


Figure 34: Examples of data links

Our solution needs to ensure *that a data link, from a source activity that is disabled, does not write to a value seen by the target of the data link.* In practice, this corresponds to adding a new activity after the receive activity and before the actual target activity. Figure 35 shows the patterns for splitting a data link between two activities, *a* and *b*. If the original source activity (in Participant 1) fails, the newly added assign activity (in Participant 2) would be skipped. The join condition of the target activity is modified to disregard the status of the new link from this assign activity. The reason is that, unlike a control link, a data link does affect whether or not its target activity can run.

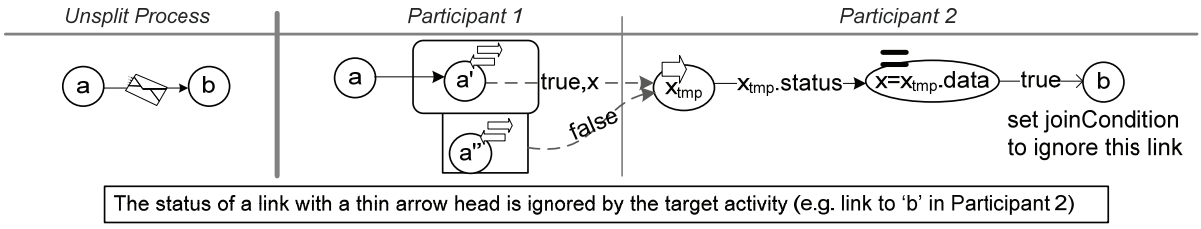


Figure 35: Splitting a data link across BPEL processes.

Consider the details of the generated BPEL constructs for the 'explicit data sending block': create in the fragment of the source activity *a* (Participant 1) a scope *s*, containing an invoke activity *a'*. A link *l(a, a', 'true')* is created from *a* the *a'*. Activity *a'* has suppressJoinFailure='no', and its input variable has a part with the value 'true' and a part containing the data to be flowed by the data link. The scope *s* has a fault handler *f('joinFailure', a",s),* where *a"* is an invoke activity whose input variable has a part with the value 'false' and a part with the value 'null'. Both invoke activities call the same operation on Participant 2.

Next consider the 'explicit data link receiving block', whose pattern is shown in Participant 2. The message is received by a receive activity. An assign activity copies the data needed by *b* into the appropriate parts of its input variable as specified by the data map on the data link *d(a,b)*. Links connecting these activities are placed in Participant 2 in Figure 35. The join condition of *b* is changed so it ignores the status of its newly added incoming link: set the join condition to the conjunction of the original join condition with the disjunction of the status of the new link and the negation of that status. For Figure 35, and the rest of the figures in this chapter, a control link whose status is ignored in this manner by the join condition is illustrated using a thin arrow head (see the link whose target is *b*) as opposed to a solid arrow head (see the link from *a* to *a'*).

Here, the fault handling is used to propagate whether *a* was successful or not instead of the status of a control link. The differences between splitting a data link and splitting a control link are: for data links additional data is sent in the positive case, an assign activity is used between the receive activity and the original target activity, and the join condition of the target activity is modified.

BPEL-D data links whose source and target activities are placed in the same partition are replaced with an assign activity. A link is placed from the data link's source activity to this assign, and another link from this assign to the data link's target activity. The join condition of the target is modified as for *b* above. Variables are added for each activity container and the activities are modified to refer to variables instead of containers.

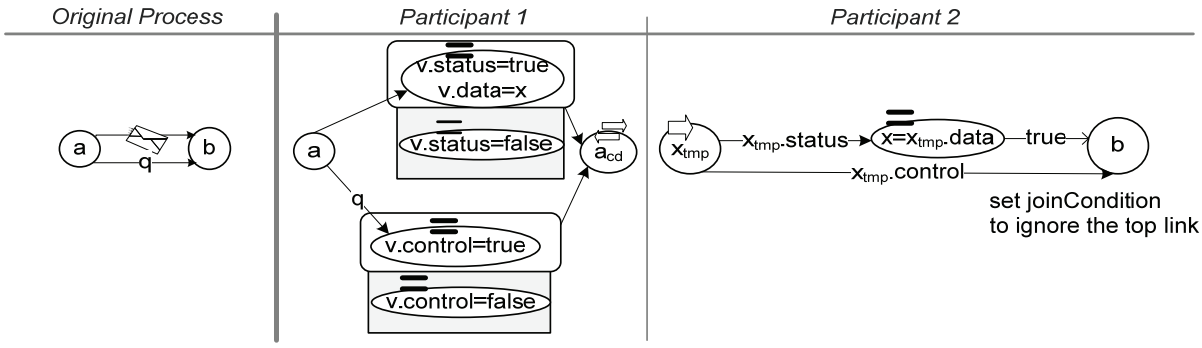


Figure 36: Splitting a data and a control link with the same source and target

Merging Sending Blocks from the Same Activity

If a control link and data link have the same source and target activity, then an improvement can be made to send one message instead of two. An example is shown in Figure 36: Two scopes are still used to determine the status of the control link and whether or not the source activity completed successfully, but the scopes use assign activities not invoke activities. The assign activities assign the state and the data (if needed) and a single invoke activity is used that sends the combined information. One modified receiving block is used in the target's fragment, with a link from the receive to the target activity for flowing control and another path to the target that includes an assign activity for flowing data.

Additionally, multiple explicit data sending blocks from the same activity can be merged by merging the scopes of their sending blocks into one scope that has a fault handler for the 'joinFailure' fault: The body of the new scope contains a flow activity that is the target of the link from the writer, instead of the invoke activities in the explicit data link sending block pattern. Every invoke activity in the body of the scopes of the separate blocks is placed in this flow. Every invoke activity in the fault handlers of the separate blocks is placed in a flow in the fault handler of the new scope. Examples of this will be shown in Figure 38 for the split loop in Figure 37: sending activity *M*'s data and sending activity *P*'s data to the fragments containing activities *N* and *O*. In Figure 38, dashed lines show invoke-receive message relations.

The End-of-split-activity Data Receiving Block

When data needs to be collected at the end of a split compound activity (such as data needed between loop iterations, after a loop, or for a compensation handler), we will use a modified version of the explicit data receiving block that will receive and assign the data at the end of a scope or the end of an iteration of a loop. This will enable it to be read in the subsequent

iteration or when a compensation handler runs. We call this modified version the 'end-of-split-activity data receiving block'.

The modification is as follows: the assign activity is not the source of a link. Additionally, a link is placed to the receive activity of the receiving block from each of the activities, in the fragment of the scope/loop in which the receiving block is being placed, that have no outgoing links. The join condition of the receive activity is set to ignore the status of all these incoming links. An example is shown in Figure 38: sending *P*'s data to the fragment containing activities *N*.

5.9.2 Data in Split Loops

Loop containers and their associated data links provide the information to determine the data needed by subsequent loop iterations and which activities produce/consume it. An example of a BPEL-D loop is shown in Figure 37, where it is assumed that *N* and *P* write to the output containers data needed in subsequent iterations by *N* and *O*, and that only *P* writes the data used by *Q*.

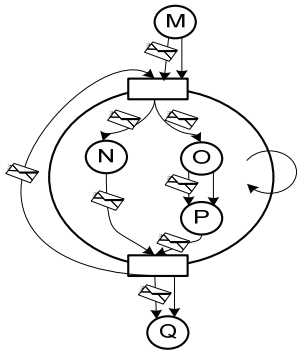


Figure 37: A BPEL-D loop

Data from Outside the loop

Data needed from activities outside the loop gets sent to every fragment containing an activity that will read this data. The receiving block is placed before the loop, and the target of the receiving block's link is the loop itself. Figure 38 shows examples of sending data from the process fragment containing activity *M* to the fragments containing activities *N* and *O* of the split loop in Figure 37.

Data for Subsequent Iterations

Data needed by subsequent iterations of a loop will be sent by the producing activities at the end of an iteration. To ensure this, we place one additional control link to the scope of the sending block from each of the activities in the sender's fragment that are in the loop and have no outgoing links whose targets are also in the same loop. The join condition of the scope is then set to ignore the values of these links.

An activity *a* in the body of a loop is determined to produce data needed by an activity *b* in the body of the loop during a subsequent iteration if all the following conditions are met:

- There is a data link mapping from containers of *a* to containers in a subset *D* of the output container set of the loop.
- There is a data link mapping from containers in *D* to containers in a subset *E* of the input container set of the loop.
- There is a data link mapping from containers in *E* to input containers of *b*.

If all of the above are true and *a* and *b* are in different fragments, then we need to create corresponding data sending and receiving blocks. We place an 'end-of-split-activity data receiving block' in the fragment of the loop where the target activity of the data link is. The data sent will be the value of the needed output containers of *a* and the status of *a*. One data link from *a* may result in several sending/receiving blocks. This will occur if it is determined that an iteration-datalink creates writes needed by more than one activity in the next iteration. If this is the case, the sending blocks are merged as in section 5.9.1.

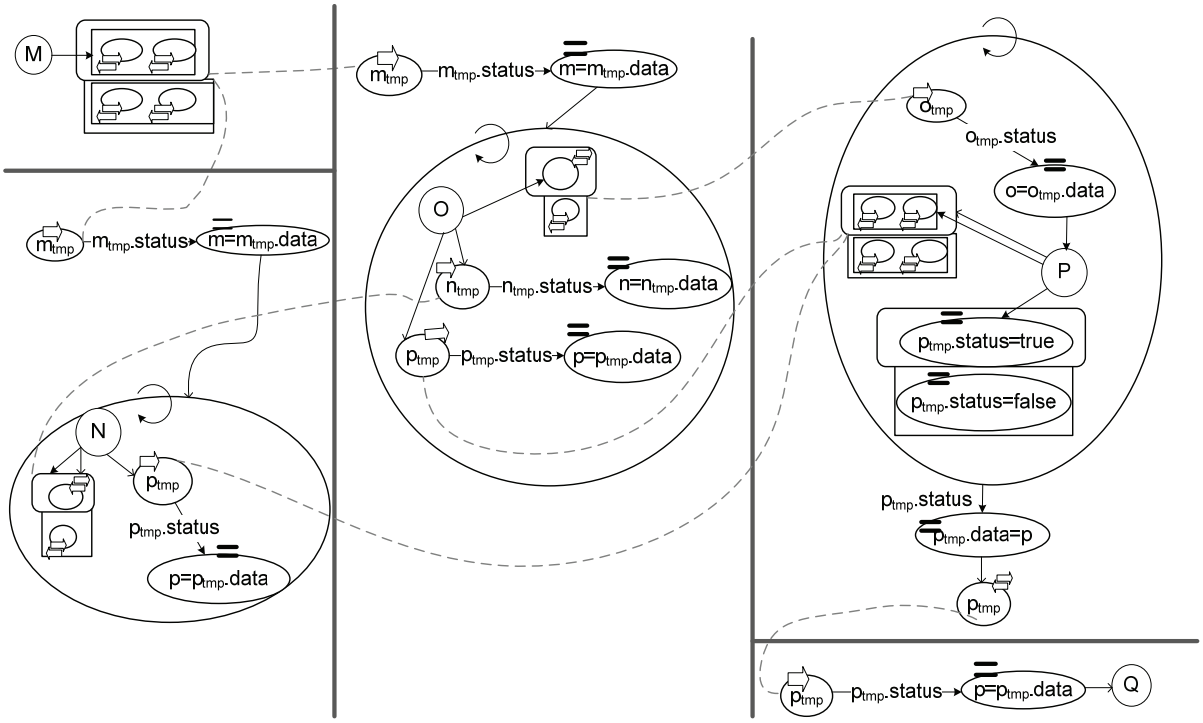


Figure 38: Data blocks from splitting the loop in Figure 37 so each activity is in a different fragment

Data for the Loop Condition

Only one fragment is responsible for the loop condition. Therefore, the fragment responsible for the condition is the reader. It needs the data before the loop runs and if it is modified inside the loop then it needs it again before the next iteration starts. Therefore this data flows the same way as data needed before the loop and data needed between iterations.

Data from the Loop to Activities after the Loop

Figure 38 shows the structures created for sending data from an activity inside a loop to an activity outside the loop and in another fragment: sending data from activity *P* in the split loop to activity *Q*.

The status of the writer needs to be determined inside the loop because links cannot cross a loop boundary; it also needs to be initialized to false in case the loop is never entered. However, the data must only be sent after the loop has completed.

The sending block is therefore modified from the explicit data sending block pattern as follows: assign activities are used instead of invoke activities in the scope, which is placed inside the loop. They copy the status of the activity into a variable used in the invoke activity that will send the data. After the loop completes, the data itself is copied into the variable

using another assign activity. This second assign activity is followed by an invoke activity that will send the data to the reader. The join condition of the invoke activity is set to ignore the status of its incoming link. Both these activities must be in the same scope and level of loop nesting as the reader. The second assign activity is linked from the outermost loop that does not contain the reader, i.e. the loop in which the writer is nested but the reader is in the parent scope or loop. The transition condition of the link from the loop to the second assign activity is set to be the status of the writer.

Copying the data in the second assign instead of inside the body of the loop saves unnecessary data copying. Split control links leaving from the loop boundary still have their sending blocks linked from the loop itself and not from this assign because target activities can only run once they have received both their control signals and their data. Additionally, there are no shared variables in BPEL-D so no other activity could have written to the same variable between the time the time the loop completed and the newly added assign activity ran.

5.9.3 Data to a Fault Handler

Data to a fault handler in BPEL-D may only come from the faulting activity. This will be sent to the scope fragments with the fault handler through the coordination protocols for handling split scopes. No sending/receiving blocks are created for it.

5.9.4 Data to a Compensation Handler

Similarly to data between loop iterations, there is a level of indirection between the source of a data link to a compensation handler and the actual activity or activities that need to receive the data: a compensation handler operates on the snapshot of the data that the scope had when it completed and the handler can only run if its associated scope has already completed.

An explicit data link sending block (see section 5.9.1) is created for every activity, *a*, that is the source of a data link to the input containers of a compensation handler of a split scope and for which both of the following two conditions hold. Assume the data link copies into a subset *D* of the handler's input container set:

- The compensation handler is in another fragment or is split between the fragment of *a* and one or more other fragments.
- There exists a data link from at least one input container of the compensation handler to an activity, *b*, where *b* is in another fragment and the data link copies from a subset of *D* into a subset *E* of *b*'s input container set.

For every corresponding reading activity *b*, we place an 'end-of-split-activity data receiving block' in the body of the scope fragment on which the compensation handler containing *b* is defined. Multiple data sending blocks from the same activity are merged as in section 5.9.1.

5.10 Fragmenting Implicit Data Dependencies: Standard BPEL

Standard BPEL uses shared variables for data flow, making data flow implicit. Fragmenting implicit data dependencies reuses the basic patterns of fragmenting explicit data dependencies, but involves more complexity for two main reasons: First, data analysis of the process is required in order to identify the existing data dependencies and second, BPEL's parallelism and dead-path elimination make the resolution of races between writers non-trivial. After explaining the mechanisms needed for splitting these dependencies, we provide a detailed example in section 5.10.10.

5.10.1 Determining Data Dependencies

In order to split data dependencies between BPEL activities, one needs first to describe how the necessary data dependencies are captured and encoded. Mainstream data flow analysis techniques are presented in [MUCH97] and [AHSU06], but BPEL presents special challenges due to parallelism and especially Dead-Path-Elimination. The application of the Concurrent Single Static Assignment Form (CSSA, [LEMP97]) to BPEL is shown in [GODL06] [MMGA07]. The result of the CSSA analysis is a possible encoding of the use-definition chains, where the definitions (write) of a variable for every use (read) are stated. Thus, the CSSA form can be transformed to provide a set of writers for each reading activity which can be in turn used as one of the inputs to our approach. Any data analysis algorithm on BPEL is usable provided it can handle dead path elimination, parallelism, and provide the result explained below. Therefore the details of such an algorithm are not directly in scope for this work.

We point the interested reader to one such data flow analysis algorithm for BPEL in [KOKL07]. It was created for this approach. It also details how to create the sets described in this section.

One challenging area is the handling of writes to different parts of a variable. Complexities introduced by query languages injected into BPEL are out of our control, but we do handle the common pattern: multiple queries of the form that select a named path (i.e.: (/e)*, called *lvalue* in the BPEL standard) and do not refer to other variables. For example, consider that w_1 writes *x.a*, then w_2 writes *x.b*, then *r* reads *x*; *r* should get data from both writers and in such a way that *x.b* from w_1 does not overwrite *x.b* from w_2 and vice versa for *x.a*. However, if they had both written to all of *x*, *r* would need *x* from just w_2. On the other hand, whether an activity reads all or part of a variable is treated the same for the purposes of determining which data to send.

The data flow analysis algorithm result should provide (directly or after manipulation) for each activity *a*, and variable *x* read by *a* (or any of the transition conditions on *a*'s outgoing links), a set $Q_s(a,x)$. $Q_s(a,x)$ groups sets of queries on *x* with writers which may have written to the same parts of *x* expressed in those queries by the time *a* is reached in the control flow. As in [KOKL07], the representation of a query does not include the variable itself, i.e. '.a' instead of 'x.a', and ϵ is used indicate the whole variable, i.e. 'x'. In the algorithms that follow, we treat ϵ as the empty string.

Consider w_1, w_2, and w_3 that write to *x* such that their writes are visible to *a* when *a* is reached. Assume they respectively write to $\{x.b, x.c\}$, $\{x.b, x.c, x.d\}$, and $\{x.d, x.e\}$, then $Q_s(a,x) = \{(\{.b, .c\}, \{w_1, w_2\}), (\{.d\}, \{w_2, w_3\}), (\{.e\}, \{w_3\})\}$. Thus, $Q_s(a,x)$ is a set of tuples, each containing a query set and a writer set.

The writers in this set have three properties: (1) if *a* is in a loop, then all the writers must precede *a* in the loop and belong to the same loop, (2) writers that are in a loop not containing *a* are represented in $Q_s(a,x)$ as one writer. This writer represents the largest loop containing these activities that does not also contain *a*. In other words, loops are collapsed to a single node. (3) If *a* is in a compensation handler, then all the writers must also belong to this compensation handler. Additionally, each query set contains the largest queries which these writers write to. For example, if w_3, w_4 both write to *x.b* and *x.b.e*, then the resulting tuple would be $\{(.b), \{w_3, w_4\}\}$.

The data algorithm should also be able to provide (directly or after manipulation) the following sets that are variations on Q_s such that the writer sets differ as follows:

- For each loop *l* represented as a writer in a $Q_s(a, x)$ where *x* is a variable read by *a* and *a* is any activity not nested in *l,* a set $Q_s^{postloop}(l, x)$ whose writers are all writers in the loop whose writes reach *a*.
- For each loop *l* and each variable *x,* a set $Q_s^{preloop}(l, x)$ whose writers are all writers not nested in the loop that are read by any activity nested in the loop.
- For each loop *l* and each variable *x* read by an activity in *l,* a set $Q_s^{intraloop}(l, x)$ whose writers of *x* are only those which are nested in *l* and not in any other loop *l'* in *l* and whose writes reach the next iteration of *l*.
- For each compensation handler *h* and variable *x* read by an activity in *h,* a set $Q_s^{prehandler}(h, x)$ whose writers are not in the compensation handler and whose writes reach activities in the handler.

Consider $A_d(a, x)$ to provide the set of all writers that *a* depends on for a variable *x* that appear in $Q_s(a, x)$. In other words, $A_d(a, x) = \bigcup_{q_s \in Q_s(a,x)} \pi_2(q_s)$.

5.10.2 Design Considerations

It would seem that one could create data links whose sources are the elements of A_d and whose target is *a* and then proceed as for fragmenting explicit data dependencies. While for some cases that is true, we will show that the data sharing behavior exhibited by BPEL's shared variables, parallelism, and DPE requires a more sophisticated approach.

Consider the process snippet in Figure 39 with activities *a, b, c, d,* and transition conditions *p, q* and *r.* Activities *a* and *b* write *x*. Activity *c* reads *x* and writes *y*. Due to DPE, *c* may run even if *b* is disabled. Therefore, the value of *x* that reaches *c* may be either 20 or 10, as shown in Table 12.

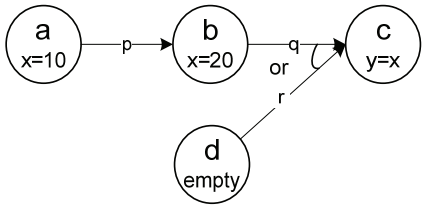


Figure 39: Example of BPEL's data flow in the presence of DPE

Executed activities, in order	*p*	*q*	*r*	*x at tf*
({(a,b),d},c)	T	T	T	20
({(a,b),d},c)	T	T	F	20
({(a,b),d},c)	T	F	T	20
({(a,b),d})	T	F	F	20
({a,d},c)	F	F	T	10
{a,d}	F	F	F	10
Invalid	F	T	F	-
Invalid	F	T	T	-

Table 12: Execution sequences and the value of *x* for the process snippet in Figure 39

Table 12 uses a tuple to represent the order in which activities run and a set to represent activities that run in parallel. Let t_f be the time that *c*'s join condition is evaluated. The means that c may read the data *a* wrote, not that of *b*. This occurs if *p* is false, so *x at* t_f is 10. If *b* does run, however, its write overwrites that of *a* and *x at* t_f is 20.

In the case that multiple BPEL-D data links whose source activities are not disabled write to the same location then the choice of the winning writer is random. This behavior is not expressive enough to allow representing the data dependencies in a standard BPEL process as data links. To illustrate, a random winner property in the example above would allow *a*'s write of 10 to reach *c* even if *b* ran successfully which is not allowed in BPEL.

To specify the behavior needed by reusing data links, we would need to be able to specify different resolution strategies such as 'last writer wins'. While this would solve the problem at hand for an unsplit process, the timing becomes ambiguous upon fragmentation: the order of message sending is difficult to reproduce at the message recipient. This is especially true if the messages are being sent from different processes deployed on different machines in a network. Some alternatives to our approach for handling this problem will be presented in section 5.10.11. Finally, if elements from A_d are in the same partition as the reader, then going from BPEL to BPEL-D and then back to BPEL for the fragments would introduce unnecessary additional assign activities. Therefore, the approach we propose is to instead recreate the control order between the possible writers in the reader's fragment.

Criteria: The criteria we aim to maintain is that conflicting writes between multiple activities are resolved in a manner that respects the *explicit control order,* as opposed to runtime completion times, in the original process model.

Restriction: We assume that data flow follows control flow. In particular, we disallow splitting processes in which a write to and a read from the same location exist in parallel. Another repercussion is that writes in a loop are not visible to the rest of the process until after the loop has completed, and vice versa, because control links must not cross loop boundaries. BPEL does allow this behavior, but it is a violation of the Bernstein Criterion [BAER73].

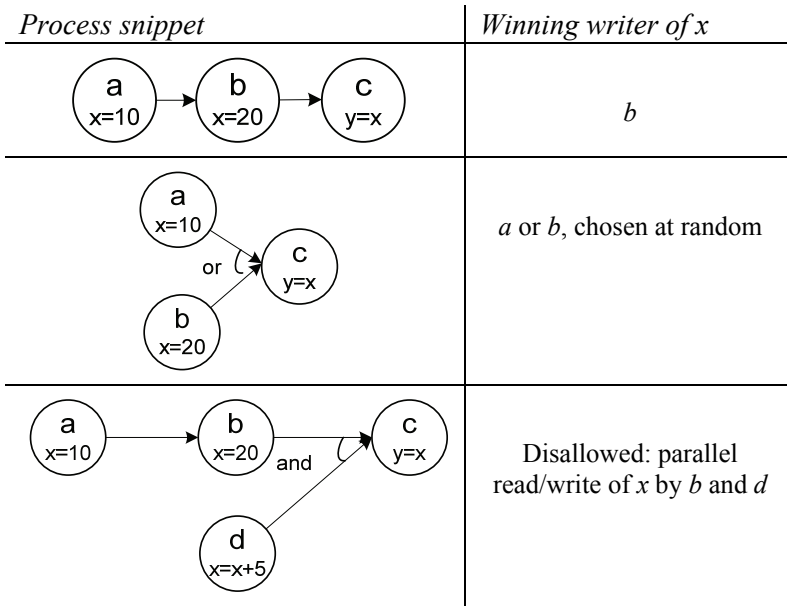


Table 13: Examples of multiple writes to the same location.

Table 13 provides illustrating examples of this write conflict resolution. The Bernstein Criterion states that if two activities are executed sequentially and they do not have any data dependency on each other, they can be reordered to execute in parallel. The restriction for loops is also the assumption made by Gowri Nanda et. al. in [GOCS04] when using compiler theory (in particular Threaded Program Dependency Graphs) to analyze activity dependencies in a BPEL process. However, they do not handle DPE.

5.10.3 Using the Data Dependencies in the Partition

A high level overview of the approach we propose, in [KHKL07B], is: Given a BPEL process and its WSDLs, a partition, and the results of data analysis on that process, we produce the appropriate BPEL constructs in the process of each participant to exchange the necessary data. For every reader of a variable, writer(s) in different participants need to send both the data and whether or not the writer(s) ran successfully. The participant's process that contains the reader activity receives this information and assembles the value of the variable. The recipient uses a graph of receive and assign activities reproducing the control dependencies between the original writers. Thus, any writer conflicts and races in the unsplit process are replicated based on the control flow.

In more detail, the steps of our approach are: (1) Create a 'Writer Dependency Graph' (*WDG*) that encodes the control dependencies between the writers. (2) To reduce the number of messages, use information about a particular partition: Create a 'Partitioned Writer Dependency Graph (PWDG) that encodes the control dependencies between regions of writers whose conflicts can be resolved locally (in one participant). (3) Create 'Local Resolvers' (LR) in the processes of the writers to send the data. (4) Create a 'Receiving Flow' (RF) in the process of the reading activity that receives the data and builds the value of the needed variable.

5.10.4 The Writer Dependency Graph (WDG)

We define a 'Writer Dependency Graph' (WDG) for activity *a* and variable *x* to be the directed acyclic graph representing the control dependencies between the activities in $A_d(a, x)$. We have:

$$WDG_{a,x} = (V_d, E_d)$$

The nodes are $V_d = A_d(a, x)$.

Due to the construction of Q_s, writers in a loop that does not contain *a* are represented in the WDG as one node corresponding to the loop. It is possible to collapse a loop in this manner for two reasons: (1) data follows control and (2) control links must not cross the boundary of a loop. Therefore, writes taking place inside the loop need only be exposed after the loop completes. However, handling data in loops requires special treatment, affecting the construction and placement of sending blocks, and addressed separately in section 5.10.8.

As for the edges, if there is a path in the process between any two activities in V_d that contains no other activity in V_d, then there is an edge in the WDG connecting these two activities. For the purpose of determining data flow paths in the presence of fault handlers, all activities in the body of a scope are treated as having a path to all activities with no incoming links in that scope's fault handlers. Recall that a fault handler is restricted to only read from the faulting activity but may write to any variable. This data that it reads is treated like an

initialization of a variable and will come from the coordinator. Therefore, a reader in a fault handler will never have writers in $A_d(a, x)$ that do not also belong to that fault handler.

Compensation handlers do not affect path construction for WDG: Since activities in them do not write data that is visible to the rest of the process, the WDG of a reader will not contain any activities that are in a compensation handler unless the reader is also in that handler. Handling of data dependencies for activities that are in a compensation handler will be addressed separately in section 5.10.9.

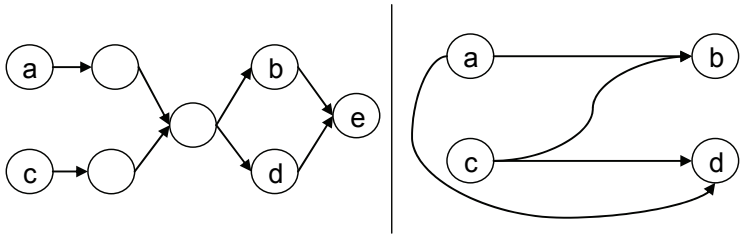


Figure 40*: a, b, c, d* write 'x' and *e* reads 'x' in a process (left) and the WDG (right)

Consider a function *Paths(a,b)* that returns all paths in the process between *a* and *b*. A path is expressed as an ordered set of activities. Formally, and where $\{v_1, v_2\} \in V_d$:

$$(v_1, v_2) \in E_d :\Leftrightarrow |Paths(v_1, v_2)| > 0 \wedge \forall p \in Paths(v_1, v_2) : p \cap V_d = \{v_1, v_2\}$$

A WDG is not dependent on a particular partition. Consider Figure 40. Assuming that the writers of *x* that reach *e* are $A_d(e, x) = \{a, b, c, d\}$, then the WDG of *e* for *x* is:

$$WDG(e, x) = (\{a, b, c, d\}, \{(a, b), (a, d), (c, b), (c, d)\})$$

5.10.5 Partitioned Writer Dependency Graph (PWDG)

The number of messages exchanged between partitions to handle the split data can be reduced by: (i) using assigns for writers in the partition of the reader; (ii) joining results of multiple writers in the same partition when possible. This section shows how to do so while maintaining the partial control order of writers amongst partitions.

The 'Partitioned Writer Dependency Graph', for a given WDG, is the graph representing the control dependencies between the sets of writers of *x* for *a* based on a given partition. A PWDG node is a tuple, containing a participant name and a set of activities. Each node represents a 'region'. A region consists of activities in the same participant, where no activity in another participant is contained on any path between two of the region's activities. These regions are constructed as follows:

1. Place a temporary (root) node for each partition, and draw a temporary edge from it to every WDG activity having no incoming link whose source activity is in that partition. These added nodes and edges are needed to build the subgraphs in step 2.
2. Form the largest weakly connected subgraphs (see below) where no path between its activities contains any activities from another partition.
3. The regions are formed by the subgraphs after removing the temporary nodes and edges added in step 1.

Each edge in the PWDG represents a control dependency between the regions. Note that the path restriction in the second bullet is there to avoid circular dependencies between the regions. Recall that a directed graph is *weakly connected* if replacing all directed edges with undirected edges results in a connected graph. The edges of the PWDG are created by adding

an edge between the nodes representing two regions, r_1 and r_2, if there exists at least one link in the WDG whose source is in r_1 and whose target is in r_2.

For each node in A_d that corresponds to a split loop, one participant is chosen by the algorithm as the 'owning participant' for the purposes of PWDG construction. The mechanism for choosing the owning participant and the details of its data collection will be detailed in section 5.10.8.

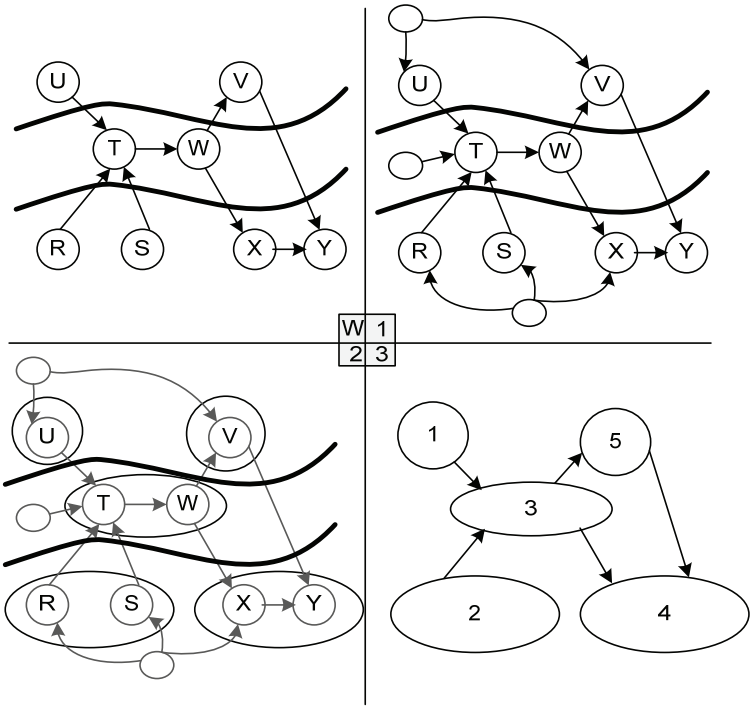


Figure 41: Three steps to creating a PWDG (box 3) from a WDG (box W)

Figure 41 shows the creation of a PWDG from the WDG with a particular partition shown in box W. In this partition, writers *R, S, X*, and *Y* are in one fragment, *T* and *W* in another, and *U* and *V* in a third. Box 1 shows step 1: the addition of roots nodes, shown unnamed in the figure. Box 2 shows the selection of the connected subgraphs for each fragment, and the PWDG is shown in box 3 with each subgraph reduced to a single node and the edges between the regions pushed to the boundary of the PWDG's nodes. Note that the resulting PWDG has two nodes for each of the top and bottom fragments, and one node for the middle fragment.

If all writers and the reader are in the same partition, no PWDG is needed or created. Every PWDG node results in the creation of constructs to send the data in the writer's partition and some to receive it in the reader's partition. The former will be the Local Resolvers. The latter will be part of the Receiving Flow for the entire PWDG. Special cases will be made for loops and compensation handlers.

5.10.6 Sending the Values and the Use of Local Resolvers (LR)

A writer sending data to a reader in another participant needs to send: (1) whether or not the writer was successful and if so, (2) the value of the data. First, we consider how the writers send data to the RF. The idea is to take advantage of the fact that write conflicts between writers in one PWDG node can be resolved locally at the fragment in which the writers in that node are placed. We name the pattern of activities constructed to resolve these conflicts and send the data a 'Local Resolver' (LR).

Intuitively, the LR waits for all writers. It is linked to a sending block that sends the resulting value of the variable when the writers have completed or were disabled. Therefore, each node in the PWDG results either in: (1) no change, (2) a sending block at the writers' fragment and a receiving block at the reader's fragment, or (3) an assign activity if the reader is in the same participant as the PWDG node. The joinCondition at a Local Resolver's invoke in case 3 or assign in case 2 is a disjunction of the incoming links ('or' join), because if none of the writers ran successfully then the data is invalid.

If there is only one writer in a PWDG node: if the node is in the same partition as the reader, do nothing (case 1); otherwise, the LR is an 'explicit data sending block' (case 2).

If there is more than one writer, the algorithm in the function CREATE-LOCAL-RESOLVER-MULTIPLE-WRITERS below is used (cases 2, 3). Basically, conflicts between writers in the same PWDG node, *n=(p,B)*, are resolved in the process of *p*: An activity waits for all writers in *n* and collects the status for each set of queries.

Assume a PWDG for variable *x* and a reader *a* in partition p_r. For each PWDG node, *n=(p,B)*, *with more than one writer,* several constructs are added to the process of participant *p* as defined by CREATE-LOCAL-RESOLVER-MULTIPLE-WRITERS below. Note that in the algorithms, '+' used on Strings means String concatenation as is commonly done in programming languages.

```
function CREATE-LOCAL-RESOLVER-MULTIPLE-WRITERS(Node n,
                                                Activity a,
                                                Variable x)
1  Q = Q_sp(n, a, x)
2  if p = p_r
3    Add b=new empty, v=new variable, v.name=idn(n)
4    t=idn(n)
5  if |Q| = 1, let Q = {q_s}
6    if p ≠ p_r
7      b=CREATE-SENDING-BLOCK(name(x))
8    ∀w ∈ π_2(q_s)
9      if type(w) = loop
10       //see section 5.10.8
11     else add link l = (w, b, true())
12 else  // more than one query set
13   if p ≠ p_r
14     Add b=new invoke, v=new variable
15     b.inputVariable=v, b.toPart=("data",x), b.joinCondition=true()
16     t=name(v)
17   ∀q_s ∈ Q
18     s=CREATE-ASSIGN-SCOPE(t,qs)
19   Add link l_1 = (s, b, true())
```

Before explaining its details, we define several helper functions. We define $Q_{sp}(n, a, x)$ to be a function that takes a PWDG node, a reader *a*, and a variable *x*, and returns a set of tuples where each tuple has a set of queries and a set of possible writers. These query sets are the same as those in $Q_s(a, x)$, but the writer sets are subsets of those in $Q_s(a, x)$ such that they only contain writers in the activity set of the PWDG node, *n*. A tuple resulting in an empty writer set is not included in $Q_{sp}(n, a, x)$:

$$Q_{sp}(n,a,x) := \{(q,w) | \exists (q,w_a) \in Q_s(a,x) : w = \pi_2(n) \cap w_a, w \neq \emptyset\}$$

Also, consider *id* to be a map associating a unique string given a query set, and *idn* to do the same for each PWDG node.

If the reader is in the same partition as the writers in this node, then we put an empty activity (line 3). If all writers write to the same set of queries, and the node is not in the reader's participant, use an explicit data link sending block (see CREATE-SENDING-BLOCK below). Create a link from every writer to *b*, which is either the empty activity or the sending block's invoke activity (line 6-11). An example is in Figure 42, where *R* and *S* write to the same query set and so a sending block is used that is linked from both.

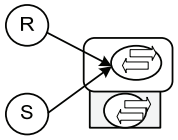


Figure 42: Sending data from the node of the PWDG in Figure 41 containing R and S

If there is more than one query set, the status for each one needs to be written. If the reader is in another participant we create an invoke activity that runs regardless of the status of the writers (line 13-18). For each query, use a structure similar to a sending block (i.e.: scope, fault handler) to get the writers' status (line 16, CREATE-ASSIGN-SCOPE), but using assign activities rather than invoke activities. The assign activities write true or false to a part of the status variable corresponding to the query. Create links from each writer of the query set to the 'assign' in the scope. Create a link (line 19) from the scope to either the 'empty' from line 3 or the 'invoke' from line 14.

```
function CREATE-ASSIGN-SCOPE(String t, Set qs)
 Add s= new scope
 s.addFaultHandler('joinFailure', asf=new assign)
 s.setActivity(as=new assign),
 as.suppressJoinFailure='no'
 asf.addCopy(false, t + ".status" + id(qs))
 as.addCopy(true, t + ".status" + id(qs))
 ∀w ∈ π2(qs)
   if type(w) = loop
     //see section 5.10.8
   else add link l = (w, as, true())
 return s
```

```
function CREATE-SENDING-BLOCK(String x)
 Add s= new scope
 s.addFaultHandler("joinFailure", invf=new invoke)
 Add v = new variable
 invf.inputVariable=v
 invf.toPart=("status",false)
 s.setActivity(inv=new invoke)
 inv.inputVariable=v
 inv.toPart=("status",true)
 inv.toPart=("data",x)
 inv.suppressJoinFailure="no"
 return inv
```

Note that we assume variable initialization (using an assign or in the variable declaration) in the fragments to avoid XPath selection failures where a writer writes using a query to a

location deeper than an uninitialized part of a variable. An example is writing into *a/b/c* in a fragment if *a/b* is not written in that fragment.

5.10.7 Collecting the Data Using a Receiving Flow (RF)

Having explained how data is sent, we now address how it is received. A Receiving Flow, for a reader *a* and variable *x,* is the structure created from a PWDG that creates the value of *x* needed by the time *a* runs. It contains a set of receive/assign activities, in *a*'s process, to resolve the write conflicts for *x.*

Consider p_r to be the reader's partition, and *G* to be the PWDG from $WDG(a, x)$. An RF defines a variable, $v_{tmp,}$ whose name is unique to the RF. A Receiving Flow is created from *G* by CREATE-RF below, where *ba* and *ea* denote the first and last activities of a block, respectively and a subscript is used on them to identify which node's block they are for (i.e.: ea_{n1} is the *ea* set created in PROCESS-NODE(n1)):

```
CREATE-RF(PWDG G)
1  F=new flow
2  for each n = (p, B) ∈ π1(G)
3    PROCESS-NODE(n)
4  for each e = (n1, n2) ∈ π2(G)
5    for each d ∈ ea_n1
6      Add a link l = (d, ba_n2, true())
7  Add a_f=new assign
8  a_f.addCopy(v_tmp, x)
9  Add links l_f = (F, a_f, true()) and l_r = (a_f, a, true())
10 joinCondition(a) = joinCondition(a) ∧ (l_r ∨ ¬l_r) //recall: a is the reader
```

Create a flow activity (line 1). For each node (line 2-3), we will add a block of constructs to receive the value of the variable and copy it into appropriate locations in a new, uniquely named variable v_{tmp}. Link the blocks together (line 4-6) by connecting them based on the connections between the partitions, Link the flow to an assign (line 7-9) that copies from v_{tmp} to *x*. Link the assign to *a* and modify *a*'s join condition to ignore the new link's status (line 10). Next, consider the PROCESS-NODE function:

```
PROCESS-NODE(Node n) //recall n=(p,B)
1  Q = Q_sp(n, a, x), ea = ∅
2  //All activities added in this procedure are added to F
3  if p = p_r
4    if |Q| = 1, let Q = {q_s}
5      ba=new assign
6      for each q ∈ q_s, ba.addCopy(name(v_tmp) + q, x + q)
7      if |B| = 1, let B = {b}
8        Add link l_0 = (b, ba, true())
9        ea = ea ∪ {ba}
10   else //there is more than one query set
11     ba=new empty
12     for each q_s ∈ Q
13       CREATE-Q-ASSIGN(q_s, "x", idn(n)+".status"+q_s)
14   if |B| ≠ 1
15     Add link l_0 = (em, ba, true()), where em=empty from LR
16   joinCondition(ba)=status(l_0)
17 else //p ≠ p_r
18   Add rrb=new receive,joinCondition(rrb)=true(),rrb.variable=r_i
19   ba=rrb   //note that ea will be created in lines 21,24
```

```
20   if |Q| = 1, let Q = {q_s}
21       CREATE-Q-ASSIGN(q_s, "r_i.data", "r_i.status")
22   else
23     for each q_s ∈ Q
24       CREATE-Q-ASSIGN(q_s, "r_i.data", r_i+".status"+q_s)
```

For each node: (1) If the node is in the same participant as *a* and has, one query set, add an 'assign' copying from the locations in *x* to the same locations in v_{tmp} (line 3-6). If the node has only one writer, link from the writer to the 'assign' (line 8). If it has more than one writer, an 'empty' was created in the Local Resolver (LR), so link from *that* 'empty' to the 'assign' (line 14-15). If the node has more than one query set, create an 'empty' instead of an 'assign' and (line 10-12) create one 'assign' per query set. Create links from the 'empty' to the new assign activities whose status is whether the query set was successfully written (line 13, line c3). Add a 'copy' to each of these assign activities, for every query in the query set, from the locations in *x* to the same locations in v_{tmp} (line c4). Then, (line 16) set the join condition of the empty/assign to only run if the data was valid. (2) If the node is in another participant, create a 'receiving block' (line 18) instead of an 'assign'. Set to true the joinCondition of the 'receive' so it is never skipped. Again copy the queries into assign activities (line 20-24). Note that the creation of the assigns for each query is as follows:

```
CREATE-Q-ASSIGN(Set qs, String var, String statusP)
c1 Add act=new assign
c2 ea = ea ∪ {act}
c3 Add link l = (ba, act, statusP)
c4 for each q ∈ q_s, act.addCopy(name(v_tmp) + q, var + q)
```

Figure 43 shows the sending and receiving blocks for the example whose PWDG is in Figure 41, assuming that the reader is in a fourth fragment shown in the second row of Figure 43.

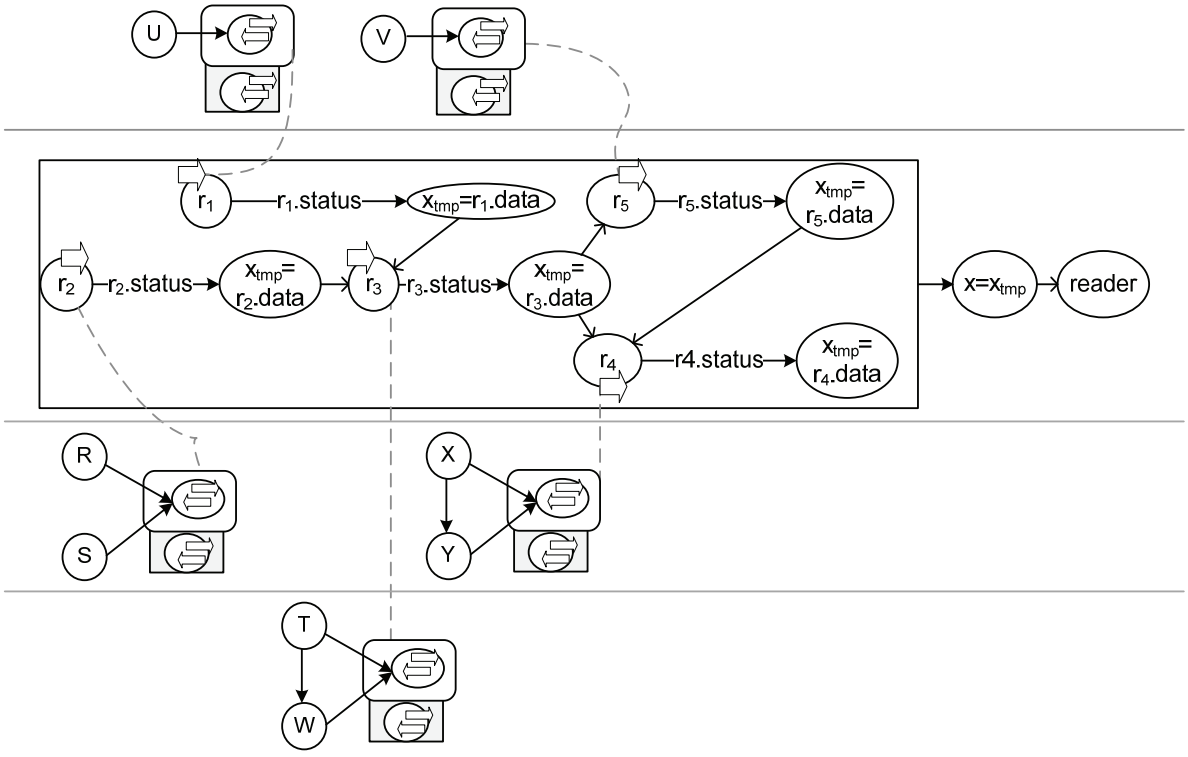


Figure 43: Sending blocks and receiving flow at the receiver's fragment of example in Figure 41

Notice in Figure 43 how each PWDG node whose writer is not in the same fragment as the reader results in adding to the receiving flow a receive activity linked to an assign. The transition condition of this link is the value of the sent status. These receive/assign pairs are joined to each other to match the control dependencies in the PWDG. The figure relates the activities to the corresponding PWDG nodes in Box 3 of Figure 41 by matching indices, i.e. r_3 corresponds to node 1 in Figure 41 (*T* and *W*'s fragment). Default conditions are used unless otherwise noted.

Note that receiving flows reproduce the building of the actual variable using BPEL semantics. Thus, the behavior of the original process is mirrored, not changed. The special handling needed for the case of split loops and split scopes with compensation handlers is described next.

5.10.8 Loops: Receiving and Sending Blocks

The key to supporting loops in this approach is to treat the activities in a loop as a process. This provides data needed between activities for each iteration, using the same steps as have been specified for data between activities in a process. As with BPEL-D, three more cases need to be considered: data needed before iteration begins, data needed from previous iterations, and data needed by subsequent activities that occur after the loop. An example of propagating data in split loops is shown in section 5.10.10.

The data needed before loop iteration begins is assembled as for any other activity, treating the whole loop as a reader. Therefore, one runs the algorithms above starting with $Q_s^{preloop}(l, v)$ for every variable *v* read in the loop and written outside the loop. The difference is in the placement of the receiving block: one such block is placed in the process fragment of each loop that reads the needed variable. If more than one fragment of the loop needs the variable, then a receiving block is placed in each such fragment and the sending blocks have extra invokes added to send to the fragments. If the receiving block is a complex receiving flow, one may reduce the number of messages and created activities by making one process fragment contain the receiving flow and forward the value of the variable to the other fragments.

Data from other fragments in the same iteration is treated like data from other parts of the process in the non-loop case, and found using the normal $Q_s(a, v)$ where *a* is a primitive activity in the loop. However, if the reader also needs the value of the variable from a previous iteration, then additional constructs are needed as explained next.

Data needed from one loop iteration to the next, and/or from the values collected before the loop is now considered. One needs to ensure that the data that reaches a reader properly combines: data from before the loop started, data from previous iterations, and data from other fragments in this current iteration. Data needed in the next iteration needs to be gathered at the end of the current iteration in the fragment(s) that need it. It is handled by starting the algorithms above starting with $Q_s^{intraloop}(l, v)$. The difference is that the receiving block is an end of split activity receiving block. As for data needed from before the loop, the receiving block may need to be placed in more than one fragment and if so extra invokes are added in the sending blocks in the same manner.

This provides the loop fragments with the proper variable value when they iterate again. One must still combine this with data needed from the current iteration. If there is at least one

reader activity in the fragment of the loop that also needs the value of the variable from its current as well as its previous iterations (i.e.: it also needs a receiving block), then an assign activity is added in that fragment of the loop. For each such reader: (1) the assign activity copies from the variable to the target variable of the reader's receiving block and (2) a link is created from the assign to the first activity (receive or flow) in the receiving block.

Next we consider data needed after a loop ends. An example is shown in Figure 51 for the variable *acctPoints*. This will be the case when a loop itself appears as a node in a PWDG. If the loop is split, then the definition of PWDGs in section 5.10.5 stated that a corresponding PWDG node is assigned to one participant by the splitting algorithms. The selection of this participant occurs as follows: For each loop node l_d, in a PWDG, that corresponds to a split loop, and for each participant *p* containing a fragment of that loop, assign a cost function $suitability\_loop(l_d, p, A_d(a, x))$ that assigns a value to making *p* the owning participant of the loop. This value is a measure of the suitability of doing so, based on the cost of collecting the data in *p*'s fragment. The suitability increases as the number of necessary inter-fragment messages decreases and is calculated as follows: add 1 for every writer from $A_d(a, x)$ that is nested in l_d and contained in *p*. Add 1 if *p* contains the reader, *a*. For each l_d, the participant with the highest suitability is the participant to which the loop node will be assigned in the PWDG. In case of a tie, one is chosen at random between the tied participants. Data written in the loop and needed after the loop is restricted to be only for whole variables. In other words, multiple queries in the presence of write conflicts in RFs containing writers inside a loop and writers *outside* the loop are not supported.

We now explain how data needed after the loop is collected and sent by this fragment. The data is valid if it was written by a writer in at least one iteration of the loop. Therefore, a status variable is created in the owning fragment and initialized to false before the loop starts: using an assign activity outside the loop that has a link to the loop. The status of this new link is ignored by the join condition of the loop.

If all writers are in the owning fragment, such as in Figure 51, then an assign activity is created for each writer. A link is created from each writer to the respective assign. The assign contains one copy statement that copies 'true' to the status of the data.

If at least one writer is in another fragment of the loop, the data is collected at the owning fragment by creating an end of split activity receiving block inside the loop. This receiving block and its corresponding sending blocks are constructed starting from $Q_s^{postloop}(l, v)$ and treating as being in the owning fragment. The receiving block is modified such that the value of the owning fragment's status is set to true if at least one writer was successful: An additional copy statement is added to each assign in the receiving block that copies from 'true' to the status of the data. Thus, the value and status are made available at that fragment once the loop completes.

The sending block is created in the owning fragment outside the loop. The change for loops is the writer is now the loop itself, and the sending block link from the writer has a transition condition: the status of the data. There is no special treatment needed for the corresponding receiving block.

5.10.9 Compensation Handlers: Receiving and Sending Blocks

Compensation handlers read data from a snapshot of the process state taken when their associated scope instance completed and do not write data visible outside of the

compensation handler itself. Therefore, one can provide the data needed from the body of the associated scope before the scope ends and treat the activities in a compensation handler as a process. Data needed between activities of a split compensation handler is treated in the same way as data between activities of a process.

Data needed for the compensation handler, *h*, is assembled by repeating the steps for a reader activity, and treating the handler itself as the reader. Therefore, one runs the algorithms above starting with $Q_s^{handler}(h, v)$ for every variable *v* read in the handler and written in the body of the scope. The difference is the placement of the receiving block: An end of split activity receiving block is used where the split activity is the corresponding scope. In line with the treatment for loops, if more than one fragment of the handler reads the variable, then a receiving block is placed in each such fragment and the sending blocks have extra invokes added to send to the fragments. If the receiving block is a complex receiving flow, the number of messages and created activities may be reduced by placing the receiving flow in one fragment of the handler's scope and forwarding the variable value to the other fragments.

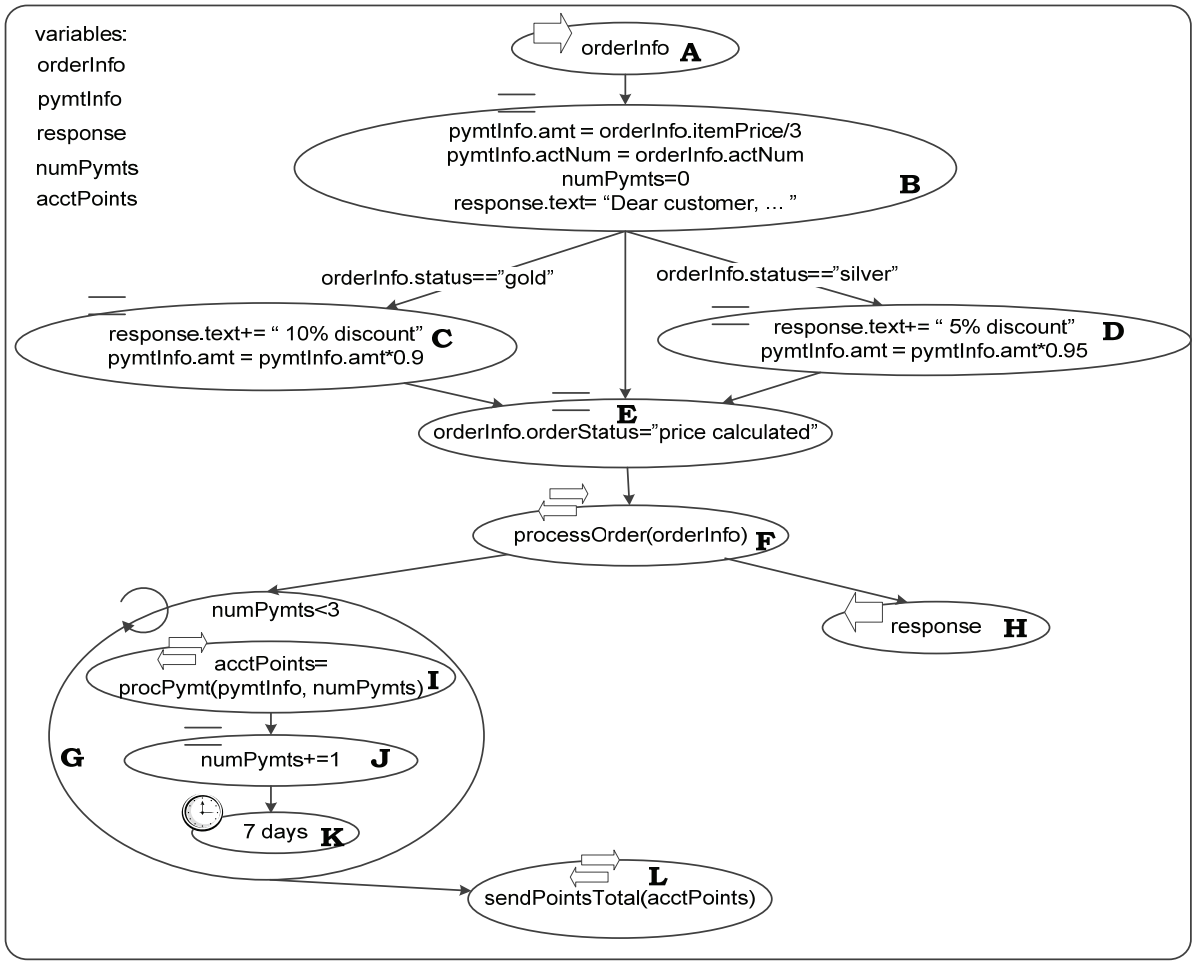


Figure 44: Purchasing scenario, detailed view of process in Figure 26

5.10.10 Scenario Illustrating Split BPEL Data Dependencies

The splitting of implicit data dependencies is shown using the scenario in Figure 44. In fact, this is the same process that was shown at a high-level in Figure 26, except that in this section the details of each activity are provided.

The process provides a 10% discount to members with 'Gold' status, a 5% discount to those with 'Silver' status, and no discount to all others. After receiving the order (*A*) and calculating the appropriate discount (*C*, *D*, or neither), the order status is updated (*E*), the order is processed (F), the customer account is billed in 3 weekly installments (*G*, with *I, J, K*), and a response is sent back stating the discount received (*H*). Additionally, a message with the rewards point total is sent (*L*). The data flow analysis for this scenario is detailed in [KOKL07].

Consider that this process is split as shown in Figure 26. As a reminder, that was:

$$L_c(G) = y, P1 = \{p_w = (w, \{I, H\}), p_x = (x, \{B, L\}), p_y = (y, \{E, C, J, K\}), p_z = (z, \{A, D, F\})\}$$

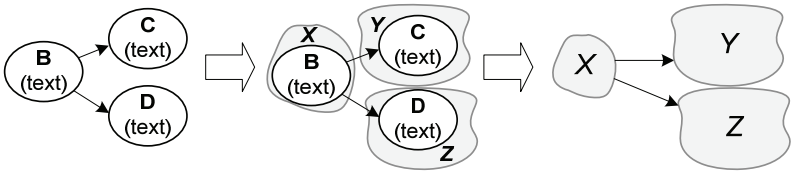


Figure 45: Creating the PWDG of *H*

The salient activities for showing the algorithms' details are those with more than one writer in their PWDG. The PWDG of *H*, which reads *response*, is in Figure 45. The resulting sending and receiving blocks for getting data to *H* are shown in Figure 46. In the figures in this section, we do not put icons showing activity types of added activities nor do we show the split of control, so as to focus the reader's attention and reduce clutter.

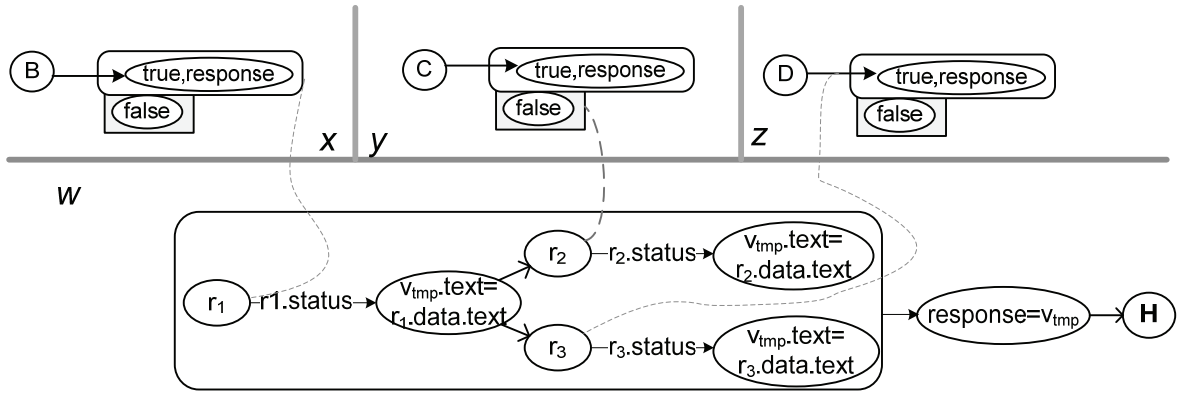


Figure 46: Sending/receiving the value of *response* for *H*

Next, we show, in Figure 47, a receiving flow containing an assign activity instead of a receive activity because a writer (*A*) is in the same partition as the reader (*F)*.

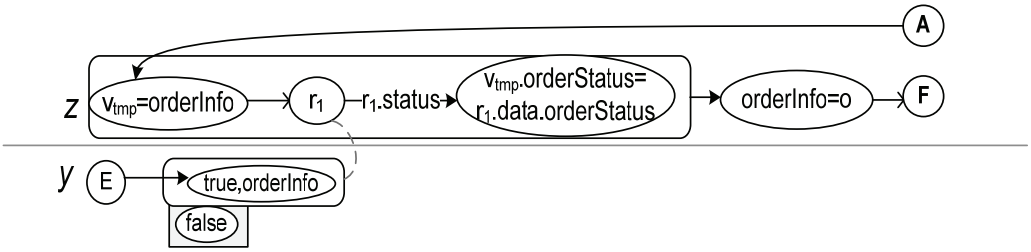


Figure 47: Sending/receiving the value of o*rderInfo* for *F*

Next, consider data in the split loop, *G*. It needs *numPymt*s and *pymtInfo* before it starts. Figure 48 shows the transfer of *numPymts*.

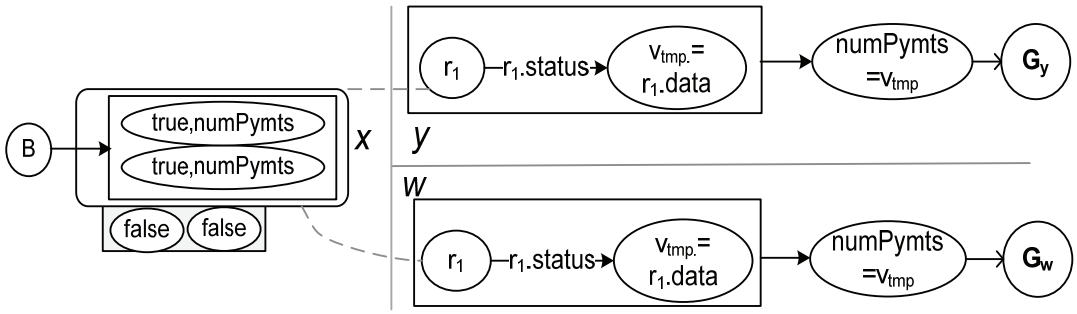


Figure 48: Sending/receiving *numPymts* for *G*'s fragments before they start

The PWDG of G for *pymtInfo* is shown in Figure 49. Formally, it is $PWDG_{G,pymtInfo} = (\{n_1 = (x, \{B\}), n_2 = (y, \{C\}), n_3 = (z, \{D\})\}, \{(n_1, n_2), (n_1, n_3)\})$.

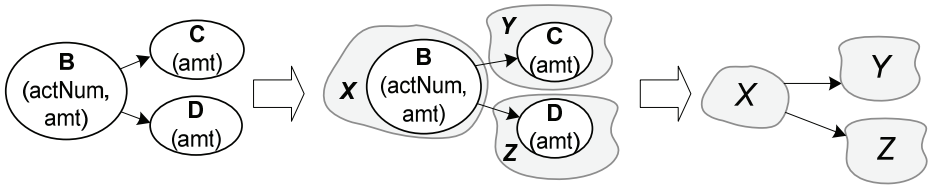


Figure 49: Creating the PWDG of *G* for *pymtInfo*

Thus, the structures in Figure 50 are created to send the value of *pymtInfo*.

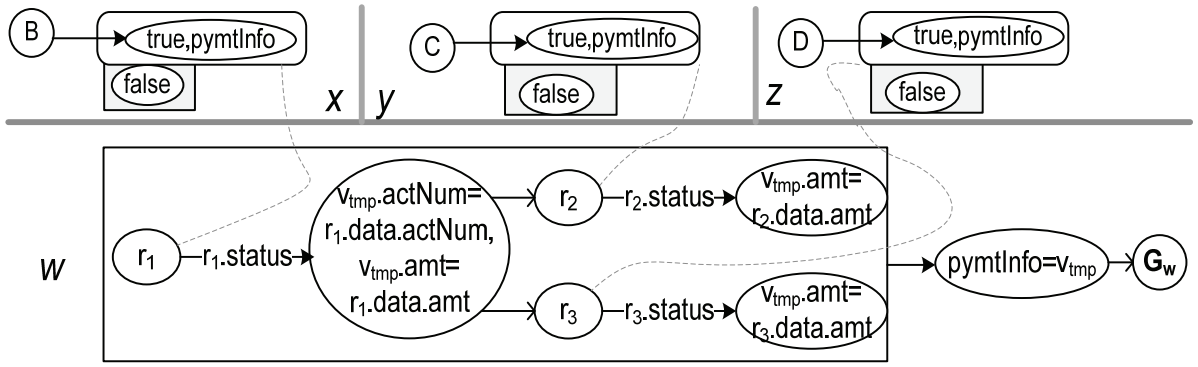


Figure 50: Sending *pymtInfo* to *G*'s fragment in p_y before it starts

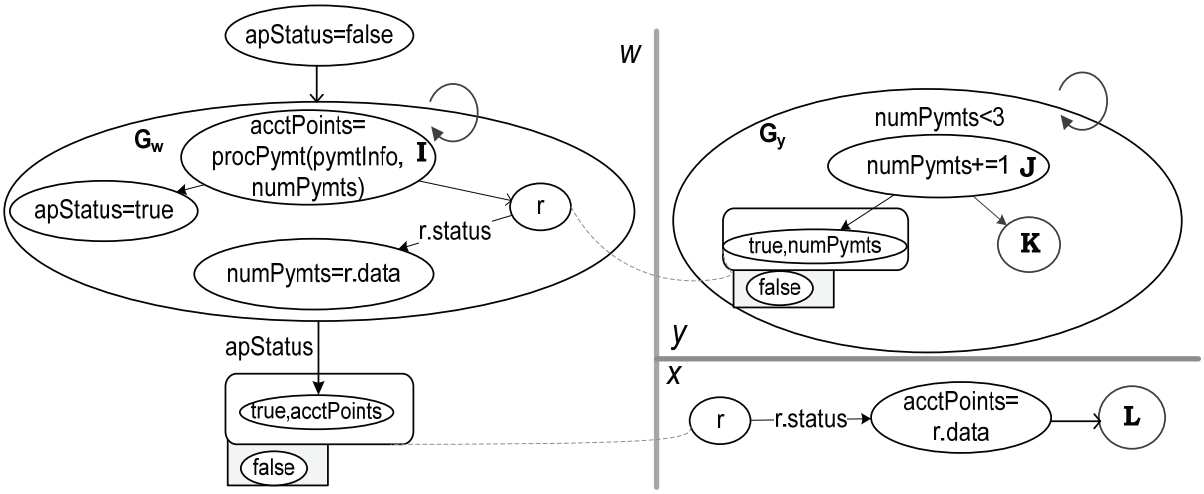


Figure 51: *numPymts* to I from a previous iteration, and *acctPoints* to *L* after the loop

Figure 51 shows the inside of loop *G*. *G* sends *acctPoints* after it ends, and needs *numPymts* between iterations: *I* needs the value of *numPymts* that has been updated by *J* in the previous iteration. Additionally, *L* needs the value of *acctPoints* from *G*.

5.10.11 Discussion of Alternatives

Several options were considered in determining how to split data dependencies in split BPEL. With our goal of minimizing the need for new middleware when possible, several options such as the use of shared databases and/or synchronized/partially ordered logical clocks [FIDG91] were stricken from the start. One other option was to resolve the choice of which write wins by numbering the activities such that control order is reflected in the numbering. The receive that receives the data from a particular writer can be then assigned the number of that writer, allowing the number's use in making the choice of which assignment to make at runtime. However, BPEL does not support reordering activities based on message contents. The receiving flow concept was chosen because it does not create extra reconciliation activities, it uses BPEL to reproduce BPEL behavior, and it makes local optimizations easier to find and integrate into the main model. The approach does make the assumption that the BPEL engine does not discard messages for a process having a receive activity that may still activate, which is typical.

5.11 Conclusion

In this chapter, we have provided the definition of a partition, the representation and derivation of information about the unsplit process model to enable the coordination of split loops and scopes, as well as algorithms for splitting BPEL-D and BPEL processes. In the introduction, we stated our driving design decisions are interoperability, transparency, and to avoid new (proprietary) middleware. The result of the fragmentation is a set of standard BPEL process models including proper communication of data and control between participants, satisfying interoperability. The fragments can run on any BPEL engine provided that there are no split loops and scopes that require coordination, satisfying the driver for not requiring new middleware. Furthermore, the splits are transparent, i.e. it is clear where the changes are and they are done in the same modeling abstractions as the main process model. Having placed the activities that handle data and control dependencies at the boundaries of the process, one can use naming conventions on the newly added activities to enable graphical/text-based filters to toggle views between the activities of the unsplit process and the 'glue' activities we have added. This would help comprehension of the fragments by the process modeler.

Transparency has been achieved by using patterns of sending and receiving activities. In particular, dead-path elimination is propagated through the creative use of fault handling scopes. We showed that splitting standard BPEL reuses the basic concepts of splitting BPEL-D, but has additional complexity due to the use of shared variables, especially due to the control semantics of BPEL such as dead-path elimination and parallelism. An example is the ability to 'revive' a dead path with an 'or' join condition, making the data flow non-trivial. We also showed that respecting the Bernstein Criterion is the requirement that enables one to split standard BPEL process models. If the process did not respect this criterion, one would have to take into consideration actual completion times of activities in order to split data dependencies, which goes beyond BPEL's capabilities.

In the next chapter, we will show how split loops and scopes can be executed using new coordination protocols.

Chapter 6 Protocols for Inter-Fragment Coordination

In Chapter 5, we showed how a single BPEL(-D) process can be split into several BPEL processes, one for each participant. The corresponding algorithms provided the information needed to wire the resulting processes together at deployment time and to ensure correct instance-level propagation of data and explicit control dependencies at runtime. In this chapter, we provide the ability to split the implicit control flow implied by loops and scopes.

One option for fragmenting these constructs could be the use patterns of BPEL activities similar to the ones described in Chapter 5. However, this option would result in a significant increase in complexity of the resulting process models. The complexity is mainly due to the fact that one needs to synchronize the start and end of all fragments that result from splitting loops and scopes. For example, if one fragment of a loop completes then it has to wait for the loop's other fragments to complete and notify them that it itself has completed. Additionally, splitting compensation and fault handling behavior in this way is very difficult: One must ensure that the proper fault handler runs, fragments of a fault handling scope wait appropriately, termination and handler start and end are synchronized, and faults propagate appropriately up the scope hierarchy. Moreover, no single scope fragment may have enough information to determine and enact default compensation order. Therefore, we find that splitting loops and scopes is one clear point where it is worth extending BPEL to easily support fragmentation. But even these extensions are minimal: three new attributes applicable to scopes and loops.

Furthermore, we propose solving the problem of executing split loops and scopes by adding a coordinator that coordinates the fragments so they can behave as logical units. An example is shown in Figure 52, where a loop is split between three participants and the resulting fragments are run with the use of a coordinator. As a result, process fragments can run at different participants' sites such that the resulting behavior is the same as that of the original business process running at a central location.

In this chapter, we motivate the use of WS-Coordination for this problem, describe how it is used to solve it, and present the coordination protocols required to execute split loops and fault/compensation-handling scopes.

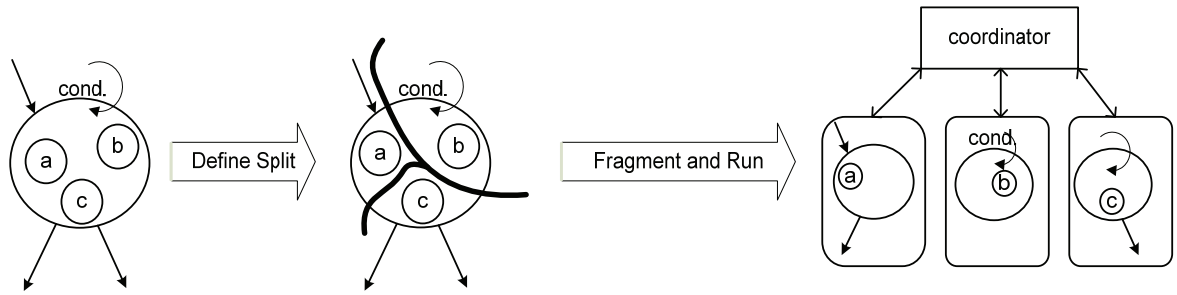


Figure 52: Fragmenting a loop, using a coordinator at runtime

6.1 Motivation

There are cases where constructs in a BPEL process need to reach agreement before continuing (i.e.: a loop or a scope). A loop cannot iterate again until all the work of its current

iteration has completed; a scope cannot make its fault handlers unavailable until all the work of the scope has completed; links leaving the boundary of a loop or a scope cannot fire unless the loop or scope has completed. Therefore, there is implicit coordination already occurring at the boundaries of such constructs even in a process that is not fragmented.

One goal of this thesis is to reproduce the same behavior whether fragments of such constructs are running at different partners or whether the entire construct is being run locally at one partner. In this chapter we demonstrate that this is achieved by making the implicitly coordinated behavior already present in these constructs *explicit,* thereby using coordination among fragments.

The constructs for which we use coordination are scopes and loops. The commonality is that these constructs have an explicit beginning and an explicit end. Beyond that, explicit coordination of these constructs will need additional explicit coordination messages dependent of the type of the construct.

6.2 Using WS-Coordination

The choice of WS-Coordination over other coordination frameworks lowers the barrier of entry for adoption. This is due to its status as a member of the WS-* stack. The coordination logic for fragmented loops and scopes is thus modeled as protocols that plug into the specification [OASI07C]. Recall from the description in Chapter 2 that WS-Coordination describes a pluggable framework for coordinating the agreement of the outcome of the execution of a collection of services that jointly perform a ‘distributed action’. It specifies an Activation Service and a Registration Service.

When splitting processes each fragment of a split activity (loop or scope) corresponds to a service of a distributed action. The fragments of a split activity are known at design time and their location is known at deployment time. Additionally, the fragments are peers: a fragment of a loop in one process can be reached before its other fragments (in other processes), and it is not known ahead of time which one will start the corresponding protocol instance. In the case of a split loop, for example, the associated agreement protocol is about agreeing whether another instance of the split loop body must be run.

The case in this thesis differs from traditional coordination protocols such as WS-BusinessActivity (WS-BA) and WS-AtomicTransaction(WS-AT). Most importantly, the optional Activation Service is not used for two reasons: (1) A protocol instance is automatically created when the first fragment of a split activity is reached. (2) The context does not need to be dynamically flowed between the participants, because the participants are known in advance and each participant has enough information for the coordinator to determine exactly which protocol instance it belongs to. We define two new coordination protocols, one for split loops and the other for split scopes.

6.2.1 Coordination for Fragments Versus Child Elements

Current work on coordination protocols for BPEL is focused on the coordination between a parent scope and its child scopes. In contrast, our work focuses on coordination between scope fragments of a single scope.

While the behavior may make the problem seem similar on the outset, it in fact manifests itself very differently because fragments are not nested. Consider the question: Is the relationship between fragments of a scope and a coordinator coordinating them the same as

that between child scopes and their parent scope? If the answer is yes, then we could reuse the WS-BA protocol for parts of our approach. However, the answer is no.

There are several reasons for this. In BPEL, the entire scope has a 'place' in the navigation. If it is fragmented, then navigating into it can only occur once all its fragments have been reached in the control flow. Similarly, navigating out of it can only occur once all have completed. Child scopes, on the other hand, can begin when they *alone* are reached in the control path and can complete once their individual work completes without having to wait for their sibling scopes. This also affects whether the compensation handler can be installed. Another difference is the behavior of links leaving and entering a fragmented scope. A third is fault scoping: In a nested scope, one looks immediately up the scope hierarchy for a matching fault handler. In a split scope, the other fragments must be searched for handlers before searching up the hierarchy. Finally, a fault in a child scope may be caught and handled in that scope and thus not affect the scope's siblings in any way. On the other hand, a fault in one fragment will cause the other fragments to end abnormally even though they completed all their activities successfully. The behavior for coordinating fragments and child elements, while different, is close enough that the prior art motivates the choice of also using coordination protocols for fragments.

In this thesis, the parent-child scope relationship is handled locally at each fragment. It does not need to be externally coordinated (i.e. by a coordinator) because every BPEL engine must support this kind of coordination to support the BPEL language. To the best of our knowledge, prior uses of BPEL and coordination have a single scope or activity spawning or controlling the participants in the coordination. This results in properly nested transactions with single points of control. As described in [WEVO02], nested transactions occur as a result of spawning from a single node. It is clear how and when the context must be sent to the transaction participants and the resulting transaction tree is built dynamically at runtime. [WEVO02] explains nested transactions mainly in the context of multi-threaded processing: The parallel threads are siblings and the spawning thread is their parent. Their nesting is tree-structured and dynamic (based on thread spawning).

The relationships between the fragments in this thesis, however, are predetermined and peer-to-peer. Any of the fragments might be ready to start before any of the others. It is not known in advance that only one of them will attempt to start the protocol (and if so, which one), because we do not have a single user spawning the coordination. However, we have a-priori knowledge about where the fragments are, the scope nesting hierarchy, and information about existing fault handlers (fault names and handler locations). This is the information encoded in the relationship tree defined in Chapter 5.

Unlike typical WS-Coordination usage, protocols for fragmented loops and scopes are driven by process lifecycle events and not by application messages. Another difference is that there is no need for participants in this thesis to flow to each other a coordination context containing the coordinator endpoint and protocol identifier: The fragmented processes already know which business process definition they belong to, which activity they are a fragment of (the activity name), and which instance of that process they belong to (the value of the common correlation set). Therefore, the participants can provide enough information in the Registration step of WS-Coordination for both coordinator and participant to identify which instance of which protocol is being used. Additionally, the participants and coordinator are known a-priori so the context does not need to be dynamically flowed between the

participants. The protocol instance is created when the first fragment of a split activity registers. For these reasons, the (optional) Activation Service is not needed.

6.3 Coordination Approach Requirements

The main ideas in using coordination here are to enable synchronization of work across the partners to keep them in consistent states, as well as to pass enough information so that *local behavior* can then take over to reach the common global behavior.

The following is implied by our coordination approach to take into account the nature of the behavior needed for splitting scopes and loops:

- Extending BPEL to denote a scope or loop as a fragment, using the new attributes introduced in section 5.6: 'belongs-to', 'is-responsible', and 'fragmented'.
  - This signals to the BPEL engine running the fragments that an activity must be coordinated and cannot be run as any other loop or scope.
- Defining the protocols and interfaces for protocol services.
  - WS-Coordination requires two WSDL interfaces for each protocol: one for the participant's protocol service and one for the coordinator's.
- Adapting the BPEL engine so it may be controlled as the protocols dictate.
  - The protocols require information from and need to be able to affect the process instances. For example, upon reaching a split loop or scope, the engine must wait for the remaining fragments of that loop or scope to start. Upon completing it, it must wait for the others to complete. Upon a fault being thrown in one fragment, the others must be stopped. Therefore, one needs a mechanism by which to control the running process based on messages from the coordination protocol.
- Providing the coordinator with information about the process model.
  - Each process fragment alone does not have enough information on the entire split process. It sees only part of the picture: that in which it is involved directly. For example, a fault handler may exist in one fragment of a scope but not in the scope's other fragments. Such global information is placed, in our approach, in the coordinator to enable it to make decisions that require information involving more than one fragment of the split process. The coordinator also uses it to handle multiple instances of a fragmented scope without collision. This information is the relationship tree and the Default Compensation Order graphs defined in section 5.5.
- Providing the logic at the participant and coordinator sides to implement the protocols.
  - The WSDL interfaces for the protocol must be implemented so that they send and receive the proper messages, as well as perform the appropriate expected behaviour both in the process and in the coordinator. For example, if one fragment of a split scope is ready to start then the participant side of the protocol implementation must send a starting[3] message to the coordinator. Additionally, that scope must be blocked from running any of its activities until the coordinator sends it a start message. Furthermore, the coordinator will not send the start message unless it has received starting from *all* fragments of this split scope.

6.4 Initial Set-Up

This section addresses the two parts needed to set up the connections between the processes and the coordinator: deployment and registration.

[3] Underlining is used to distinguish messages of the coordination protocols from the surrounding text.

6.4.1 Deployment

The 'coordination' element in the deployment descriptor defined in section 5.5 is only used if coordination is needed. It includes the address of the registration service of the coordinator so that the first process fragment to start an instance can register for the split scope protocol. It also includes one 'participant' element defined for each fragment process, containing the endpoint of a default 'starter service'. This service, detailed in section 6.4.3, is used to enable the coordinator to start process instances.

```
 <xs:coordination>
  <wscoor:RegistrationService>
   <wsa:Address>
    …
   </wsa:Address> …
  </wscoor:RegistrationService>
  <xs:participant name="xsd:QName">+
    <xs:starterService>
      <wsa:Address>…<wsa:Address>…
    </xs:starterService>
  </xs:participant>
</xs:coordination>
```

Table 14: Syntax of the coordination element in splitProcessDeployment.

Before being able to run the protocol for split scopes, the coordinator must be aware of the relationship tree and Default Compensation Order graphs for the process model. They may be created at any time before the first process fragment creates an instance.

Several options exist for when and how these are passed to the coordinator. One option is to provide them at deployment time. Providing this information statically is similar to the case in SELF-SERV [BEDS05], where coordinators are statically provided with information such as routing tables, peers and locations. For a more dynamic approach, one may instead pass a serialized representation of these two items, along with the starter service endpoints of the fragments, in the WS-Coordination registration message.

WS-Coordination allows extending the payload of its registration messages based on the requirements of the protocol at hand. For our protocols, the registration messages
are extended to include enough information for the coordinator to retrieve the appropriate relationship tree/DCO graphs and protocol state.

6.4.2 Registration

WS-Coordination's Registration Service is in charge of establishing a connection between the environments hosting the various fragments. This connection is established based on 'protocol handlers' being able to receive and process the corresponding protocol messages.

The registration request message contains an identifier of the protocol a participant will respect, as well as an address for the protocol handler of that participant. One protocol identifier is needed for the scope protocol and another for the loop protocol.

Five more pieces of information are needed to set up the connections. Therefore, we use WS-Coordination's built-in extension mechanism to extend its 'wscoord:Register' element to include this information. One new construct is created for each of these items:

- The split process (iaas:ProcessName) that the activity is in.

- The scope/loop name (iaas:LoopName, iaas:ScopeName), used to determine which split activity this registration is for.
- The name of the participant (iaas:ParticipantName) for which this registration is for, used to determine which fragment is registering and match that to the information in its relationship tree.
- An identifier of the instance of the coordinated work (iaas:CorrelationSetValue). This is the value of the common correlation set. It is used to identify which instance of the split process the message is for.
- A counter that identifies which instance of a scope/loop this is (iaas:Counter). It is used for cases when a split activity may run more than once in the same process instance. The counter distinguishes fragments of different instances of the same scope/loop in the same split process instance so that they are not joined to the same protocol instance.

An example of the extended registration message is shown in Table 15.

Consider the following example for why the counter is needed: a scope is nested in a loop. The scope is aborted in the first loop iteration and then run successfully in the second. Upon a request for compensating the scope, the coordinator needs to be able to tell the two instances apart.

When a scope fragment begins, it sends to the registration service a registration message that also includes the address of the participant protocol handler for that particular scope/loop instance. In return, the coordinator sends a registration response containing the address of the coordinator protocol handler that will receive all messages for that particular scope/loop instance from that particular participant.

The pieces of data identified above are used by the coordinator and the participants to generate the specific addresses for specific instances of each coordinator protocol instance and participant protocol instance. How they generate these addresses is implementation dependent: the WS-Coordination contract is just that if the exchanged addresses are used, the messages will be handled by the right protocol instance. We recommend using WS-Addressing reference properties to encode information about which particular scope instance is at hand. These properties can directly be the data items we have highlighted above (scope name, etc) or middleware generated ones (workflow engine created process instance ID, etc). An example of generated addresses used in our prototype is presented in section 7.5.

```
<Register xmlns="http://schemas.xmlsoap.org/ws/2004/10/wscoor"
 xmlns:iaas= "http://www.iaas.uni-stuttgart.de/wscoor/" ...>
   <ProtocolIdentifier>
    http://www.iaas.uni-stuttgart.de/wscoor/protocolID/LoopProtocol
   </ProtocolIdentifier>
   <ParticipantProtocolService>
      <wsa:Address …>…</wsa:Address>
      <wsa:ReferenceProperties …> ... </wsa:ReferenceProperties>
   </ParticipantProtocolService>
   <iaas:ProcessName
xmlns:nsq="http://example.com">Nsq:ALargeProcess</iaas:ProcessName>
   <iaas:ParticipantName>Fragment2</iaas:ParticipantName>
   <iaas:LoopName>loop1</iaas:LoopName>
   <iaas:CorrelationValue>[prop1=Hello]</iaas:CorrelationValue>
   <iaas:Counter>0</iaas:Counter>
</Register>
```

Table 15: Example of a registration message for a split loop

Traditionally, the identifier of a specific instance of coordinated work (i.e. transaction id) is flowed in the context after calling the activation service. For the problem specifically at hand in this thesis, using the activation service is superfluous hence the instance identifier is placed in the registration message.

6.4.3 On Starting Process Instances

Special care must be taken in starting the fragments of the process itself (as opposed to those of loops and nested scopes). Process fragments may start very far apart in time, depending on when the message that can create an instance is received at each fragment. For the case of process fragments that include split loops and scopes all fragments must be notified in case one of them fails, for both runtime (if another fragment has the necessary fault handler) and auditing purposes. When dealing with a nested loop or scope one can wait for the other fragment of the loop or scope to start before allowing any of them to do so. When dealing with the process itself as the split scope and using coordination, however, one cannot simply wait for all other process instances to start before letting any of their activities run. One example why this is so is that the message that starts one fragment may very well come from another fragment. The coordinator therefore needs to start all process instances once it is known that at least one has started, making a 'lazy starting' not enough.

The starting of fragment instances is enabled in this thesis by using a 'starter service' that can create an instance of the process upon receiving a startInstance message containing the correlation set value. To deal with a possible race condition, we place the requirement that the coordinator must ignore duplicate registration messages and the participant must ignore a startInstance if an instance with the same correlation set value already exists.

Starting an instance in this manner will result in all fragments of the process registering, starting, and being in the active state once one of them starts. It will not cause activities to occur out of order because in BPEL the first activities to occur are the 'create instance receives'. Receive is a blocking activity in BPEL, so these instance creating receives will block and wait for the actual external messages before allowing work in the body of the process to execute. Furthermore, since the split process was the result of splitting a larger BPEL process then starting all fragments at the same time is equivalent to having created one instance of the larger process.

6.5 Join Failures

Recall that a split loop or scope may have a join condition and that if a join condition evaluates to false, then the activity throws the 'join failure' fault. Suppressing this fault using 'suppressJoinFailure="yes" leads to the activity being disabled and its outgoing links being sent out with a false value if its join condition evaluates to false.

Since in this thesis we support splitting loops and scopes, a split activity that will be skipped can only be skipped if it fires a join failure or if it is in a scope that is faulting. Handling join failures and faults properly are then all that is needed to ensure that a fragment does not hang if it is known that a sister fragment in another participant will never be executed.

6.5.1 Handling the Join Failure

A join failure thrown by a split activity whose suppressJoinFailure attribute is set to 'no' is simply another BPEL fault and does not require special handling in the coordination

protocols. However, if this attribute is set to 'yes' for a split activity, the protocol contains messages to propagate its suppression to fragments. This simplifies the protocols and enables built-in support for a modification in the join failure behavior made in BPEL 2.0. This modification is that an activity with suppress join failure set to yes suppresses only its own join failure and not that of any of its nested activities; whereas in BPEL 1.1 it would suppress any join failures that can reach the activity, including those thrown from its nested activities. The protocols support the BPEL 2.0 join failure behavior directly and that of BPEL 1.1. For the latter case, a simple modification may need to be made to the process definition.

The simple modification for BPEL 1.1 processes is made if it (1) contains any split scope/loop whose suppressJoinFailure is set to 'yes' and (2) the split scope/loop contains nested activities where this attribute is set to 'no' and whose join failure fault can reach the split scope/loop. If this is the case, then every split scope/loop for which these properties hold is surrounded with a new scope having a fault handler for the join failure fault and containing an 'empty' activity. The 'suppressJoinFailure' attribute value on the split scope/loop is changed to 'no'. The new scope will be split due to the rubber band effect. It will not throw a join failure because it will not have links at its own boundaries.

The restriction that the join condition is the conjunction of the local join conditions could be eased without loss of generality by providing the coordinator with the join condition definition, sending the status of each incoming link with the starting message from each fragment, and evaluating the condition at the coordinator side. However, this option was not chosen because although it is less restrictive, it obscures the behavior of the fragments by placing an important piece of the business logic in the coordinator.

6.6 The Base of the Protocols

In this section, we provide a 'skeleton protocol' that highlights and encodes the basic behavior common for both loops and scopes, forming the basis for their protocols. The skeleton protocol is not used directly in this thesis. If one were to support distributing other compound activities in BPEL through coordination it could serve as a base for those protocols as well. In fact, it could be used as-is to drive a distributed flow activity, but other BPEL compound activities require more distributed control than this handles.

The state machine in Figure 53 shows the work that has to be performed by the coordinator to run the fragments of a distributed activity as one: the beginning and end of all fragments have to be synchronized. Words in italic correspond to messages of the protocol itself. Unless noted otherwise, a 'send' state sends to *all* fragments of the split activity. Conditions on a transition are either local or they are messages received from participants (italics).

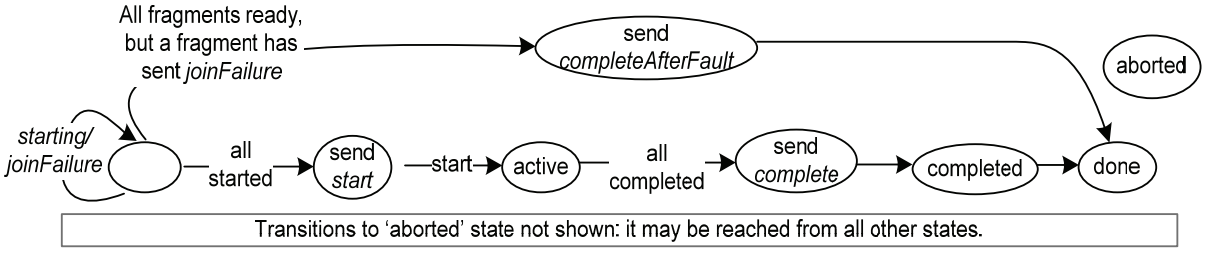


Figure 53: Coordinator behavior for the skeleton protocol

Once a fragment is ready to run (it has been reached in the navigation in its own process fragment), it notifies the coordinator. Once all fragments have stated that they are ready, the coordinator notifies them that they may start. Then, as each fragment completes the work in

its own fragment, it notifies the coordinator. Once all of the fragments have completed their work, the coordinator notifies them that they must now complete. At any point in time, the protocol may be aborted due to a fault. The aborted state may be reached from all other states.

The messages that have to be exchanged between the coordinator and each participant in order to drive the necessary behavior are shown in the state machine in Figure 54. These correspond to the points where the participant or the coordinator sends a notification regarding a particular lifecycle event.

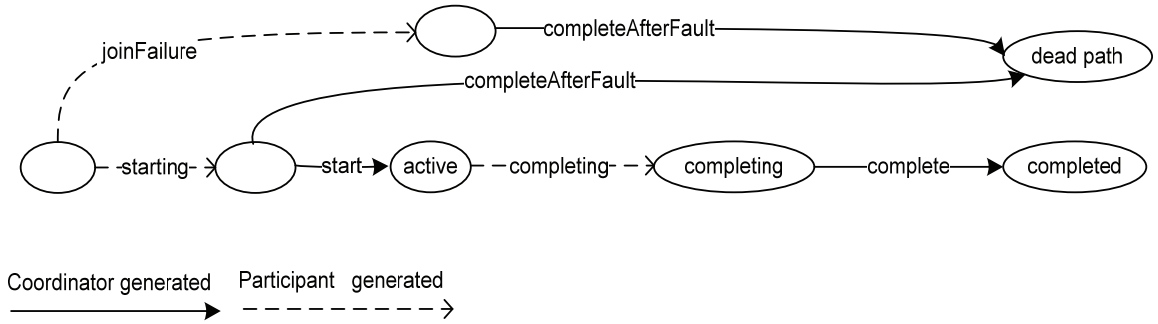


Figure 54: Participant-coordinator messages of the skeleton protocol

Consider the part of the skeleton protocol used to handle join failures. This occurs as follows. A participant whose fragment encounters a join failure upon reaching a fragment of a scope or loop, checks its join failure attribute:

- If it is set to 'yes', the participant sends a joinFailure message to the coordinator instead of the starting message. The coordinator waits for all fragments of the loop/scope to be ready (have sent starting or joinFailure), and if any had sent joinFailure, the coordinator sends a completeAfterFault message to all the fragments.
- If it is set to 'no', then the participant does not even register for the protocol. The failure causes the BPEL engine to throw the join failure just like any another BPEL fault thereby causing the faulting scope fragment to be disabled.

The WSDL files of the scope and loop protocol services, including XML Schemas of the protocol messages, and sample message instances are in [PALU07].

6.7 The Split Loop Protocol

The fragment responsible for the loop condition can be specified in one of several ways: using an attribute on the split loop (is-responsible="yes|no") or by referring to a 'special variable' (on each 'non responsible' fragment) that gets populated through a message from the coordinator with the value of the loop condition upon its evaluation. An implementation can choose to take any such approach for a split process. Currently, we use the 'is-responsible' attribute.

Figure 55 shows the behavior of the coordination protocol between a loop fragment and a coordinator. Figure 56 shows the behavior of the coordinator coordinating all fragments of that loop. Notice that any fragment can be the first to start, but the entire loop is only allowed to start once all fragments are ready. The same occurs for completion and iteration. In both these figures, states and transitions already in the skeleton protocol have been grayed out to highlight the new loop-specific ones.

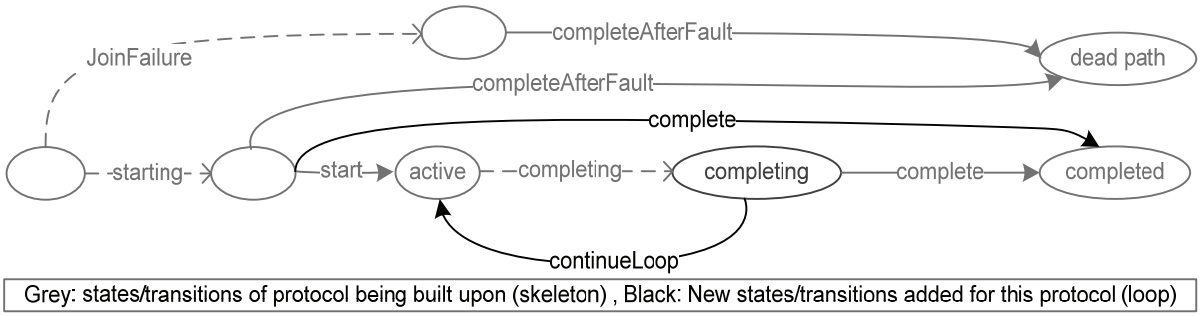


Figure 55: Participant-coordinator messages for the loop protocol

6.7.1 Participant-Coordinator Messages

The protocol takes place between each instance of a fragment of a loop and the coordinator (Figure 55). This section will focus on the behavior that is new with respect to the skeleton protocol above. The coordinator gets a message that a loop is starting and containing the value of the while condition at that fragment. If the fragment is not the one responsible for the condition, then that value is omitted.

The coordinator eventually sends back either a complete message if the loop condition is false or a start message so that the participant actually starts running its piece of the loop. Once the participant completes an iteration, it sends a completing message that, for the responsible fragment, will again contain the value of the loop condition.

The coordinator then sends either continueLoop if the loop condition is true or complete if the loop condition is false. The participant will then start a second iteration in the former case or navigate out of the loop in the latter case.

6.7.2 Participant Behavior

The behavior necessary on the participant side involves being able to listen, react to, and influence the process instance's behavior. The participant side behavior of the protocol needs to know when a loop fragment has started or had a join failure and to block the activity when this occurs. It needs to be able to unblock the fragment and let it either iterate or complete based on messages from the coordinator, instead of the value of its local loop condition.

6.7.3 Coordinator Behavior

Having explained the coordination messages between a fragment and the coordinator, we now look at the behavior of the coordinator in coordinating all fragments of one loop for one instance of that loop.

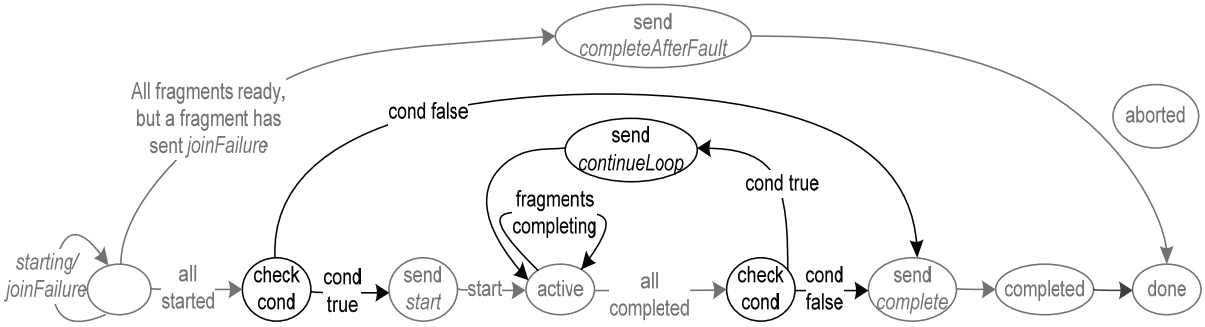


Figure 56: Coordinator behavior for the loop protocol

The protocol takes place between each instance of a fragment of a loop and the coordinator. The coordinator gets a message that a loop is starting. If the message is from the responsible fragment, then this message contains the Boolean value of the condition at that fragment. The coordinator eventually sends back either a complete message if the loop condition is false or a start message so that the participant actually starts running its fragment of the loop. Once the participant completes an iteration of the loop, it sends a completing message. If this message is from the responsible fragment, then it contains the value of the loop condition.

The coordinator then sends either continue if the loop condition is true or complete if the loop condition is false. The participant will start a second iteration in the former case or navigate out of the loop in the latter case.

6.8 The Fragmented Scope Protocol

Splitting a scope involves splitting its fault handling and compensation handling behavior. Fault and compensation handlers themselves may be split, in addition the scope body. Fault handling differs from default compensation handling in several ways, one of which is that faults propagate up the scope hierarchy, with the order of aborting siblings being irrelevant. Default compensation, on the other hand, occurs one level of scope nesting at a time and in an order that reverses control dependencies between peer scopes.

BPEL fault and compensation handling, however, are heavily related. Compensation on a scope *s* can only be triggered from: (1) a fault handler in the parent scope of *s* or (2) a compensation handler in the parent scope of *s*. The scope protocol will therefore handle both. Since the behavior is quite complex we will break it down: similar to how we started with a base protocol and then added to it, we start first with the part of the protocol that deals with fault handling and then add compensation handling to it. In fact, if one had a split process without any explicit compensation handlers, then the subset of the scope protocol in section 6.8.2 will be adequate.

6.8.1 Data Concerns

Data provided with a fault is sent to the coordinator. The coordinator then sends it to all fragments that have a corresponding fault handler. A fault handler at a fragment that has a 'faultVariable' attribute will then save this message in this variable. For example, the value of variable 'x' is saved in variable 'y' if the fault was caused by a <throw faultName="ns:flt" variable="x"/> and caught by a <catch faultName="ns:flt" faultVariable="y">…</catch>. The coordinator is not involved in data propagation in a split loop or in a split compensation handler, because this is done by the data dependency propagation patterns in Chapter 5, sections 5.9 and 5.10.

6.8.2 Fault Handling in the Scope Protocol

The scope protocol enables one to handle faults across the fragments of a scope by using the coordinator. This section describes the split fault handling capabilities provided by the protocol.

Searching For the Fault Handler

In this section, we present the algorithm that finds the fault handling scope for a particular fault. The search for a fault handler for a given fault in BPEL is the same as the search in the process meta-model in Chapter 3, specifically in section 3.2.9. Therefore, we derive the search algorithm used in the coordinator from the one in section 3.2.9. The differences when

using coordination are that the scopes are the scopes $S_{rt} \cup S_{ns}$ of the relationship tree instead of the scopes in the hypergraph of the meta-model; the edges C_{rt} are child-parent edges of the relationship tree instead of those of the graph *G* of the hypergraph; the information of whether the fault originated in a fault handler of the first scope is known from the message that arrives in the coordinator and does not need to be checked using the placement of the faulting activity.

Consider s_r to denote the root scope. First, we define a function called $parent_{rt}$ that retrieves the immediate parent of a given scope using the scope tree:

$$parent_{rt}(s) := \pi_2(c), \text{ where } c \in C_{rt} : c = (s, s')$$

The domain of $parent_{rt}$ is $S_{rt} \cup S_{ns} - \{s_r\}$ because the root scope has no parent.

The function $m(n, f)$ in section 3.2.9 determines whether a fault handler could handle a fault of name *n*. Here, we need to know whether the scope itself has a handler for this fault. We therefore define a similar function, taking the name of a fault and a split scope node as input, to check whether a particular scope has a handler for the fault:

$$m_{rt}(n, s) = \begin{cases} true & n \in \pi_2(s) \\ false & \text{otherwise} \end{cases}$$

Recall that one must skip the parent scope when a fault is crossing a fault handler boundary and that we are using '⊥' to represent 'not found'. In section 3.2.9, the next scope to search was found by a function *g(s,a)* that takes the faulting activity and a scope as input and finds the next scope based on whether the fault was crossing a fault handler boundary. It checked for this based on whether the faulting activity belongs to the activities of the fault handler of the parent. The coordinator uses a similar function $g_{rt}(s)$ that takes a scope node from the relationship tree as input and returns the next scope to search by using the relationship tree. The node does not contain the activities of the scope, but it does encode whether or not the scope is in the fault handler of its parent in the Boolean value of the node tuple. This determines whether to skip the scope or not. Therefore, we define $g_{rt}(s)$ as follows, recalling that for a scope node *s*, $\pi_3(s)$ is a Boolean stating whether or not the scope is in the fault handler of its immediately enclosing scope:

$$g_{rt}(s) = \begin{cases} parent_{rt}(s) & \neg\pi_3(s) \\ g_{rt}(parent_{rt}(s)) & \pi_3(s) \wedge s \neq s_r \\ \bot & otherwise \end{cases}$$

The scope where the search starts is either the one where the fault occurred or one of its ancestors. The coordinator is informed upon the occurrence of a fault by either a faulted or a faultedInHandler message from a participant. The coordinator then needs to search for a matching fault handler. Let the scope node in the relationship tree corresponding to the split scope that receives the fault-related message be s'_{rt_0}. Let s_{rt_0} be the BPEL scope where the search is to start, *n* the fault name in the message from the participant, and *h* a Boolean value that is true if the message was faultedInHandler and false if it was faulted. Therefore:

$$s_{rt_0} := \begin{cases} s'_{rt_0} & \neg h \\ g_{rt}(s'_{rt_0}) & h \wedge s'_{rt_0} \neq s_r \\ \bot & \text{otherwise} \end{cases}$$

In section 3.2.9, a function $y(n, s, a)$ took the fault name and the scope to start searching from and returned a matching fault handler if one was found. Here, we need the scope that has a matching fault handler. Therefore, we define a similar function $y_{rt}(n, s)$. The function $y_{rt}(n, s)$ takes as input the name of the fault and the scope where the search must begin and returns either the fault handling scope or $\perp$ if none is found:

$$y_{rt}(n,s) := \begin{cases} s & m_{rt}(n,s) \wedge s \neq \perp \\ y_{rt}(n, g_{rt}(s)) & \neg m_{rt}(n,s) \wedge s \notin \{s_r, \perp\} \\ \perp & otherwise \end{cases}$$

Finally, the scope that can handle the fault is found as follows: The message provides both *h* and *n* and is addressed to the protocol of s'_{rt_0}. At the coordinator, s_{rt_0} is found using these values and the scope tree. Then, $fh_{rt} := y_{rt}(n, s_{rt_0})$ returns either the fault handling scope if one is found and $\perp$ otherwise.

Participant-coordinator Messages

The messages between the coordinator and the participant for coordinating a fault in the absence of compensation are shown in Figure 57. Just as in the case of loops, one must also wait for all fragments to start before allowing any particular one to actually begin executing its nested activities. The completion of the scopes is synchronized, also similar to the case of loops. A BPEL scope, however, may complete either upon having run its activities successfully, or upon having run one of its fault handlers successfully. Both cases lead to the scope having completed and its outgoing links being fired normally. However, in the latter case, the completion is considered abnormal and the engine needs to distinguish between the two, especially if compensation is involved. The result is that the fault handling protocol propagates a fault from one fragment to all other fragments of the scope that can handle it, and synchronizes the completion of the (split) fault handler.

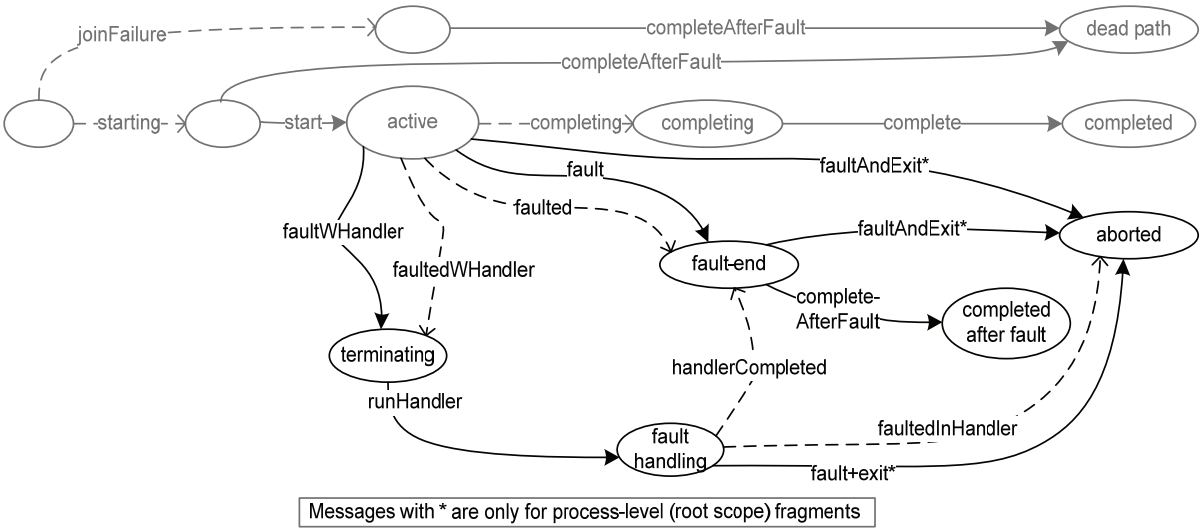


Figure 57: Participant-coordinator messages for fault handling subset of the scope protocol

For this protocol, all messages that start with fault include the fault QName and an optional fault variable value. All messages that are related to compensating a particular scope include the name of that scope.

Consider first the process fragments. A fragment will either start normally or be started by the coordinator. In the first case, it sends starting. In the second case, not shown in the figure, the starting occurs before the protocol starts and is targeted at the starter service not the participant. Here, the coordinator sends a startInstance message containing the values of the correlation set to the fragment's 'starter service' as provided in the deployment information. As each fragment creates its process instance, the process-level scope will send a registration message to the coordinator and everything continues as for any other (non-process) scope.

A non-process scope sends starting when it is reached by the navigation. After all fragments of a scope have started, the coordinator sends a start message to that fragment allowing it to actually start. A fragment stays in the active state while running its nested activities and as long as no fault has occurred anywhere within the split scope. In the normal case, the next step would be to complete as was done in the case of loops (from active, to completing, to completed state).

In the case of a process-level scope fragment, the participant may get a faultAndExit message from all states in which a fault may have been thrown: 'active', 'fault handling', and 'fault-end'. This occurs in case a fault has been encountered elsewhere and cannot be handled by any scope up to and including the level of the process itself. In other words, if y_{rt} does not return a scope. This message cannot appear in the protocol of a scope that is not the process itself.

Another step out of the 'active' state occurs if another fragment faulted, and the scope of *this* fragment has a handler for the fault, then the coordinator would send a faultWHandler message. If another fragment faulted and *other* fragments have handlers but not this one, then a fault message is sent. If a fault is thrown and there are no handlers in *any fragments of this scope*, the coordinator will throw the fault to a parent scope's protocol and that is where this case will be handled.

From the 'active' state the participant may also itself encounter fault. In this case, it sends a faulted message if it faulted and has no handler for the fault, or a faultedWHandler message if it faulted and has a handler for that fault on the *fragment itself*. It then waits for the coordinator to confirm that it should run the handler (runHandler message).

After receiving a runHandler message, the participant runs the corresponding handler. The handler itself then either completes (handlerCompleted) or faults. If the handler completes, then the coordinator waits for any other fragments of that handler to complete and then sends a completeAfterFault message. Otherwise, if the handler faults, the participant sends a faultedInHandler message that is used by the coordinator to lookup an appropriate handler using the fault handler lookup algorithm.

Participant Behavior

Here as well, the participant side needs to be able to react to and affect the process instance's behavior. It needs to be able to detect the scope fragment starting or encountering a join failure and then block it until it receives the start message. It needs to detect a fault and override local fault handling behavior. It needs to be able to produce a fault in a process instance even if that fault did not originate in that fragment at all.

Coordinator Behavior

Having described the participant-coordinator messages, we now describe the corresponding behavior of the coordinator when running an entire split scope; that is, when interacting with all of its fragments from start to finish. Figure 58 illustrates the behavior of the coordinator.

The coordinator uses the relationship tree for identifying when all participants for a particular scope have registered, to search for handlers, and to relate endpoints to participant names. The state that is filled with grey ('Send fault to handling scope's coord. behavior) and the underlined transition out of the active state are highlighted because they represent communication between the behaviors of multiple scopes in the coordinator. The protocol can be aborted at any time by a parent scope's protocol. Again, transitions are not drawn to it from every state as not to clutter the figure.
Now, we take a closer look at the protocol. First, if a fragment is the process itself then one starts with the startInstance message that will cause all fragments to create an instance and register if they have not already done so. Then, registration (not shown in the figure) takes place for the rest of the fragments and the protocol itself can begin. The behavior for any scope is that the fragments send starting and the coordinator waits for all fragments to send this message before it sends start.

A fault from a fragment (faulted/faultedWHandler) causes the coordinator to search the scope tree for a handler. If the search does not yield a scope, then faultAndExit is sent to the fragments of the process level scope. Otherwise, if the handler is on the scope that faulted (in any of its fragments), then the same fault is thrown by the coordinator to those fragments so they can handle the fault locally. If the message had been faultedWHandler, then the coordinator sends runHandler to that fragment, fault to any other fragments of the scope that have no handlers for the fault, and faultWHandler to any other fragments that have a handler. It follows with runHandler to those fragments with a handler.

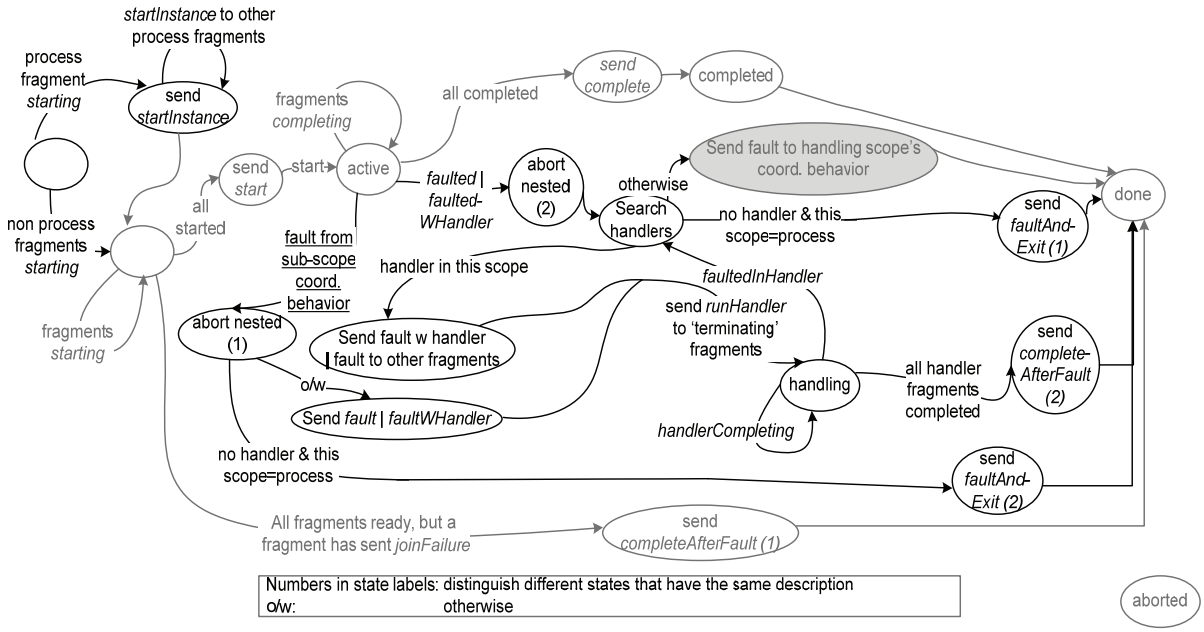


Figure 58: Subset of coordinator behavior for fault handling in the scope protocol

If the handler is in a parent scope, then the coordinator will send a local fault signal (grey filled state) to the coordinator behavior of the scope that has the handler. This signal appears

in a scope's coordinator behavior as the underlined transition out of the 'active' state. It will in turn abort all nested scope behaviors, and then send a message to the participants causing the fault to appear in all the fragments of the handling scope. At each fragment, such a fault will abort nested scopes. In this way, both the participant and coordinator sides of the nested protocols are stopped for all fragments without needing to send protocol abort messages to all. Default compensation may take place at this point as part of termination and of having caught the fault possibly several scopes higher than where it had been thrown. Next, the fault will start the fault handler fragments, and the protocol continues as shown.

If the fault cannot be handled by any scope, then the faultAndExit signal is sent to the process-level fragment ending the instance in every process fragment.

The result is an interleaving of local BPEL behavior and coordinated BPEL-inspired behavior (fault handler lookup, etc.) to achieve the execution of cross-process fault handling scopes.

The search for handlers occurs in the coordinator using the knowledge of which scope the fault was caused in, the relationship tree, and the algorithm for finding a fault handler. Note that this means that the fault and faultWHandler messages are only propagated from the coordinator to a scope that has a handler for the fault (regardless of which fragment(s) the handler(s) are in).

Finally, we also require that messages arriving at a scope whose protocol has been aborted at the coordinator side are ignored. This may occur if the message arrives between when the coordinator aborted it and when the coordinator sends a fault to it or its parent scope causing the participant to locally abort as well.

Race Conditions

Race conditions are introduced where one has a state with transitions that can be initiated either by the participant or by the coordinator. Looking at the protocol in Figure 57, one can notice possible race conditions in the 'active' and 'fault handling' states.

For the 'active' state, there is a race between faults happening in different fragments, and a race between the participant wanting to complete and the coordinator trying to send it a fault. The former race already exists in BPEL, where faults may occur in parallel branches at nearly the same time. Only one fault is dealt with, the first one to reach the scope. Here, that will be the first one to reach the coordinator.

To ensure this, we place the rule that, for the 'active' state, the coordinator messages win over the participant messages. Therefore, if the coordinator has sent out a fault, faultWHandler, or faulAndExit message, then completing, faulted or faultedWHandler from the participant are ignored. The case of completing is especially relevant since the coordinator may in fact have received it but still decided to send a fault message because another fragment had faulted. Furthermore, notice that a fragment cannot start its handler even if the fault came from it and it has a handler. It needs to wait for the coordinator to send runHandler. The reason is that if the participant could immediately start its handler and there was in fact a race, then it could start a handler for a different fault than the coordinator has deemed the winner.

For a race between handlerCompleted and faultedInHandler and the faultAndExit messages, faultAndExit wins. Therefore, the coordinator behaviour will ignore the handlerCompleted or

faultedInHandler message if it has already sent a faultAndExit. Then, the race is not a problem because it only occurs for the process level scope with the faultAndExit message. If this message arrives late, then it can still be accepted by the protocol because it can also occur in the next state and also leads to 'aborted'.

Relation to the Loop Protocol

Upon receiving a fault, a participant and the coordinator itself must abort all nested loop protocols. The nested loops are known from the relationship tree at the coordinator side. At the participant side, the nested loops are known from the process definition.

6.8.3 Adding Compensation: The Full Fragmented Scope Protocol

This section adds compensation to result in the full protocol for fragmented scopes. Recall from Chapter 5 that we make the assumption that compensation must not fail.

Compensation and its Relation to Faults and Termination

Upon faulting, a scope first stops its nested activities by terminating them. Termination was named differently between BPEL 1.1 and BPEL 2.0: In BPEL 1.1, a fault was created that could be sent from the parent scope down to its child scopes and for which a fault handler could perform termination work. The default was to stop nested activities and compensate nested scopes in default order. In BPEL 2.0, the same behavior is now renamed using the term 'termination handler'.

This thesis considers only default termination: stop nested activities all the way to the leaves and then compensate nested scopes in default order. BPEL does not impose order on which scopes between a set of active peer scopes is to be terminated first.

The order of events in case of a fault in a scope is:

- Receive a fault in a scope,
- Terminate nested running activities of that scope:
  - If a running activity is not a scope, terminate it according to the BPEL specification (wait for it to complete or simply abort it).
  - Otherwise, terminate its running immediate children and then perform default compensation on its immediate, completed, scope children.
- If an explicit fault handler is not present, then compensate in default order the immediate completed children of the scope where the fault has been received. Rethrow the fault.
- If an explicit fault handler is present, then run that handler.

Termination in BPEL occurs in innermost first order. Therefore, when running compensation for the purposes of termination, compensation is run in innermost first order of scopes nested in the scope being terminated. The order is determined by the scope tree and the state of each scope's protocol.

Compensation Handlers in Fault and Compensation Handlers

Handlers in a fault or compensation handler are only available for the life of the enclosing fault/compensation handler. Therefore, any installed compensation handlers are uninstalled once a handler completes. After that, they can no longer be reached or called.

Compensation Concerns for Splitting Scopes

Two different concerns exist for split compensation, illustrated in Figure 59:

- Determining and enforcing the compensation order. This cannot be handled purely locally because child scopes may be in several fragments. Consider Figure 59 (a): If the encompassing large scope wishes to compensate its children in default order, neither fragment alone has enough information whether the child on the left or the child on the right should be compensated first.
- Running a split compensation handler of a particular split scope once it is determined that it needs to be compensated. Consider Figure 59 (b): both fragments of the larger scope must stay in the compensating state until both of their handler fragments have completed their activities.

The first concern is handled by providing the coordinator, upon deployment, with the Default Compensation Order graphs and the relationship tree. The coordinator will use these DCO graphs to drive the compensation order at any time that default compensation needs to be performed for a particular scope. The BPEL engine is not allowed to calculate compensation order locally for a split scope or any of its immediate children: The order in which to compensate scopes whose parent is split *must* come from the coordinator. When a participant requests compensation from the coordinator upon activation of a 'compensate' or 'compensateScope' activity, the coordinator determines which scopes have to be compensated next and which handlers have to run. When done, it notifies the requesting scope that the compensation has completed.

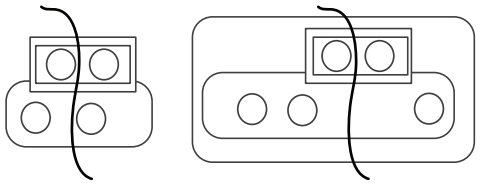


Figure 59: The default order and split handler concerns for compensation

The second concern, the split compensation handler, is handled in a similar manner as running a split fault handler: the coordinator coordinates the start and end of the handler between its fragments.

Scopes that are not split may be required by the coordinator to compensate, because the request to compensate comes from a scope's parent and that may itself be split. In such cases, we pass the request to run the handler through an ancestor split scope and therefore avoid requiring non-split scopes to run a protocol. Thus, a compensation request to the coordinator is always requested by a split scope and sent to that scope's protocol.

Effects on BPEL Engines for Scope Compensation

The behavior of the engine is affected with respect to the compensation enablement for scopes that are involved in these protocols. We consider two cases: split scopes and non-split scopes.

Split scopes: Compensation for split scopes can only be triggered from the coordinator; thus, the BPEL engine running such scopes is disallowed from triggering their compensation handlers. The engine can only start an explicit compensation handler of a split scope and can only do so upon receiving a request from the coordinator.

The default fault handler for the 'enclosing scope fault' (or in BPEL 2.0, the default 'termination handler') is defined in the specification to be a 'compensate' activity. However, a split scope fragment cannot trigger compensation itself. Therefore, an engine *must* treat the work of the default fault handler as simply an empty activity and allow the coordinator to drive the actual default compensation behavior.

Non-split scopes: This section is only concerned with non-split scopes that belong to the scope relationship tree. For any other non-split scope, all behavior in the engine is unchanged. Thus, the unsplit scopes in this section are the members of the set S_{ns} in section 5.5.2. The default fault handler for the 'enclosing scope fault' of a non-split scope is left unchanged.

The engine may also only start the *explicit* compensation handler of a non-split scope upon request from the coordinator. However, both the engine and the coordinator are allowed to trigger the default compensation handler of such a scope. Specifically, the engine will do so if default compensation is required by the scope's termination behavior.

Non-split scopes do not themselves participate in a protocol instance. The request to compensate them and the notification of their handler's completion will reach the process fragment through the protocol of the scope that is requesting their compensation.

The coordinator does not know whether a non-split scope has completed successfully or not. The engine is required to immediately send back the handlerCompleted message upon receiving a request to compensate a non-split scope that the engine knows has not completed successfully at the time of the request. In other words, it performs a no-op in this case.

Participant-coordinator Messages

In this section, we add the states and transitions needed for compensation to the subset of the scope protocol already described. The result is the full scope protocol shown in Figure 60.

First, several states are added to enable a split scope to itself be compensated. A scope can only be compensated after it has completed, so these states follow after the 'completed' state. Split scopes can only be compensated through a command from the coordinator, as we mentioned earlier. Therefore, the compensate message comes from the coordinator and sends the protocol from the state 'completed' to the state 'compensating'.

The coordinator sends compensate to all fragments of a split scope that it intends to compensate, whether that fragment has a handler or not. Any scope fragment with all or part of the compensation handler sends compensationHandlerCompleted upon completing the work in its part of the handler. Once the coordinator hears back from *all* fragments that have part of the compensation handler, it notifies all the scope's fragments that compensation has completed by sending the finalizeScope message.

Next, we consider states added in order to request and process compensation of child scopes. In BPEL, compensation can be triggered by a compensate activity. This activity may refer to a specific scope by name. If it does not, then the result of running it should be to compensate all immediate child scopes in default order. This activity can only exist in a fault handler or in a compensation handler. Therefore, the states for dealing with child scopes are reachable from the fault-handling and the compensating states. From either of these states, the participant can request that a child scope be asked to compensate, or that default

compensation must be performed. The coordinator notifies the child when the compensation completes.

This behavior does serialize the compensation of child scopes in one fragment even if they could be run in parallel. However, this is not in violation of the standard because BPEL semantics don't dictate that parallel activities must run in parallel.

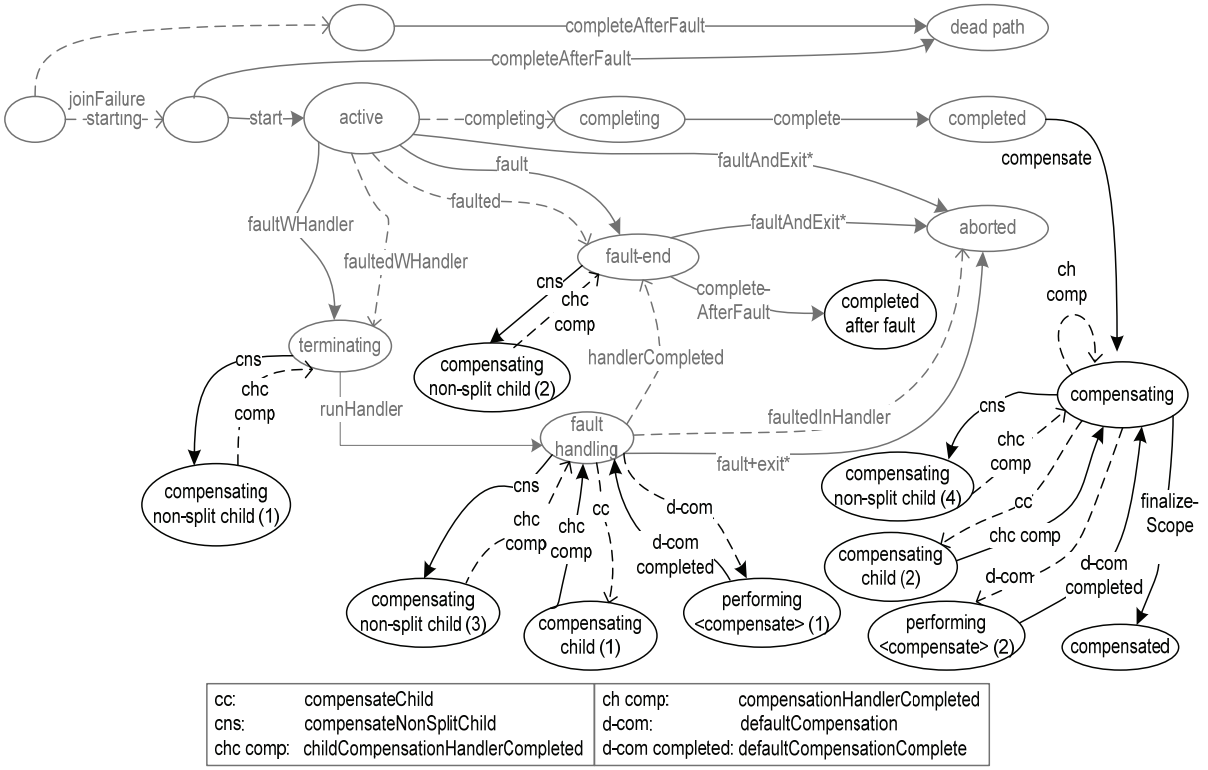


Figure 60: Participant-coordinator messages for the full scope protocol, including compensation

The protocol also has a set of transitions and states that are present to enable the coordinator to request that a fragment run the compensation handler of a non-split child scope. These will only be the non-split child scopes that are relevant to the compensation order and therefore present in the relationship tree. The protocol shows three states labeled 'compensating non-split child' that are there for this specific case. If the child to be compensated is not in the 'completed' state, then the engine is to treat this request as a no-op and send back immediately the compensationHandlerCompleted message.

Finally, notice the 'terminating' and 'fault-end' states. When in these states, the coordinator is performing compensation due to termination on the children of this scope as will be detailed in the 'Coordinator Behavior' section below. Split scopes get a compensate message from the coordinator when it needs them to run their compensation handler. However, recall that the coordinator cannot interact directly with non-split scopes whose parent is split. Such scopes therefore get their compensation request to compensate through the split scope that will handle the fault. This scope is chosen to send the compensation request because it is the smallest ancestor scope of the non-split scope whose protocol has not been aborted due to the fault.

The transitions for handling this issue are seen on the terminating and fault-end states. They allow the coordinator to send a compensateNonSplitChild request to the scope protocol of the

fragment that contains the required non split child. They also allow the coordinator to receive the notification from that fragment once the handler completes.

For the rest of this thesis, we will use the term compensation related messages to refer to those messages that lead to and from any of the states: 'compensating', 'compensating non-split child', 'compensating child', or 'performing compensate'.

Note: Some transitions are not shown in the figure for simplicity: The state 'compensating non split child' that is reachable from 'fault-end' also has a faultAndExit transition leading to the aborted state. The reason is that the fault handler (or another fragment of it) may fault while compensation is taking place. Also, all the compensation related states that are reachable from 'fault-handling' have a faultAndExit transition and a faultedInHandler transition leading to the 'aborted' state. In case of a race condition, the fault relegated messages win over any compensation related messages.

Coordinator Behavior

Recall that BPEL compensation occurs due to default fault or compensation handling, or due to explicit fault or compensation handling. Additionally, fault handling always starts with termination. Termination itself may also trigger default compensation.

In this section, we describe the work done to actually perform the coordinated compensation. First, we describe the behavior at a high level, shown in Figure 61. Items already in the fault handling and skeleton subsets of the behavior are grayed out for ease of reading. States involving communication with other scopes' coordinator behavior are again shown using a grey fill and transitions that do so are again underlined.

The first thing to notice is that compensation-related work only takes place either after a scope has completed or while a fault is being handled. Consider the three states in the figure that state 'compensation due to terminate/rethrow'. Compensation work by the coordinator will take place after the coordinator sends a fragment a fault or faultWHandler message. Once the termination is completed, the coordinator sends the runHandler message to the fragments that have a fault handler. If this is the process and there is no handler, it first sends a fault message, then performs the termination related compensation and follows with faultAndExit.

Next, we consider the states coming out of the 'handling' state. While fault handling, the coordinator may get a request for either explicit compensation of a particular child or default compensation.

Finally, we consider the states coming out of the 'completed' state. These are the states where the scope itself needs to be compensated. If there is no explicit handler, then the coordinator performs default compensation on it and notifies the coordinator behavior of the scope that requested the compensation of this scope that it has completed this work.

If there is an explicit handler, then all fragments are asked to compensate. At this point the compensation handler is running, perhaps at several fragments. Similarly to the fault handling case, they may either complete or may request compensation of a child scope or default compensation of this scope. Once all have completed, then the coordinator behavior of the scope that requested this compensation is notified of the completion, and the scope fragments are sent the finalizeScope message.

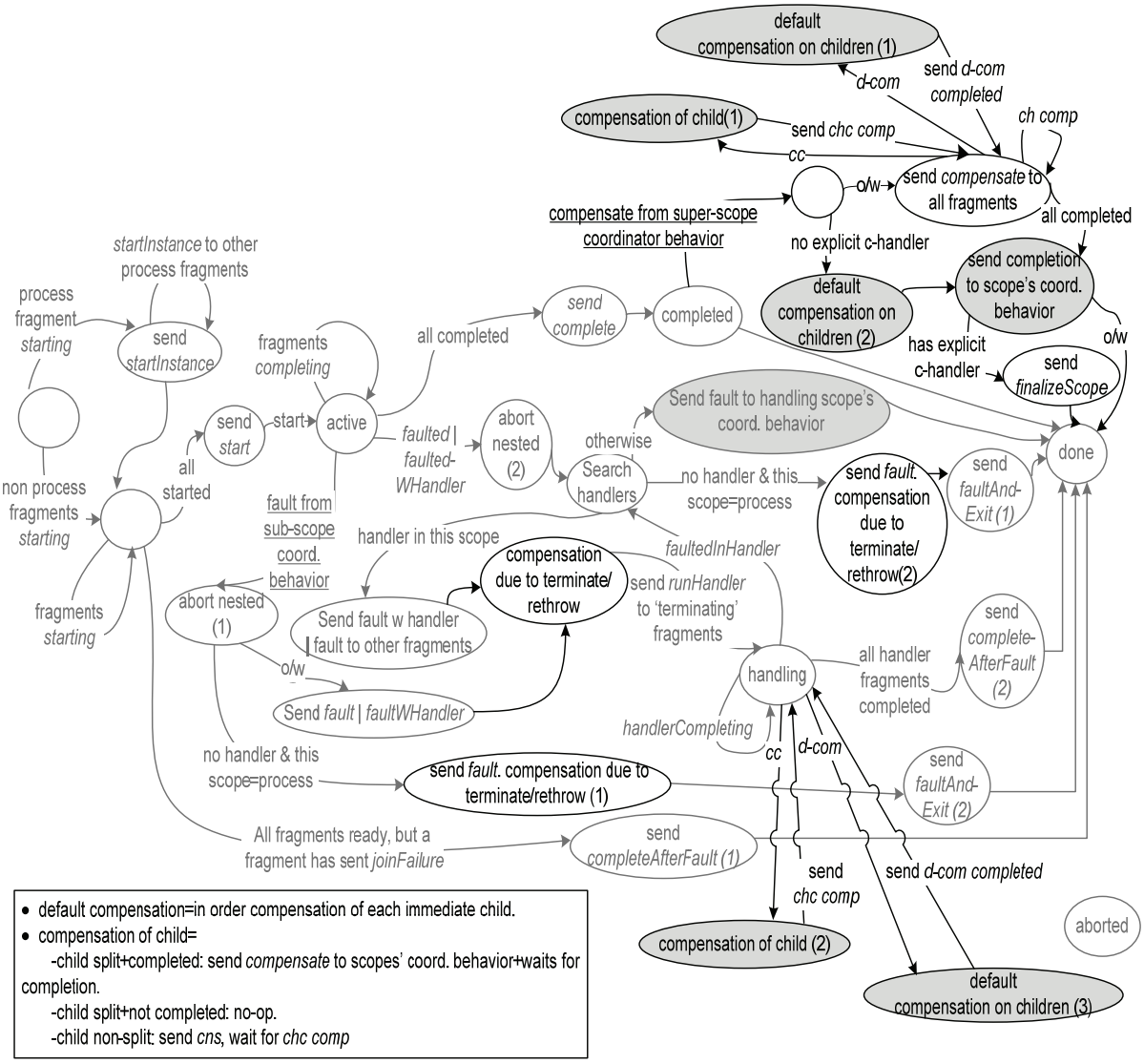


Figure 61: Coordinator behavior for the complete scope protocol, including compensation

The details of what occurs in the states labeled 'compensation of child', 'default compensation on children', and 'compensation due to terminate/rethrow' will be explained in the following sections on the details of compensation in the coordinator.

Performing default compensation using the DCO graphs

For the rest of this chapter, performing default compensation on a scope means doing so as defined in this section. Performing default compensation on a scope *s* consists of compensating its immediate children in default order. The order is prescribed by the DCO graph of *s*, which includes any additional graphs due to loops in *s*. Performing default compensation therefore corresponds to navigating this graph.

Walking a DCO graph consists of running it as if it were a BPEL process with all transition and join conditions set to 'true()', as was the case for the COGs in Chapter 3. Here too, ScopeLoopsQs are created at runtime to keep track of instances of scopes and loops nested in

each other; however, this time they are created by the coordinator for the scopes that have a DCO graph.

One difference between a COG and a DCO is that every scope node in a COG must have an explicit compensation handler whereas a DCO scope node may not have one. Therefore, a DCO is navigated in the same way as a COG except if a scope node is reached that does not have an explicit compensation handler. In this case, default compensation of the corresponding scope must be called. It, in turn, consists of navigating the DCO of that scope.

The following summarizes how a DCO is navigated:

- Upon activating a scope node, *A*:
  - Check whether the current instance of the scope corresponding to *A* is in the 'completed' state.
  - If not, skip *A*.
  - Otherwise, check whether *A* has an explicit compensation handler:
    - If it does, request compensation of the scope.
    - Otherwise, walk the default compensation graph(s) of *A*.
  - The scope node in the compensation graph is considered 'completed' for navigation purposes after it had been reached and either it is found to be not in the completed state, or its (explicit or default) compensation handling has completed. Navigation then continues by following the edges leaving *A*.
- Upon activating a loop node:
  - The graph of the loop itself is navigated, as many times as the loop has run for that instance of the scope or loop in which the loop node is nested (using the ScopeLoopQs).
  - The loop node is considered 'completed' for navigation purposes once all such runs of the loop's graph have completed. Navigation continues by following at the edges leaving the loop node.

Coordinating compensation due to the compensate activity

A compensate (or in BPEL 2.0, compensateScope as well) activity is used to explicitly trigger compensation in a process. Recall that the engine is not allowed to trigger compensation on split scopes or their immediate children. Instead, we require the engine to simply forward the compensation request to the coordinator. The engine will allow the activity to complete once the coordinator informs it that the compensation work has been carried out. Note that a compensate request from a fragment can only occur while a scope is running a fault or compensation handler.

First consider the case that the request to compensate does not refer to a scope by name. In this case, the coordinator performs default compensation on the requesting scope itself. Next, we consider a set of cases that result when the compensation request does name a scope.

Consider the case that the scope to be compensated *is not* split. In this case, its parent must be split if the request to compensate has gone through the coordinator. The coordinator sends a message to request its compensation through its parent scope's protocol. As noted in the previous section, the participant side will send the childCompensationHandlerCompleted message either if the scope is not in the completed state or once it has completed the handler.

Now consider the cases that result if the scope to be compensated *is* split:

- If this scope has not completed successfully, the coordinator will immediately return that the compensation has finished without doing any work (no-op).
- If this scope has completed successfully and also has an explicitly defined compensation handler, then the coordinator will send a request to each fragment of that scope asking it to compensate. Once all fragments of the compensation handler have completed, the compensation due to the <compensate scope=”..”> activity is considered completed.
- If this scope has completed but has no explicit handler, then the coordinator performs default compensation on that scope.

Scopes nested in loops may need to be compensated more than once. We handle this using ScopeLoopQs as was done in section 3.6.3, but where the instance identifiers are the identifiers of the corresponding loop or scope *protocol instance* created by the coordinator. For non-split scopes, the coordinator places a dummy instance identifier as it does not run a protocol. As a result, if the child scope had been nested in (one or more) loops, we would compensate it as many times as the loop(s) had run. On the other hand, if a non-split scope is nested in one or more loops, then it is up to the process engine in which that scope is running to track and compensate multiple instances of the non-split scope if the coordinator requests that the scope be explicitly compensated.

Once the compensation due to the request triggered by a ‘compensate’ activity in a fragment has been completed, the coordinator notifies the requestor. This then enables the ‘compensate’ activity to complete.

Performing compensation due to a fault

The work in this section corresponds to the label ‘compensation due to termination/rethrow’ in Figure 61. The coordinator has all the information necessary to determine the compensation order - whether that order is due to termination or to default fault handling. After all required termination, compensation, and fault handling are done, the scope with the fault handler is the one in which forward navigation will continue. We therefore perform the compensation in the order prescribed in the BPEL specification. In order to use fewer messages, we do so without performing, in the protocols themselves, the upward and downward propagation that the specification uses to describe the combination of fault, compensation, and termination behavior. What we do is as follows: stop all activities in the scope that will handle the fault, and then perform the compensation order necessary by calculating what it would have been had we, at each step between the scope that threw the fault and the one that caught it, actually rethrown the fault. As a result, all activities will be terminated and compensation will occur in the prescribed order. Therefore, our approach does not violate the required BPEL behavior.

Once a fault is thrown, the fault handler is searched for as in the previous section and one item in the scope tree is marked, call it s_m: the smallest split scope from which the fault originated. This is the same scope whose protocol would have received the fault.

Call the scope, that has a matching fault handler or the process if no fault handler is found, s_f. First, all activities nested in s_f are aborted. Any non-split scopes (whether in the scope relationship tree or not) get terminated according to the BPEL specification. That is, any compensation that needs to occur *due to termination does* take place at this time for the children of these non-split scopes. However, termination of split scopes does not trigger compensation at this time. BPEL default termination of a scope does not require a prescribed termination order for the scope’s immediate child scopes, making it safe to terminate the non-

split scopes first in this way. This is in contrast with default compensation, where both split and non-split scopes are compensated in a possibly interleaved manner by the coordinator based on the order prescribed in the DCOs.

Then, the coordinator uses the scope tree and the compensation graphs in order to perform the termination/fault handling compensation and finally hands control to the fault handler of s_f:

1) Start with the marked scope, s_m.:
   a) 'Compensation due to Termination' of activities in s_m: Perform default compensation on every *split* scope s_t nested in s_m, innermost first, if s_t :
      i) is not in the fault or compensation handler(s) of s_m,
      ii) is not in the completed state, and
      iii) has at least one immediate child scope s_c such that s_c is both in the completed state and in the compensation graph of s_t.
   b) 'Compensation due to Default Fault Handler': If s_m has no handler for this fault, perform default compensation on s_m. Continue to step 2.
   c) If s_m has a fault handler for this fault, run the fault handler and skip steps 2 and 3.
2) 'Compensation due to Rethrow' of the fault: For each scope node, s_p, along the path of scope nodes joined by *cp-edges* in the scope relationship tree from s_m to the fault handling scope, excluding the start and end node, perform the following. Notice that this is simply going up the tree through the ancestors of s_m:
   a) If s_p's 'in_fault_handler' Boolean is true, skip it. Recall that since we are in a fault handler, then those scopes have already been touched before the fault handler was started.
   b) Otherwise, perform exactly the same behavior for s_p as was done for s_m in step 1 with one change: skip scopes already touched in a previous iteration of this step or step 1a.
   c) Continue to step 3.
3) 'Compensation due to Termination' of activities in the fault handling scope, s_f:
   a) Perform exactly the same behavior for s_f as was done for s_m in step 1a with one change: skip traversing down subtrees already touched in steps 1 or 2.
   b) If s_f does have a fault handler (i.e. a matching fault handler has been found in the process), run the appropriate fault handler of s_f.

Note that since compensation handlers must not throw faults, step 2 will never go from A to B in the case that A is a scope in a compensation handler of another scope B.

If a scope needs to be compensated by name, the coordinator checks the ScopeLoopQs for that scope based on which scope the request came from. It then requests compensation of the named scope as many times as the named scope had run for the instance of the requesting scope.

Race Conditions

Unlike in the case of races between fault handling related messages where one wins over the other, any compensation related messages that create a race must be queued and handled. For example, if a fault handling fragment sends the coordinator a request to compensate a nested scope A and the coordinator sends it a request to compensate its nested non-split scope B, then the fragment will queue the coordinator's request and handle it once it goes back to the state 'fault handling'.

In case of a race condition between fault and compensation messages, fault messages win over any compensation messages.

6.8.4 Example

In this section, we illustrate the protocol using the process shown in Figure 62. It combines fault handling, compensation due to termination and compensation due to fault handling.

The process is split between two participants, P1 and P2 (left and right of the black line respectively). The resulting scope tree is shown in the figure as well. Notice the 'wait' activity in scope *B*. This will provide the case that once the 'throw' in scope *A* is activated, scopes *C, D*, and *E* would have completed but scope *B* will still be in the active state. The result is enabling a close look at compensation due to termination. Notice the 'compensate' activity in the fault handler of scope *A*. This fault handler will run when the fault is caught. It will trigger default compensation of scope *A*.

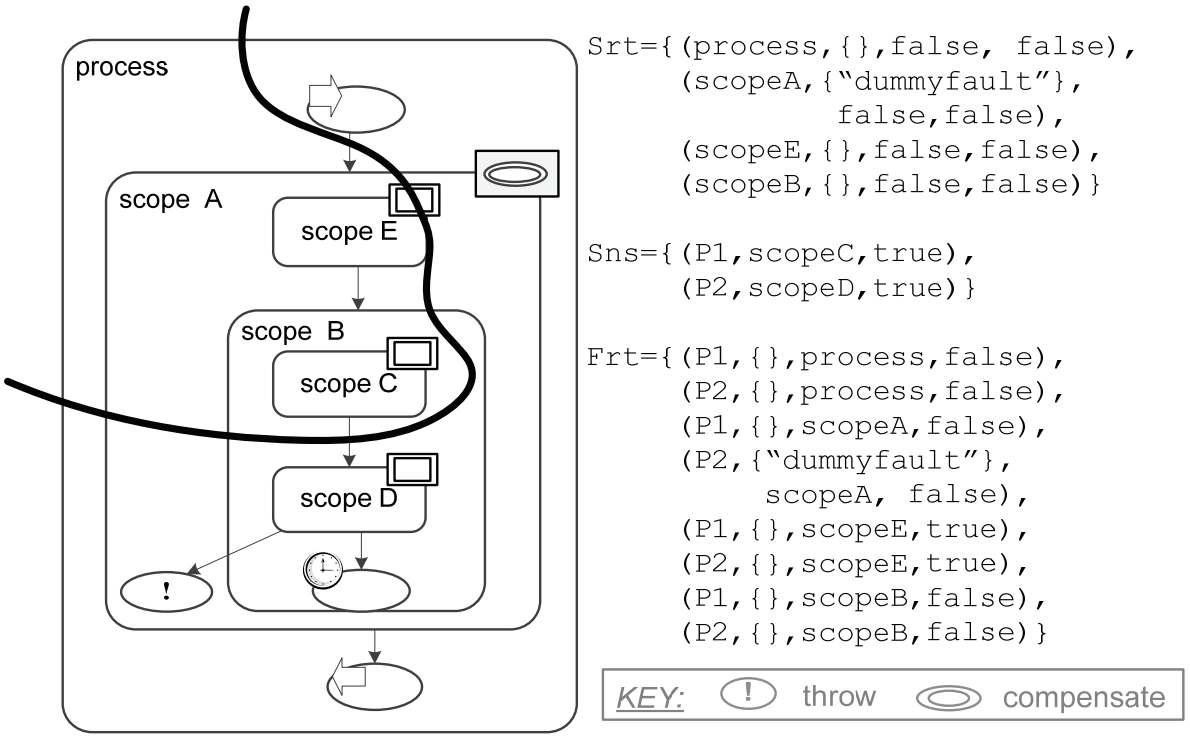


Figure 62: Sample split process (and scope tree): P1 (left of line) and P2 (right of line)

Consider the behavior of this process, before fragmentation (i.e. if it was not split): Once the throw activity in scope *A* activates, it throws a fault. The fault is caught by scope *A*. Scope *A* will then terminate its activities. The termination of scope *A* consists of terminating scope *B* because it is the only active child. Thus, the default termination behavior of scope *B* will occur: abort the wait activity and perform default compensation on scope *B*'s completed children of scope *B*. As a result, first scope *D* then scope *C* will be compensated. This ends the termination of scope *B* and thus of scope *A*. Next, the fault handler of *A* is ready to run and it reaches the 'compensate' activity. This will cause the completed immediate children of scope A to be compensated. The only such child is scope *E*, so scope *E*'s compensation handler runs. This completes the 'compensate' activity, which in turn completes the fault handler of scope *A*, then scope *A*, then the entire process.

How the scope protocol provides the same behavior when the process is split between P1 and P2 is shown in Table 16. It provides the protocol messages exchanged starting from when the throw activity is reached in P2.

Sender	*Recipient(s)*	*Protocol of*	*Message*
P2	Coordinator	scopeA	faultedWHandler(dummyfault)
Coordinator	P1	scopeA	Fault
Coordinator	P2	scopeA	compensateNonSplitChild (scopeD)
P2	Coord	scopeA	childCompensationHandlerCompleted(scopeD)
Coordinator	P1	scopeA	compensateNonSplitChild (scopeC)
P1	Coordinator	scopeA	childCompensationHandlerCompleted(scopeC)
Coordinator	P2	scopeA	runHandler
P2	Coordinator	scopeA	defaultCompensation
Coord	P1, P2	scope	Compensate
P1,P2	Coordinator	scope	childHandlerCompleting(E)
Coordinator	P1,P2	scope	finalizeScope
Coordinator	P2	scopeA	defaultCompensationCompleted
P2	Coordinator	scopeA	handlerCompleted
Coordinator	P2, P1	scopeA	completeAfterFault
P2	Coordinator	Process	Completing
P1	Coordinator	Process	Completing
P2,P1	Coordinator	Process	Complete

Table 16: Protocol messages for the example, in order from top to bottom

6.9 Discussion of Alternatives

Several alternatives, aside from replacing one's BPEL infrastructure, were considered for split loops and scopes. One was to use links to one fragment, making it in charge of synchronizing the boundaries of a fragmented construct. This would work for loops fragmented with few participants and dependencies. One may consider a hybrid approach, where this is done for splitting simple loops based on a cost function. If the cost exceeds a threshold, one can then use a coordinator. However, it is intractable to use it for faults and compensation. Another alternative is to use a new BPEL process that acts as a lightweight coordinator and mediates all the interactions. However, again it was thought that a first class treatment of the coordination is the cleanest approach while still promoting reusability and transparency.

6.10 Conclusion

In this chapter, we have shown a mechanism for splitting the implicit control dependencies imposed by loops and scopes, in particular in the presence of fault, explicit compensation, and default compensation handling. This is enabled at the language level by adding just three new attributes to BPEL and at the runtime level by using coordination protocols. The complexity involved in BPEL default compensation, interleaved with termination and fault handling, as well as in the presence of loops, was detailed and supported with the new protocols.

Chapter 7 Architecture and Implementation

This chapter presents the architecture and implementation of the system created to support process fragmentation. The fragmentation of BPEL/BPEL-D processes is done using a build time module and a runtime module that supports the coordination protocols. The build time module consists of a graphical editor for designing process that is extended to allow the designer to graphically assign different activities to different participants. The editor also contains a transformation module that creates the resulting process fragments. If the input is standard BPEL, the editor uses a data analysis module. The runtime module consists of a BPEL engine and a coordinator, extended to support the split loop and scope coordination protocols. The mapping between the steps of our approach presented in Figure 1 and these modules is illustrated in Figure 63.

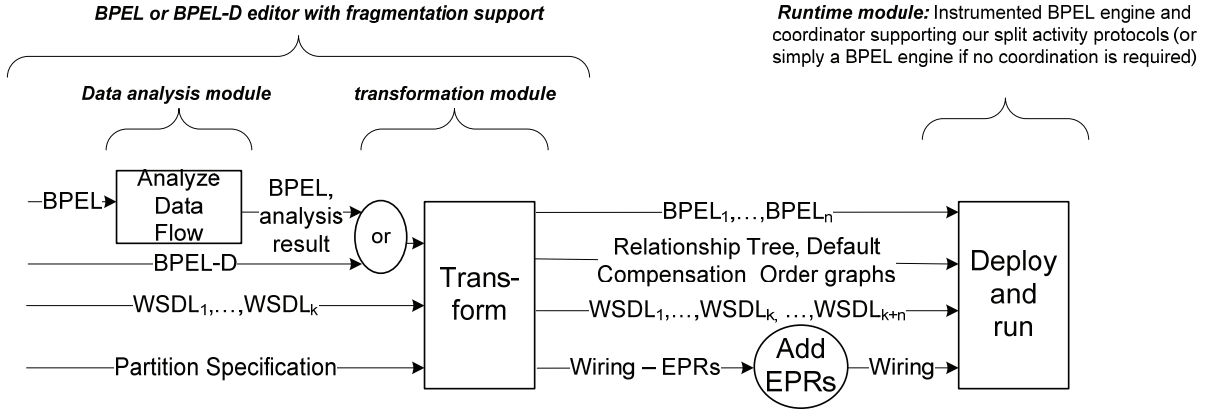


Figure 63: Steps of the approach mapped to implementation artifacts

7.1 Creating the Partition

The approach requires a user to define a partition by assigning different activities to different participants. One option for doing so is for the user to create an XML representation using XPath to associate participant names with the activities in the process. This would be similar to how policies are attached to BPEL activities in [CHKM07]. Manipulating XML artifacts is cumbersome and can only be done by IT specialists. Therefore, we provide a graphical editor that allows one to define partitions in a user-friendly manner. Using the editor, the user is able to create named participants, and assign to each participant a set of activities from the process. The editor provides the capability to split a process. If a user requests this, the editor interacts with the transformation module, and in the case of standard BPEL also the data analysis module, to produce the necessary artifacts needed as input for the runtime module. The implementation of the data analysis and transformation modules is based on the algorithms in this thesis.

If one wishes to separate the editor and the transformation module, to make the latter reusable by multiple editors or for manually created input, then a combined approach is advised: provide the user with the graphical interface and produce as a result an XML representation of the partition. This can then be passed to the transformation module regardless of how it is represented internally in an editor.

In order to extend a BPEL editor to support support fragmentation of BPEL(-D), several steps

have to be taken as shown in Figure 64. First, the editor must be extended to allow graphically assigning activities to participants and to finally provide the transformation module that supports the fragmentation algorithms. For the BPEL-D case, the editor must be extended to support BPEL-D syntax (e.g. data links and containers). For the BPEL case, data analysis is provided. Section 7.5 details the realization of this support in the created prototype, as an extension to the Eclipse BPEL editor [ECLI07A].

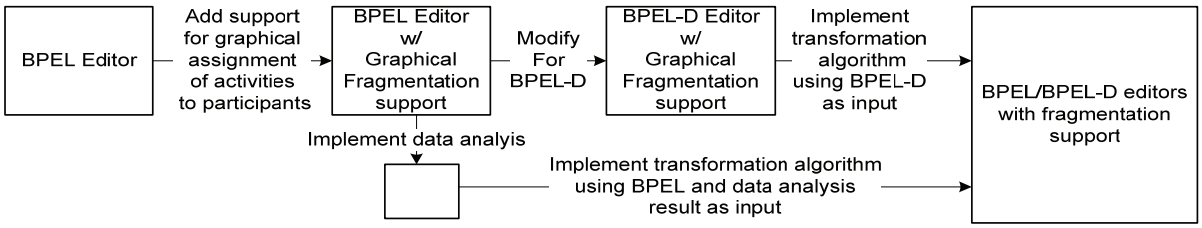


Figure 64: Creating a fragmentation-enabled BPEL/BPEL-D editor from an existing BPEL editor

7.2 The Runtime Module

BPEL processes that have been fragmented such that coordination is not required (for example, no split loops) can be run just with a standard-compliant BPEL engine. It is for the processes that do require coordination where we introduce a new runtime module.

The motivation to avoid introducing new middleware where possible leads to starting with existing BPEL engines, and providing a mechanism by which any BPEL engine can be instrumented to control its navigation such that it can support the participant side of our coordination protocols. The other aspect of the runtime module is to support the coordination protocols themselves: the participant side interacting with the engine and the coordinator side plugged into a coordination framework.

7.3 Instrumenting a BPEL Engine

One can see from the coordination protocols that the participant side needs to able to not only send out notifications of different lifecycle events of a process (e.g. that a scope is starting), but also to react to incoming events from the coordinator (e.g. fault thrown in another fragment) or block and wait on a path until it is instructed what to do (e.g. being notified that all other fragments of the same scope are ready so it may proceed).

The first step is to provide a mechanism for the engine to know which processes need to participate in this special behavior, and more specifically, which constructs of these processes. This has been enabled in this thesis using the BPEL language extensions, in section 5.6, that identify whether loops and scopes are split: *belongs-to*, *is-fragmented* and *is-responsible*. The next step is to modify the engine so it may provide the proper behavior for fragments. In providing support for behavioral extension of BPEL, we created an approach that adds the extension behavior in a modular, minimally obtrusive and maximally reusable manner.

In designing the instrumentation of the engine, we looked to common patterns found in the large number of projects that extend BPEL behavior, either using the XML extensibility of the language itself or otherwise at runtime. The extensions we are interested in are those that complement the language's constructs without changing their existing fundamental behavioral semantics.

Examples of projects that extend BPEL include adding BPEL-SPE's subprocesses [IBSA05], BPELJ's Java capabilities [BGKL04], BPEL4Chor's choreography connections [DKLW07], parameterization in [KLNW06] and others. Tai et. al [TAKM04] add transactional capabilities via policy attachment; Charfi and Mezini [CHME04] enable Aspect Oriented Programming for BPEL; Mandell and McIlraith [MAMC03] plug in (chains of) services that can fulfill the need of the process using semantic information. BPEL 1.1 itself described how the WS-BA coordination protocol can be used to enact the relationship between parent and child BPEL scopes. It has been found that the usual course of action has been to go into the internals of different engines and add mechanisms to trigger and react to external events in a way that was not geared to generalization and reuse.

Our approach, shown in Figure 65, uses an event-based mechanism with pluggable 'controllers' and an engine-agnostic generic event model documented in [KKSP06]. The BPEL engine is modified to support interacting with and reacting to a Generic Controller that is responsible for the events [KHKL07A]. The modification includes providing the glue code that can map between the event model being used and the native navigation model and capabilities of the engine. For extensions of a particular application domain, in our case split activity coordination, a Custom Controller is provided that plugs into the Generic Controller and interacts with it via the generic events. A Custom Controller filters the events to those that are relevant for its specific application domain, and provides the behavior needed by that particular domain (such as creating, sending, and receiving WS-Coordination messages). The system provides a custom controller subscription mechanism. A Custom Controller optionally interacts with external subscribers involved in implementing the extension logic, in our case the coordinator is such a subscriber.

The engine and a controller need to be able to send and react to events from each other. We categorize these into three types: (1)'Notification Events' are sent from the engine, causing the custom controller to react by performing a custom action (such as send a WS-Coordination message); (2)'Incoming Events' are sent from the controller into the engine, enabling it to react to external events while processing (such as receiving a fault from another fragment); (3) 'Blocking with Callback' events, or BC events enable the engine to be sent an event and block the path of the process that produced this event. A particular Incoming Event is related with each BC event, unblocking the path and determining how the engine must continue on the path. To avoid blocking unnecessarily, BC events may only be used if there is a registered custom controller for them. That controller must be able to produce the corresponding unblocking Incoming Event.

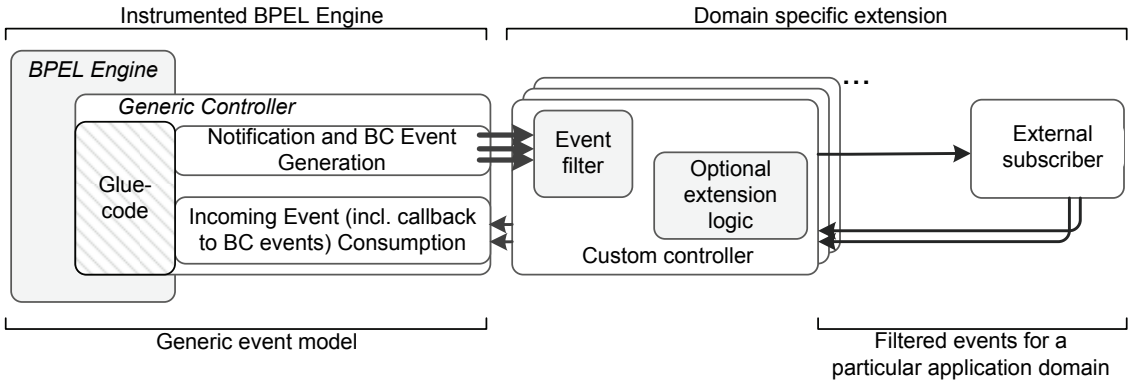


Figure 65: The architecture for instrumenting a BPEL engine [KHKL07A]

Each extension is associated with the set of its needed events. In our case, the BPEL language extensions are mapped to the events for loops and scopes, in the generic event model, that are described in section 7.3.1. A custom controller is created for handling the participant side of the split activity coordination protocols in Chapter 6. This controller is then registered with these scope and loop events. Since all BPEL engines have their own (distinct) internal navigation models, the glue-code maps between the internal navigation and the generic event model.

In addition to the prototype presented in this thesis, two other projects make use of this BPEL extension architecture, showing that it is in fact applicable for multiple BPEL extensions and not just for splitting: the pluggable transaction models in BPEL scopes: [MIET06] and the addition of aspect-oriented features to BPEL [SCHR06].

7.3.1 Extensions and Events for Split Processes

A process with scopes or loops having activities decorated with the split activity BPEL extensions triggers the events for loops and scopes in the common event model. A custom controller for split loops and scopes, detailed in the next section, is registered with the instrumented engine for loop and scope events only from the split activities.

In what follows, if there is a choice between several events being published on a transition from the same state and under the same conditions then the decision of which one gets published depends on extension registration and controller subscriptions. This will usually be a case of choosing between a blocking path and a non-blocking one as can be seen in the dead-path links out of the ‘Inactive’ state in Figure 66. Also, when it is stated that a BC event is published, then it implies that that occurs only if there is a registered controller that can provide the corresponding unblocking event.

Loop Events

The loop event model, shown in Figure 66, is presented in this section. A loop starts like any activity in the ‘Inactive’ state. Once all of its incoming links have been evaluated, its join condition is evaluated. If the condition is false, then either a notification ‘Activity_Dead_Path’ event or a BC ‘Activity_Dead_Path_Blocking’ event is thrown. In the latter case, it waits for the unblocking incoming event ‘Complete_With_Fault’ and in both cases enters the ‘Dead-Path’ state. If the join condition is true, the ‘Activity_Ready’ event is published. If there is a controller registered for the ‘Activity_Ready_Blocking’ BC Event, then that event is published instead and the path is blocked until the ‘Start_Activity’ incoming event arrives. Once the activity starts, the ‘Activity_Executing’ notification event is published and the loop condition is ready to be checked.

After that, one of the ‘Loop_Condition_True’ or the ‘Loop_Condition_False’ BC Events is published. The controller then decides whether the execution may continue by sending the incoming event ‘Continue_Loop_Execution’ or ‘Finish_Loop_Execution’. In the latter case, the activity enters the end state, while in the former case, the body of the loop will be executed and the nested activities enter the state ‘Executing’.

Once a loop iteration completes, the ‘Loop_Iteration_Complete’ BC Event is sent to any registered controllers. The loop iteration gets unblocked by receiving either the ‘Continue_Loop’ event, at which point the loop condition is evaluated again and we continue as before from ’Executing’.

If no controller is registered for the 'Loop_Iteration_Complete' BC event or the 'Loop_Condition_True/False' BC event, then the engine performs the loop's work in the 'Executing' state and the loop will continue or complete based on its local condition evaluation.

A loop only completes from the 'Check Condition' state. The other paths to completion exist simply because the event models of all activities was designed to extend a base activity event model (here, integrated into the loop model).

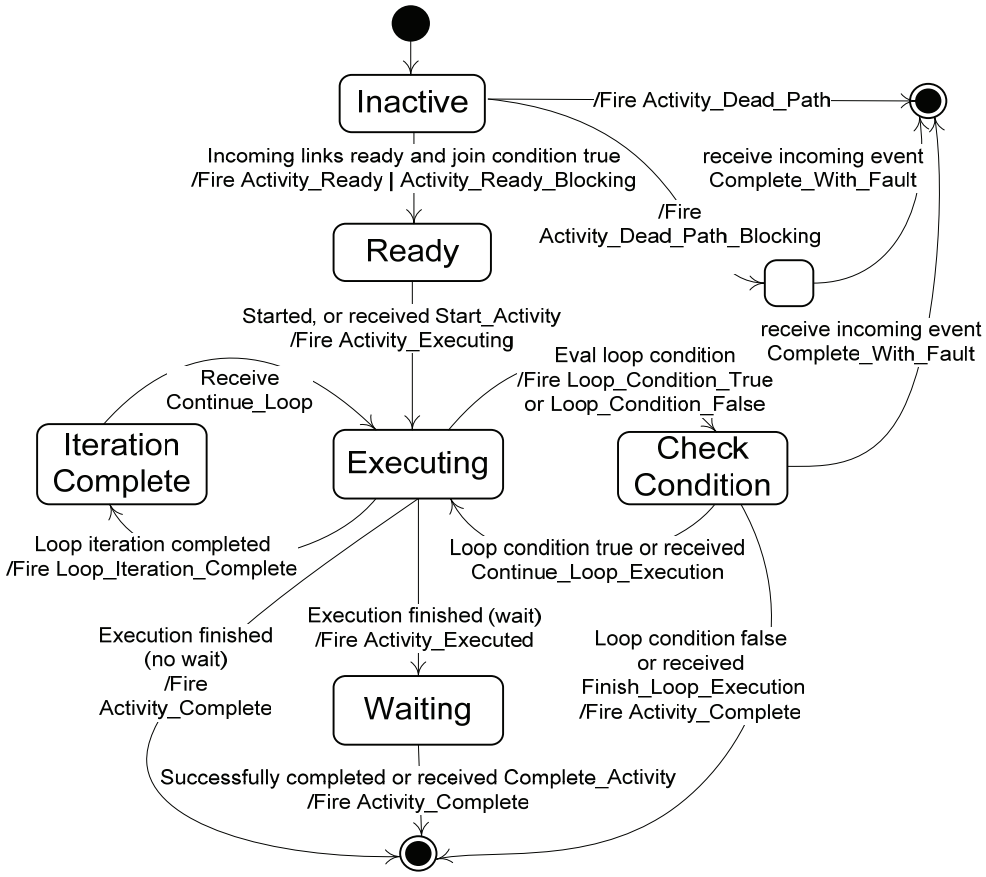


Figure 66: Event model for loops, based on [KKSP06] [KHKL07A]

The 'fragmented' and 'is-responsible' extension attributes are registered for the following loop notification and BC lifecycle events as well as all incoming events:

- Activity_Dead_Path_Blocking
- Loop_Condition_True
- Loop_Condition_False

A custom controller for split loops subscribes to these notification and BC lifecycle events and provides the corresponding incoming events.

Scope Events

Scopes again start in the 'Initial' state, and have the same dead-path elimination arcs and path to the 'Executing' state. The protocol is more complex due to the fault and compensation handlers. The event model for states is shown in Figure 67, which is a slightly modified version of [KKSP06] with the event related to BPEL event handlers omitted to reduce paths not relevant to this thesis.

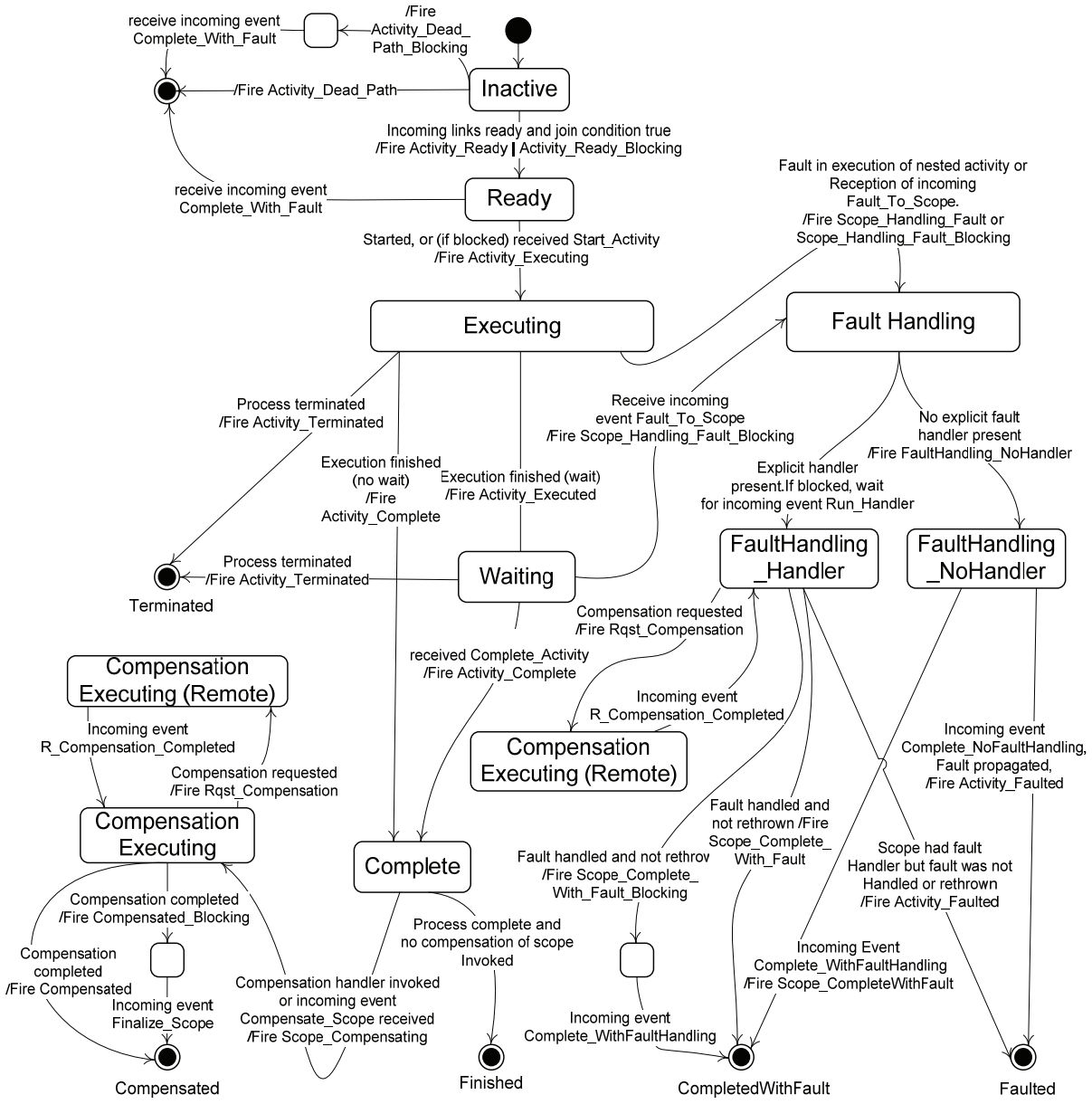


Figure 67: Event model for scopes, based on [KKSP06]. Event handler-related events not shown

If the scope runs normally without faults and no one is registered for the BC 'Activity_Executed' event, then it simply performs its work and upon completion fires the notification 'Activity_Complete' event leading it to the 'Completed' state. If a controller had registered for 'Activity_Executed', then the lifecycle would go to the 'Waiting' state instead and wait for the incoming event 'Complete_Activity' to arrive at 'Completed'. If the scope faults due to a nested activity, then it goes to the fault handling state. Additionally, a provision is made to enable controllers to inject faults: the incoming event 'Fault_To_Scope' is another way the scope lifecycle can reach the 'Fault_Handling' state.

Focusing on possible paths from the 'Completed' state, one can see that either nothing happens and the lifecycle ends when the process instance completes, or the scope is asked to compensate. While compensating, the scope may call compensation handlers of its nested scopes. These may be in other fragments, hence the path through the 'Compensation Execution (Remote)' state. Once the compensation handler is complete, then again there are

two options: blocking and non-blocking completion notification. The former requires an incoming 'Finalize_Scope' event to arrive, while the latter directly takes the lifecycle to the 'Compensated' end state. We currently do not handle faults from a compensation handler.

Next, consider the fault handling section on the bottom right of the figure. If there is no fault handler, but there is subscriber to the BC event that signaled the fault, then the process stops all nested activities and waits until it is allowed to navigate from the scope. Either the fault has been handled externally (here, by another fragment) and the incoming event 'Scope_Complete_With_Fault' occurs leading to the 'CompletedWithFault' end state, or it wasn't and the incoming event 'Complete_NoFaultHandling' occurs leading to the 'Faulted' end state, disabling the scope (and hence its outgoing links as well).
On the other hand, if there is a fault handler, then it waits for the 'Run_Handler' event if it was blocked or otherwise goes directly to 'FaultHandling_Handler'. While fault handling one can again get compensation requests, hence the duplicate 'Compensation_Executing (Remote)' state as from the 'Compensation Executing' state. Once the fault handler completes, it signals by publishing either 'Scope_Complete_With_Fault' or 'Scope_Complete_With_Fault_Blocking', where the latter needs the unblocking incoming event 'Complete_WithFaultHandling'. Either way, the lifecycle ends at the 'CompletedWithFault' end state which will result in outgoing links from the scope being navigated normally.

The 'fragmented' extension attributes are registered for the following scope lifecycle notification and BC events, as well as all incoming events:

- Activity_Ready_Blocking
- Activity_Dead_Path_Blocking
- Activity_Executed
- Activity_Terminated
- Scope_Handling_Fault_Blocking
- Scope_Complete_With_Fault_Blocking
- Rqst_Compensation
- Compensated_Blocking.

A custom controller for split scopes subscribes to these notification and BC events and provides the corresponding incoming events.

7.4 Supporting the Coordination Protocols

The support for instrumenting the engine provides core infrastructure upon which the coordination protocols at hand are implemented. First, a high-level view is provided, and then a deeper look is taken into the components.

The high-level view, shown in Figure 68, consists of a custom controller that provides the event filtering and the logic needed for the split loop and scope protocols (a.k.a. the split activity protocols). The controller is plugged into an instrumented BPEL engine (see Figure 65) to create a 'fragmentation-enabled' BPEL engine. Fragmentation-enabled here means fragmentation that requires coordination. The controller provides the participant side of the protocols and interacts with the engine to control its behavior accordingly. Each such engine running a fragment of a process (that needs to be coordinated) interacts with a coordinator using SOAP messages in accordance with the split activity protocol WSDLs. The coordinator is a system implementing the WS-Coordination specification, which itself provides support

for plugging in new protocols. One could use an existing WS-Coordination implementation, and add to it support for the coordinator side of the split activity protocols. Figure 68 only shows WS-Coordination's Registration Service and not its Activation Service as the latter is not used in this thesis. Currently, fragments of the same process instance must interact with the same coordinator.

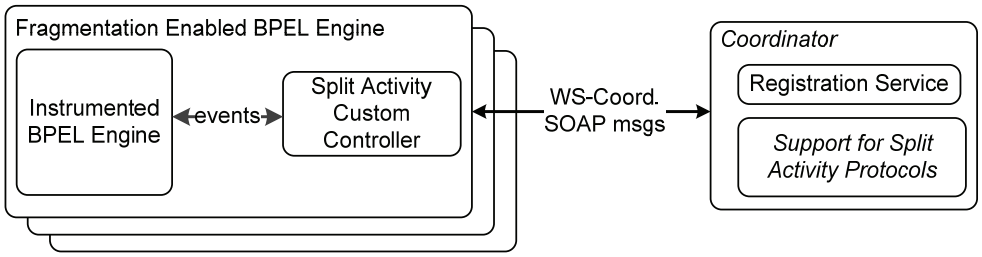


Figure 68: BPEL engines and a coordinator interact to enact fragmented loops and scopes

The custom controller and the coordinator are explored further in Figure 69. They correspond to the 'domain specific extension' of the architecture of an instrumented BPEL engine (see Figure 65), because they provide the logic specific to executing split activity fragments.

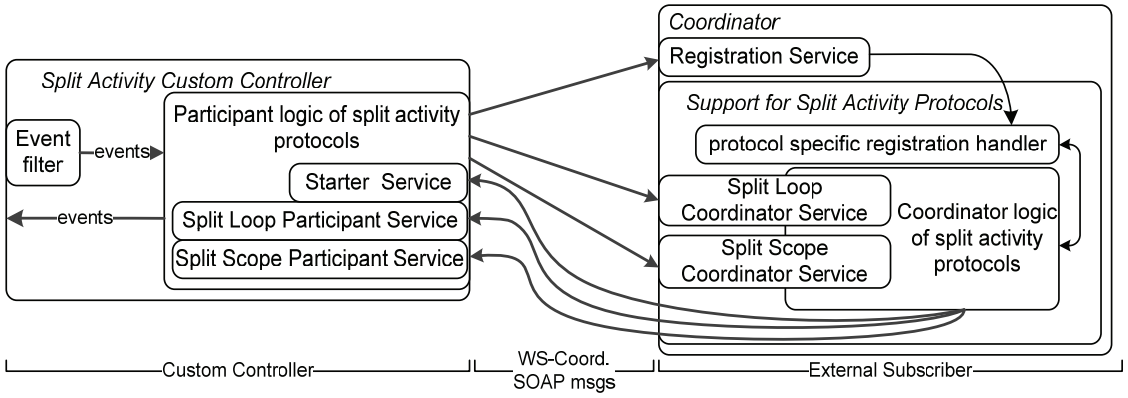


Figure 69: Details of the 'domain specific extension' (see Figure 65) for split loops and scopes

Recall that the coordination of split processes requires three fragment-side services: a starter service to ensure that all fragment processes are ready and can receive incoming protocol messages before any activities execute, a loop participant service implementing the participant side of the loop protocol, and a scope participant service doing the same for the scope protocol.

The Split Activity Custom Controller interacts with the instrumented engine (specifically the Generic Controller) using the events, as described in the previous section. Like all custom controllers, it contains two main parts: an event filter and the domain specific logic. The domain specific logic in this case is (1) the implementation of the participant services, the starter service and the outgoing invocations needed by the split activity protocols and (2) the logic that determines when to create which coordination message (usually in reaction to an engine event), and when to send which events into the engine (usually in reaction to a coordination message). That also includes any adaptation necessary between the event and coordination message formats (usually SOAP) and contents. The expected behavior has to follow the description of the participant side of the protocols as detailed in Chapter 6.

In section 7.3, it was shown that an external subscriber could be provided to perform any additional logic needed by an extension. In our case, this subscriber is in fact the coordinator.

It provides (1) a registration handler that performs protocol specific work triggered by registration. This includes pulling extensions out of the registration message and using them to create a coordinator side instance identifier, seeing whether an instance of the activity already exists, creating the reference properties needed for the coordinator address sent back in the registration response message; (2) the implementation of the coordinator side services and outgoing invocations needed by the split activity protocols; (3) the logic of reacting to the messages and creating new ones. This includes searching for fault handlers, grouping fragments of the same instance together based on keys generated using input from the registration handler, calculating compensation order, and other behavior detailed in the protocol descriptions in Chapter 6.
The registration request needs a protocol identifier and also needs to send the registration extensions (ProcessName, LoopName, ScopeName, FragmentName, CorrelationSet, and Counter). The protocol identifiers are:

- `http://www.iaas.uni-stuttgart.de/wscoor/protocolID/ScopeProtocol`
- `http://www.iaas.uni-stuttgart.de/wscoor/protocolID/LoopProtocol`

In section 6.4, we stated that the manner in which an implementation creates the participant and coordinator addresses is implementation dependent. How this was done for our prototype is shown in section 7.5. The identifiers in these addresses can be machine generated unique identifiers that do not follow a particular pattern such as including known items (like scope name, etc.). The reason is that the registration extensions contain the common information needed by coordinator and participant to create these addresses.

7.4.1 Fragment Deployment

Upon knowing where the processes are to be deployed and where the coordinator is provided (if one is needed), the necessary endpoint references are provided in the wiring description (iaas:splitProcessDeployment). Deployment descriptors in BPEL are engine specific. Therefore, one needs to generate the deployment descriptors from the wiring definition depending on which engine is chosen. At that point, each fragment is deployed to the target engine in such a way that all partnerLinks are bound to endpoint references. If a coordinator is provided, then the deployment step also binds the coordinator endpoint with the controllers at each endpoint.

The process is then registered with the coordinator. Before an instance starts, the coordinator needs the scope relationship tree, default compensation order graphs, and starter service addresses. When these are provided is not critical as long as they are provided before any instance of the split process runs.

Sending the coordinator only the tree and the DCO graphs provides better separation of concerns because the full process model may contain information that must not be sent out of the organization that owns the process model. It is also more efficient to create them at design time, instead of at runtime, so they can be reused across deployments.

On the Use of Partner Endpoint References

In this thesis, we have taken the approach that endpoint references of fragments and external partners are provided at deployment time in the ‘splitProcessDeployment’ element. In reality, partners may use the ‘replyTo’ headers of WS-Addressing to point their clients to new locations. Additionally, the process itself may use an ‘assign’ to copy an EPR from a message into the partnerLink. The promise of SOA often mentions about focusing on interfaces and enabling late and possibly dynamic binding to physical endpoints. These cases are not

supported at this time as they are on the periphery of the focus of our work. They can in fact be handled by extending the approach without loss of generality, using the same building blocks and mechanisms presented in this thesis.

The case of updating an EPR using an assign activity can be handled by treating the EPR like a variable and then creating sending/receiving blocks to propagate it. In this case, even BPEL-D processes would need a (very basic) form of data analysis and again one would restrict to processes where EPR read/write complies with the Bernstein Criterion.

The case where the EPR is modified using lower level protocols (such as WS-Addressing) is more complex as it occurs deep in the platform middleware. Additionally, the priority for conflicts between such updates and updates in the process definition itself is still under-specified in BPEL: the specification intentionally stays out of implementation specific issues and avoids depending on a singular addressing specification. These are some of the reasons that an in-depth solution for this is out of scope for this thesis. However, one can see a solution along the following lines: extend the eventing approach of the implementation to notify a new controller of such changes, and then propagate this EPR to the middleware of the other fragments. The controller must be able to accept and perform a request to update an EPR. The propagation can possibly use WS-Notification to send the EPR.

While we have shown that the common obvious cases are tractable in our framework, an in-depth treatment of the problems of EPR propagation is actually a non-trivial area for future research. Solution for dynamic EPR propagation and other updates to shared context have so far been focused on the two-node, client-server paradigm, and this would need to go beyond that to handling it in a system consisting of multiple client-service entities (such as BPEL processes that act both as client and service). One suggestion for a framework in which this can be investigated is the Service Component Architecture (SCA) [BCII06]. SCA provides specifications for describing, connecting, deploying, and running applications consisting of several services or client-service entities across multiple machines.

7.4.2 Participant Logic

The loop and scope protocol services handle the incoming protocol messages, extract the reference properties to determine which process instance and which loop or scope in that process is targeted by the message, and most times result in creating an incoming event for the engine. The rest of the participant logic module is responsible also for accepting the events from the engine, determining which coordinator service address it corresponds to, and most times create an outgoing call to the WSDL operation of that service using a message from the loop or scope protocol.

There will be one WS-Addressing EPR for each instance of each split activity of a process. This means one may think of them as different services. However, their implementations are closely coupled. We therefore use the same URLs in the EPRs and differentiate them only using reference properties in the EPR. Then, all messages for the same process would arrive to the same implementation code that identifies the reference properties and sends the message with the information of which process, process instance, and activity to a common module that oversees the behavior for the entire split process. There are two ways one may do this: (1) stateless method calls with information about protocol state in database tables, or (2) maintaining the state in the module itself. The former is recommended due to the long running nature of business processes.

There is a causal relationship between the events and the protocol messages, described in the next subsections. The behaviors of the participant and protocol side were provided in Chapter 6 with respect to the protocol and process state. Here, we relate that to the actual events in the instrumented BPEL engine.

Loop Protocol and Engine Events

Table 17 summarizes event-message relationship for loops. Notice that the controller may send incoming events that cause a loop fragment to finish even if its own fragment's loop condition was true, and it can cause it to dead-path even if it locally did not get a join failure.

Loop Event	*Protocol Message*	*Notes*
Activity_Dead_Path_Blocking	joinFailure	
Loop_Condition_True	starting, completing	The responsible fragment's message includes its value of the loop condition.
Loop_Condition_False	starting, completing	
Continue_Loop_Execution	start, continueLoop	Unblock Loop_Condition_True/False
Finish_Loop_Execution	complete	
Complete_With_Fault	completeAfterFault	Unblocks Activity_Dead_Path_Blocking or Loop_Condition_True/False

Table 17: Causal relationship between loop events and loop protocol messages

Consider how the events are used to drive a split loop: The controller sends the coordinator a message when it gets to any of the 'Loop_Condition_True' or 'Loop_Condition_False' or 'Activity_Dead_Path_Blocking' events. This means that the particular fragment of the loop has been reached in the navigation in that process. The other fragments in the loop are in other processes and their controllers also send a message to the coordinator once the loop is reached in the navigation of their processes. The coordinator now has the information it needs to notify the controllers of all the fragments of whether the fragments should go ahead in executing or not. The coordinator sends a message to the controller that the latter maps to the appropriate unblocking event: 'Continue_Loop_Execution', 'Finish_Loop_Execution', or 'Complete_With_Fault'. If the loop should iterate, the controller waits for the next 'Loop_Condition_True' or Loop_Condition_False' event and sends a corresponding message to the coordinator to signal the end of the iteration. The coordinator waits until all fragments have reached the end of the iteration. It then sends each controller the appropriate unblocking message. The fragments will either iterate or end together as a result.

Scope Protocol and Engine Events

The case for scopes is less direct. One reason is that the scope lifecycle does not contain the ability to accept incoming events to compensate its children while it is fault handling even though the protocol does do so. Recall that the coordination protocol is between the coordinator and one fragment of a split activity and that nested non-split scopes that may need to be compensated do not run a protocol. Instead, the protocol messages regarding compensating non-split scopes are sent and received in the protocol of an ancestor split scope. Upon receiving such a compensation message that names a non-split scope from the coordinator, the controller finds the named child scope and sends an event to that particular child scope in the engine - not to the ancestor scope to whose protocol instance the coordinator sent the message.

Scope Event	*Protocol Message*	*Notes*
Activity_Ready_Blocking	starting	
Activity_Dead_Path_Blocking	joinFailure	
Activity_Executed	completing	
Compensate_Scope	compensate, compensateChild, compensateNonSplitChild, defaultCompensation	The protocol message would have come to an ancestor scope of the one that will receive the actual request to compensate
Compensated_Blocking	compensationHandlerCompleted, childCompensationHandler-Completed	
Rqst_Compensation	compensateChild, defaultCompensation	
Scope_Handling_Fault_ Blocking	faultedWHandler, faulted	
Activity_Faulted	faultInHandler	Only if the protocol is in state fault handling (that is, scope lifecycle is in state FaultHandling_Handler)
Scope_Complete_With_Fault_ Blocking	handlerCompleted	
Start_Activity	start	Unblocks Activity_Ready_Blocking
Complete_Activity	complete	
Complete_With_Fault	completeAfterFault	Unblocks Activity_Dead_Path_Blocking or Activity_Ready_Blocking
Finalize_Scope	finalizeScope	Unblocks Compensated_Blocking
Compensation_Completed	compensationHandlerCompleted, defaultCompensationCompleted	Unblocks Rqst_Compensation
Run_Handler	runHandler	Unblocks Scope_Handling_Fault_ Blocking
Fault_To_Scope	faultWHandler, fault, faultAndExit	
Complete_WithFaultHandling	completeAfterFault	Unblocks Scope_Handling_Fault_ Blocking or Scope_Complete_With_Fault_ Blocking: The former is when there is no handler here but another fragment has handled the fault; the latter is when there is a handler here waiting for other fragments to complete

Table 18: Causal relationship between scope events and scope protocol messages

Table 18 shows the scope events' relationship with protocol messages. The rows in the table first address outgoing events (Notification and BC), then go over Incoming events. For both the former and the latter, the table starts with the normal lifecycle case, then moves on to the compensation paths from the 'completed' state of the protocol, and then over the events for fault handling (leading from and then out of the 'fault handling' state).

Consider how the events are used to drive a split scope: When a scope starts, the engine sends 'Activity_Started_Blocking' or 'Activity_Dead_Path_Blocking'. This causes the controller to register the scope and then send the corresponding protocol message. Once the coordinator has received such a message from all fragments, it notifies the controller using the protocol, and the controller in turn unblocks the scope by sending in either the 'Start_Activity' or the 'Complete_With_Fault' event. After a fragment completes its own work, it sends the blocking event 'Activity_Executed' to the controller and waits to hear back when it can complete its scope. The controller sends 'completing' to the coordinator and waits to hear whether it can in fact complete or whether there was a fault in another fragment that should be handled. The coordinator waits to hear from all fragments about their completion status and notifies the fragments of the result. Based on what message comes back from the coordinator, the controller sends in the appropriate event ('Fault_To_Scope' or 'Complete_Activity'). Once the coordinator is notified that all the process level fragments have completed, then it sends 'finalizeScope' to all completed scope protocol instance, causing the controller to send the 'Finalize_Scope' event to the engine.

In order to avoid repetition, we now highlight a few other interesting aspects of the events instead of walking through every path of the lifecycle. Notice also that if this scope has no handler for a fault that occurred in the fragment and no other fragment handled it, then the protocol does not cause the 'Complete_NoFaultHandling' incoming event shown from the state 'FaultHandling_NoHandler'. The reason is that the parent scope where the fault is caught (or the process itself) will get the (rethrown) fault and it will terminate its nested scopes as per usual BPEL nested scope termination upon the reception of a fault.

In our case, scope compensation by the engine itself is disabled for scopes in the scope relationship tree, so there will not be a case where the scope will fire 'Scope_Compensating' without having received an incoming 'CompensateScope' event.

7.4.3 Coordinator Support

The module implementing the coordinator logic uses the scope relationship tree to initialize a store of the information needed to run the protocols for each split process. It is preferable that this store is in persistent database tables due to the long running nature of business processes. Upon receiving a registration message, the coordinator checks whether it already has a previous fragment registered for that same activity. If not, then it creates a new record for that instance of the split activity in which it places four items: (1) the state related to the message, (2) the participant unique identifier (address), (3) new reference properties to create the unique 'coordinator instance identifier' address for that fragment of the split activity's coordinator service, and (4) the relationship tree to determine how many more fragments it must wait for before it can send them the 'start' or 'completeAfterFault' message. Otherwise, it adds the fragment to the table for that instance, and then creates and sends the address for the coordinator instance using the same reference properties as for other fragments of this split activity. As the messages for the protocol of that activity instance are exchanged, the tables get updated accordingly.

The relationship tree and the default compensation order graphs, in combination with the store of fragment instance state and the state of the split activities of a process instance, needed to provide the necessary coordination behavior were described in Chapter 6.

While the service EPRs are different for each fragment of each instance of a split activity, the coordinator logic in fact uses information about the overall process state, stored in the tables, to determine how to react to the incoming messages and what to send back in response.

7.5 Prototype

An implementation of this architecture has been created, consisting of:

- An editor
- An instrumented BPEL engine
- An extended WS-Coordination middleware
- An implementation of the coordination protocols

The prototype enabled testing the overall approach as well as edge cases and race conditions in the protocols. One particular challenge faced was that current BPEL engines and editors support a hybrid version between BPEL 1.1 and 2.0. In particular, the BPEL engine and editor we based our prototype on are not fully compatible and some mapping had to be done so that the output of the editor can be used as the input to the BPEL engine.

The build-time environment consists of an editor that supports fragmentation. The editor extends the open source Eclipse BPEL editor [ECLI07A] to support BPEL-D, provides graphical support for partitioning and implements the transformation module for splitting BPEL-D processes. The BPEL editor, being Eclipse-based, already provided an in-memory model of BPEL using the Eclipse Modeling Framework (EMF). EMF [ECLI07B] provides the ability to define a structured data model that can be used as the input to a code generation utility. This utility produces (among other things) Java classes that represent the artifacts in the model and adapter classes to aid in graphically representing these artifacts.

To support BPEL-D in Eclipse, the BPEL EMF model provided by the BPEL editor had to be extended by the corresponding new BPEL-D constructs (data links, maps, and containers). The existing models of processes and activities had to be extended to support these constructs. Modifying an existing EMF model consists of modifying the corresponding ‘ecore’ file, i.e. *bpel.ecore*. Java interfaces and implementations of the model artifacts were created by using the EMF code generation utility passing a corresponding ‘genmodel’ file as input. The BPEL editor’s existing classes for the graphical representation of the EMF model must also be modified to support the use of data links instead of variables, i.e. the rendering of variables had to be suppressed.

The extended editor, shown in Figure 70, supports partitioning a process by providing a graphical representation of participants as well as a capability to associate activities with participants, create the fragments, and deploy them. The process opened in the editor in Figure 70 has four activities connected by four control links. A circular icon on a control link depicts a transition condition. The process has two data links shown in the right hand panel (Link4 and Link5). The data links are from ‘Assign’ to ‘Reply’ and from ‘Assign1’ to ‘Reply’ (not shown in the figure). To create a partition of a process opened in the editor, a user adds some participants and associates them with activities, as shown in the left side of Figure 71. This is shown by using a different color for each participant’s activities. In the right side of Figure 71, ‘Assign1’ is blue, while ‘Receive’, ‘Assign’, and ‘Reply’ are green.

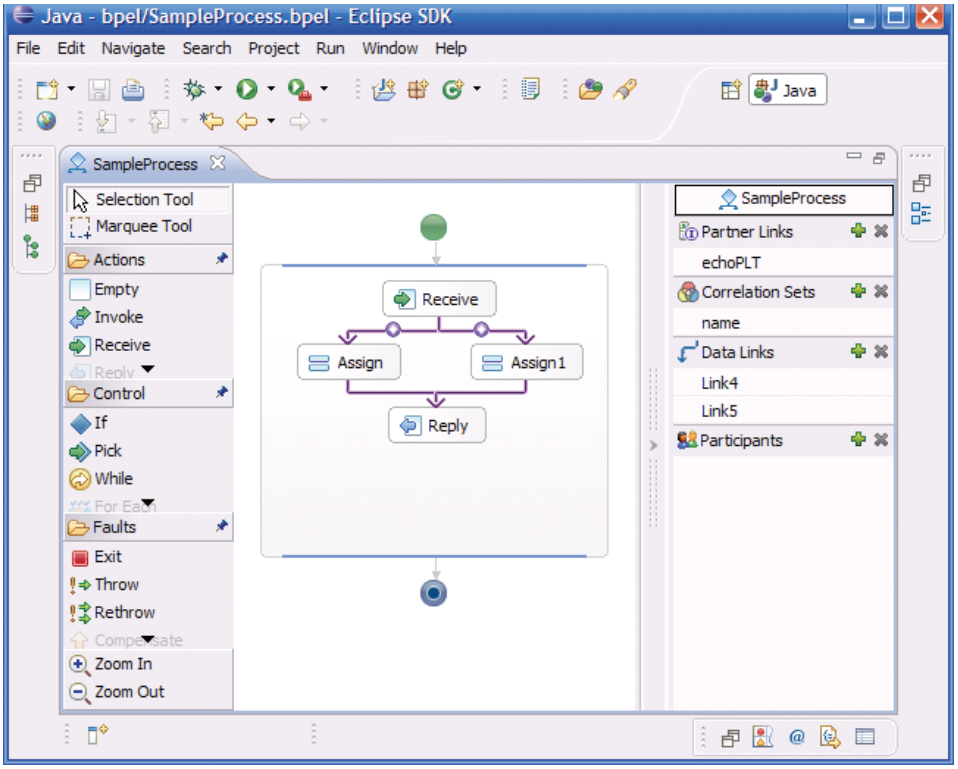


Figure 70: The editor, with a sample BPEL-D process created

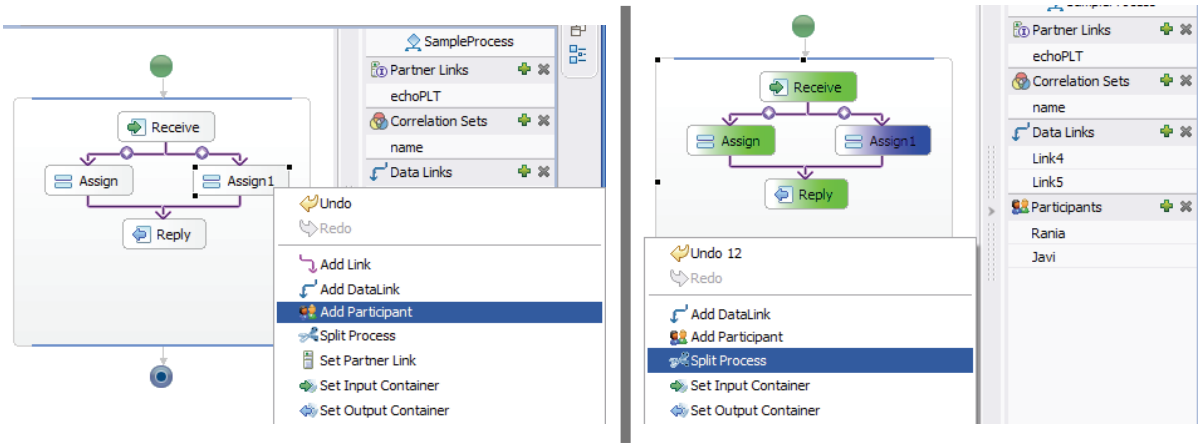


Figure 71: Splitting a process in the editor

Upon completing the partition, the user has the option, shown in the right side of Figure 71, to request the automatic creation of the fragments ("Split Process"). The editor checks that the partition is valid and creates the artifacts corresponding to the fragments [KHAL07B]. It then prompts the user for the necessary endpoints and deployment locations. Using this information and since ActiveBPEL is the basis for our BPEL engine supporting split processes, the editor directly generates the deployment artifacts in the format required by ActiveBPEL and transfers them to the deployment location of the corresponding engines.

The resulting fragments themselves can be opened in the editor, as shown in Figure 72 for the participant having the original activity 'Assign1'. Recall that this activity had one incoming control link, one outgoing split data link and one outgoing control link. All these links were split by the partition. Notice the explicit control link receiving block at the beginning of the process, with a link to the original activity 'Assign1'. On the bottom, one sees an explicit data

sending block on the left and an explicit control sending block on the right. The fault handlers are collapsed in this screenshot. Another thing to notice is that the sending blocks have assign activities in them. These are used to initialize the variables containing 'true'/'false', the correlation set, and the necessary data. In BPEL 2.0, this initialization can be done in the declaration of the variables thereby reducing the number of necessary assign activities. Further details on the editor implementation are provided in [VAZQ07].

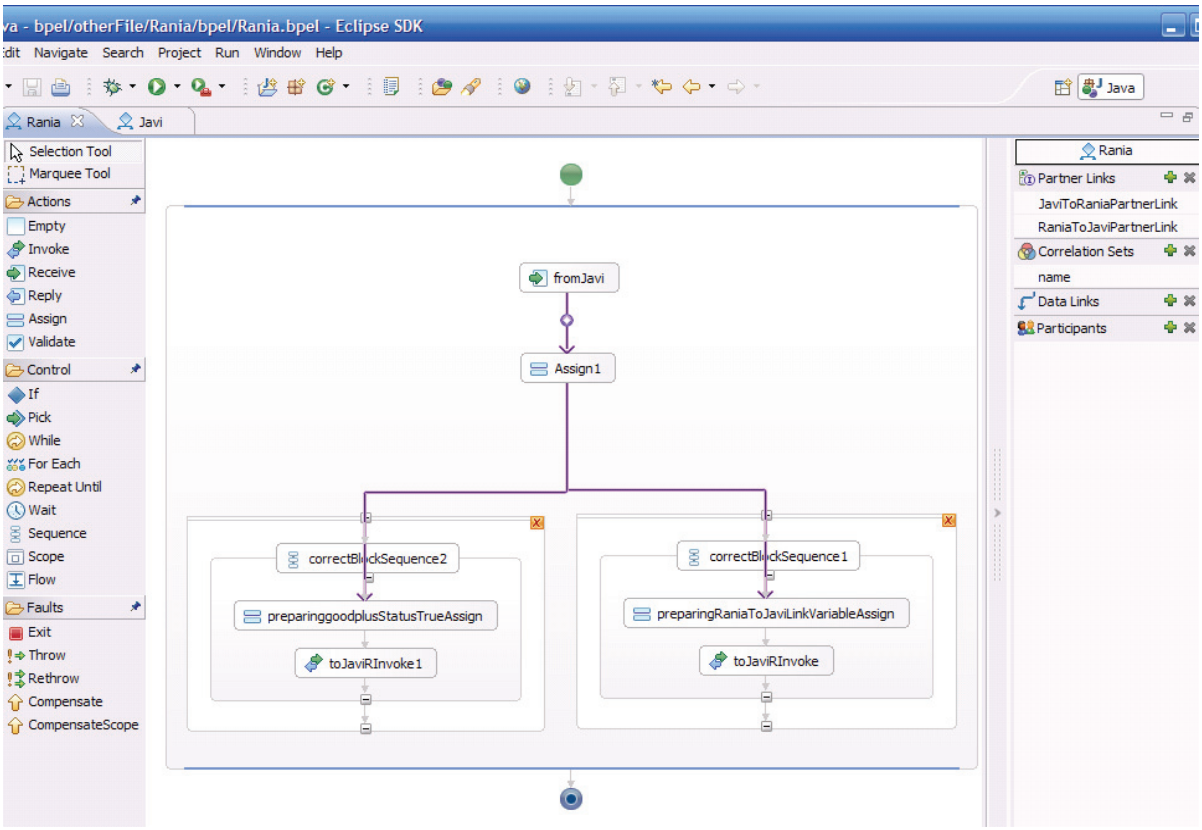


Figure 72: The BPEL process (fragment of the participant having 'Assign1')

A future extension of the editor will be able to hide the "artificial parts" of the fragments, generated by the splitting algorithms. One way in which this can be achieved is through the use of naming conventions on the newly added activities.

The runtime environment extends the open-source ActiveBPEL [ACEN07] engine following the architecture described above. The event-based system uses the ActiveMQ implementation [APAC07] of JMS to send and filter events between the Generic Controller and the Custom Controllers. This use of JMS for interacting with the ActiveBPEL engine is based on the BPEL monitoring extensions added in [WUTK06] and propagation of process instance lifecycle events to a monitor added in [NITZ06], [WUTK06].

The coordinator extends the WS-Coordination implementation presented in [VETT06] to provide the registration handler and the coordinator services and logic for the split loop and scope protocols. The coordinator stores the protocol and fragment state, including participant addresses, loop condition status and so forth, in stable storage using mySQL database tables.

According to section 6.4.3, endpoint references (EPRs) for each instance of the services involved in the coordination must be created; their structure is implementation-dependent. The prototype implements a generation scheme based on WS-Addressing, thus such an EPR

consists of an address that is a URL and a reference property for identifying different instances of the services at this URL. In the following section, square brackets are used for points of variability in the URL. The square brackets contain a description of the item that will be actually placed at that location at runtime. If a literal String value is used in such a description, it is between single quotation marks. The generation scheme is as follows:

- A single URL, in each participant, points to the participant protocol service of the loop protocol. Similarly, a single URL points to the participant protocol service of the scope protocol. A reference property named 'ParticipantInstanceIdentifier' is used in the EPR. This property is created by the participant logic such that it identifies exactly the right scope/loop instance for which the registration is taking place. The counter is incremented once for each registration of all fragments of a new instance of a scope or loop. It is created as follows:
  - Participant Instance Identifier:
    - http://www.iaas.uni-stuttgart.de/Participant/['Loop' or 'Scope']/[unsplit process name]/[participant name]/[loop or scope name]/[engine process instance identifier]/[counter]
- A URL for all coordinator-side loop protocol services, in the same coordinator, and another for all coordinator-side scope protocol services in the same coordinator. A reference property named 'CoordinatorInstanceIdentifier' is used as part of the coordinator side address. It is created by the coordinator's registration handler to uniquely identify this instance of the protocol. The coordinator process instance identifier is a value generated by the coordinator for each process instance it is coordinating. The counter is incremented once for each registration of all fragments of a new instance of a scope or loop. It is created as follows:
  - Coordinator Instance Identifier:
    - http://www.iaas.uni-stuttgart.de/Coordinatort/['Loop' or 'Scope']/[loop or scope name]/[unsplit process name]/[participant name]/[loop or scope name]/[coordinator process instance identifier]/[counter]

The participant side includes the implementation of the starter service and the participant protocol services of the coordination protocols. The starter service's implementation enables the BPEL engine to load a process instance based on a message from the coordinator instead of a message to an initiating receive activity. The implementation of the participant protocol services interact with the engine using the event-based approach described in section 7.3.

Figure 73 shows an example of running a split process requiring coordination. The top shows the process and its partition. The process is split so that the fault from the left fragment is caught by the handler in the right fragment. It does not have any split explicit dependencies (data or control links). The bottom part of the figure shows the state of each fragment process instance after their execution using this runtime. The state is shown using two screenshots of the ActiveBPEL engine's monitoring tool. In this tool, a check mark near an activity means that the activity completed successfully. An 'x' means that an activity faulted. An 'x' in a circle means that an activity was disabled. Notice in the left fragment that even though the scope faulted, the link leaving the scope fires normally and the 'reply' on the left runs. The reason is that the fault has been caught by the fault handler in the right fragment. Notice that in the right fragment, the wait activity has been disabled and the fault handler has run even though no visible fault is seen in this fragment itself. After the fault handler runs, the link from this scope fragment fires normally and the 'reply'on the right runs.

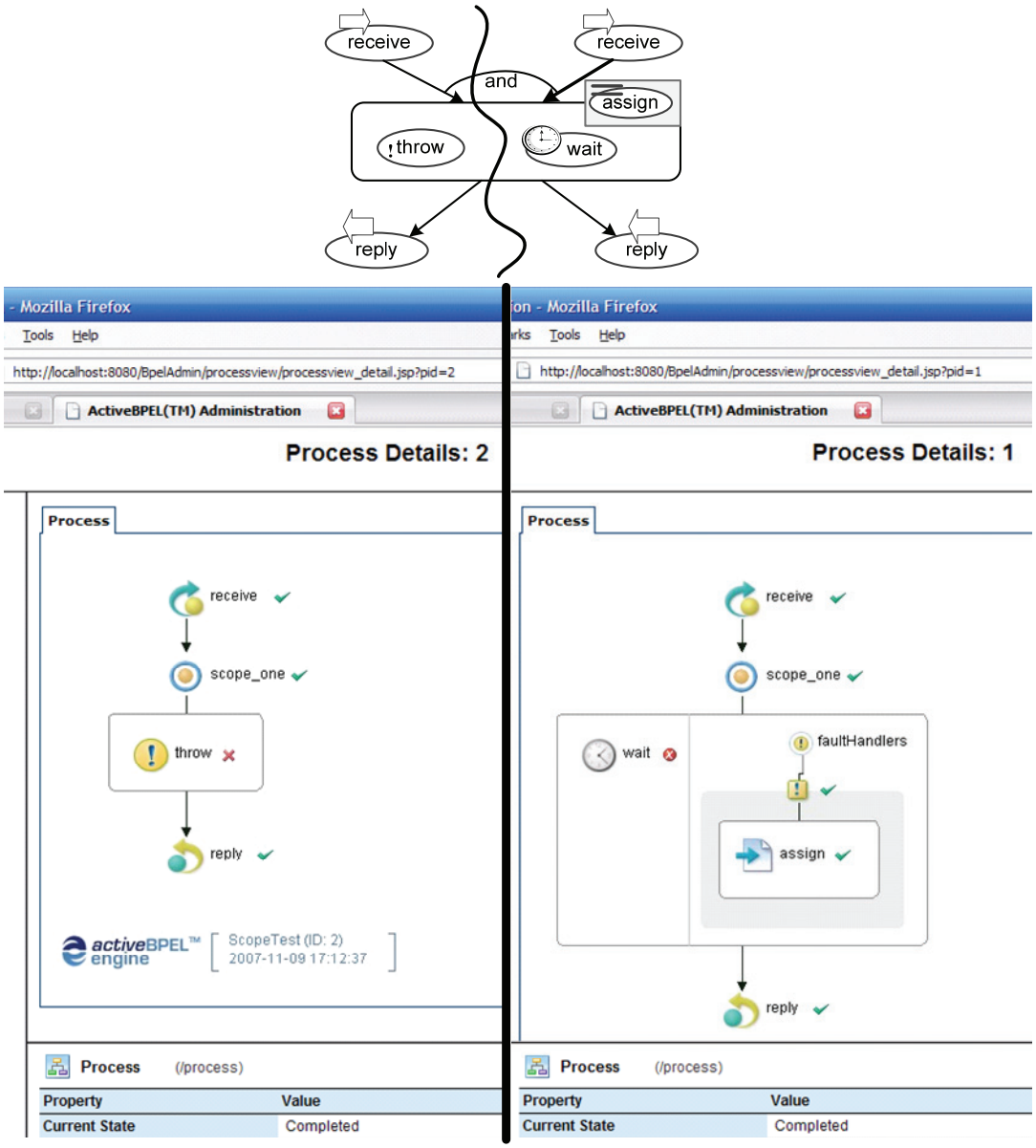


Figure 73: Running the fragments of the process on top

The system can run split processes with multiple nested loops and scopes, supporting the subset of behavior in [KHLE07] that excludes default compensation. Details, the WSDL definition of the protocol services, and several examples of varying levels of complexity are in [PALU07]. Currently, the fragment state on the participant side is kept in memory; to make the implementation more robust a future version should use persistent storage.

7.6 Conclusion

This section has introduced an architecture and implementation for the build-time and runtime support for business process fragmentation. The build-time consists of an editor for BPEL-D, with capabilities to assign different activities to different participants, and generation of process fragments and deployment artifacts. The new runtime is only needed if coordination is required for the split, as otherwise any standard-compliant BPEL engine can be used. The new runtime consists of an extended BPEL engine modified to support the new coordination protocols.

Chapter 8 Conclusion and Outlook

This contributions of this thesis started with a survey of Web services aggregation techniques (Chapter 2). A combination of three of these techniques was used to enable splitting business processes: orchestration, coordination, and recursive wiring.

The process meta-model (Chapter 3) provides graph-based processes with fault and compensation handling. It shows that direct support for fault handling and the use of the 'joinFailure' in fact enable one to support process languages like BPEL that combine the graph and calculus based approaches. A new compensation model is defined that, while divergent from compensation in BPEL, provides a simpler and more flexible approach for processes where links can be targeted at/from a scope itself or at/from activities inside a scope. The inherent complexity of each compensation model is the calculation of default compensation order, and in particular in the presence of loops. A book-keeping method was presented, enabling one to insure the compensation of proper instance groups in the presence of loops. The relationship between scopes, the fault handler look-up algorithm, the usage of compensation order graphs to encode default compensation order, and compensation book-keeping were used in subsequent chapters to enable splitting processes.

The ability to model explicit data flow in business processes that are closely aligned with Web services standards has been provided through the introduction of BPEL-D (Chapter 4). It has the same runtime semantics as BPEL from a control flow point of view, but it substitutes BPEL's implicit data flow with explicit data flow through the use of data links. Splitting BPEL-D processes is less complex than splitting BPEL due to the clear delineation of data dependencies. The explicit data links also make it easier for the designer to visualize the dependencies that will exist between process fragments.

The thesis has presented an automatic and operational semantics-preserving decomposition of business processes, in the presence of shared variables, loops, compensation, and fault handling. The approach has been shown to be interoperable through the use of open standards, as well as transparent. It has also met the goal of not requiring new middleware unless loops and scopes are split, in which case, it requires extensions to existing middleware (i.e. BPEL engine and WS-Coordination framework).

The fragmentation approach is presented in Chapters 5 and 6. A process is fragmented by assigning different activities to different participants. The specification of a partition is separate from the business process definition itself, increasing agility by enabling one to define more than one partition of the same business process. Algorithms are presented that generate local BPEL processes for each participant in a partition. These processes reproduce the control and data dependencies present in the non-split process using Web services messages across the Internet. The intricacies of BPEL's behavior, such as join failure propagation, aborting activities in case of a fault, and initializing data for a compensation handler, are reproduced by making extensive use of the fact that the fragments are themselves modeled as BPEL's processes. In other words, we introduced patterns of BPEL constructs for use in the fragments to re-enact unsplit BPEL behavior.

As expected, the amount of activities added to each fragment and the number of messages sent between the fragments increase with the amount of dependencies between the fragments. In particular, standard BPEL's shared variables result in creating 'receiving flows' to resolve write conflicts in keeping with the explicitly defined write order specified by the control flow. Two key enablers were identified for splitting the data dependencies in BPEL without new middleware: the Bernstein Criterion and maintaining write-order as prescribed by the explicit control flow instead of actual writer completion times.

Fragmentation of loops presents challenges in several aspects: synchronization of control, propagation of data (even for BPEL-D), and effects on compensation handling. This resulted in the use of a coordination protocol for splitting control in loops, the use of several activity patterns in the fragments to manage data dependencies, and book-keeping in the coordinator to maintain which scope instances form a compensation instance group. Scope fragmentation resulted in little change in the syntax of the fragments themselves but substantially larger coordination protocols than was the case for split loops. The coordinator was placed in charge of propagating the value of the loop condition as well as a substantial part of scope behavior, i.e. fault handler lookup, fault propagation, and default compensation order.

The concepts presented in this thesis are realized with the implementation presented in Chapter 7. The editor for BPEL-D was created by modifying an open-source BPEL editor. A user can use it to design BPEL-D processes, and also to split such processes. The user simply clicks on each activity and assigns it to the partner of choice. The editor then creates the fragments for each participant and any other information needed for deployment and coordination. The corresponding runtime for split processes supports our new coordination protocols for split loops and scopes. The runtime architecture is modular and provides extensible support for new BPEL extensions.

8.1 Future Work

During the course of this thesis, several areas have been identified for future research:

- Meta-model extensions for defining additional common properties on a set of activities using scopes: this would include a generic approach that can handle intersecting, overlapping scopes as well as relationships between new properties attached to these scopes. Quality of service policies are one example of such properties.
- Advanced BPEL data analysis: The state of the art on BPEL data analysis is still in its infancy. [KOKL07] was created specifically for the fragmentation algorithms in this thesis and provides smaller possible writer sets than earlier work. However, it can still be improved. One area, for example, that we are still exploring is the ability to analyze transition conditions on links to determine whether paths are mutually exclusive.
- Fragment evolution and autonomy, including migration and behavior modification: It is optimistic to assume that a fragment will not be changed once it has been created. The owner may need to modify some of its behavior or even simply migrate it to a new endpoint. Supporting such dynamicity and providing rules and best practices is an open research area. BPEL abstract processes are one technology that can aid in this endeavor.
- Versioning of an unsplit process and propagation of changes to existing partitions: It is not impossible that the original, unsplit process may need to be modified. The open questions involve how one maintains versions of the fragments and its process.
- Fragmentation Advisor: Currently, we require each and every activity to be assigned to a participant, and provide the mechanisms for creating and running the corresponding fragments. However, it is possible that one only requires a subset of the activities to be run

by certain participants. The rest can be placed anywhere. In such cases, it would be desirable to have a tool that can optimize activity placement according to predefined criteria (size and number of messages exchanged, geographic preferences, etc.). One could also present cost metrics of different partitions for the user. We are exploring this by combining our splitting approach with the region-based process analysis in [VAVL07].

- Process fragment design, reuse, and composition: In this thesis, we have used the term 'process fragments' to refer to fragments created from one process and that can always work together. However, the concept of process fragments is more general. One could start with a fragment and enable it to be stitched into more than one process model. In other words, fragments are of value on their own. They may be catalogued, searched, and sold. The area of fragment design, reuse, and composition has many interesting problems to be solved. For these, BPEL abstract processes again provide a technology of interest. Additionally, one may consider using our split scope contributions to draw scopes around a set of activities from multiple existing processes and cause them to run in unison.
- Distributed BPEL middleware and engines: We have attempted to keep middleware to a minimum, causing us to reproduce BPEL's split behavior using BPEL artifacts injected into the process models at design-time. We showed that unless one splits loops and scopes or starts from a BPEL process that must violate the Bernstein criterion, then no new middleware is needed. On the other end of the middleware-requirement spectrum are fully distributed business process engines such as OSIRIS [SWSS03] and the coordination space of Dumas et. al. [DFMV05]. Such engines require each participant to run that specific system. The problems for research include studying where the tipping points are, the requirements one would use for needing new middleware, and whether it would be possible to create a system that can provide a seamless transition along the different points on the middleware-requirements curve based on different user requirements.

Besides of the long-term research problems above, near-term work includes relaxing the restrictions on the unsplit process models in Chapter 5 and adding optimizations to the splitting algorithm by considering groups of dependencies. An example is whether and how receiving flows can be combined. Also, enhancement of the prototype could include increased coverage, enhancing usability of the modeling tool (e.g. views that add/remove the activities added by fragmentation), storing participant state in a database, or adding recovery and restart capabilities to the coordinator.

References

URLs in references were checked on Nov. 11, 2007.

University of Stuttgart Diploma theses are available online at http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-X where X is replaced by the thesis number, i.e.: 2444 for [MIET06].

University of Stuttgart technical reports are available online at http http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=TR-X-Y where X is replaced by the digits before the '/' in the report number and Y is replaced by the digits after it, i.e. 2007-01 for [KHLE07].

[AALS01] van der Aalst, W.M.P. and Weske, M., "The P2P Approach to Interorganizational Workflow", Proc. of CAiSE 2001, Springer, LNCS Volume 2068, Berlin, 2001.

[AALS99] van der Aalst, W.M.P., "Interorganizational workflows: An approach based on message sequence charts and petri nets", Systems Analysis-Modelling-Simulation, Volume 34, Number 3, pp. 335–367, 1999.

[AWBH00] van der Aalst, W. M. P., Barros, A. P., ter Hofstede, A. H. M. and Kiepuszewski. B., "Advanced Workflow Patterns", Proc. of Conference on Cooperative Information Systems (CoopIS2000), Springer, LNCS Volume 19, 2000.

[ACEN07] ActiveEndpoints, "ActiveBPEL Engine", online at http://www.activebpel.org/docs/architecture.html

[AABI07] Active Endpoints, Adobe, BEA, IBM, Oracle and SAP AG, "WS-BPEL Extension for People (BPEL4People)", Version 1.0, online at http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel4people/BPEL4People_v1.pdf , June 2007.

[AHSU06] Aho, A., Sethi, R. and Ullman, J., "Compilers: Principles, Techniques, and Tools", Second Edition, Addison Wesley, 2006.

[AAAB07] Alves, A., Arkin, A., Askary, S., Barreto, C., Bloch, B., Curbera, F., Ford, M., Goland, Y., Guízar, A., Kartha, N., Liu, C. K., Khalaf, R., König, D., Marin, M., Mehta, V., Thatte, S., van der Rijn, D., Yendluri, P. and Yiu A. (Eds.), "Web Services Business Process Execution Language Version 2.0", Apr. 2007, online at http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf

[APAC07] Apache Software Foundation. ActiveMQ. online at: http://activemq.apache.org/

[BAER73] Baer, J. L., "A Survey of Some Theoretical Aspects of Multiprocessing", Computing Surveys 5, No. 1, pp. 31-80, 1973.

[BAMM06] Baresi, L., Maurino A. and Modafferi, S., "Towards Distributed BPEL Orchestrations", Proc. of the Workshops on Software Evolution through

Transformations: Embracing the Change (SeTra 2006), Electronic Communications of the EASST, Volume 3, Sep. 2006.

[BADH05] Barros, A., Dumas, M. and ter Hofstede, A., "Service Interaction Patterns", Proc. of the International Conference on Business Process Management (BPM 2005), Nancy, France, Springer Verlag, Sep. 2005.

[BADO05] Barros, A., Dumas, M. and Oaks, P., "A Critical Overview of the Web Services Choreography Description Language (WS-CDL)", BPTrends Newsletter, Volume 3, Number 3, March 1, 2005.

[BCII06] BEA Systems, Cape Clear Software, IBM, Interface21, IONA Technologies PLC, Oracle, Primeton Technologies Ltd, Progress Software, Red Hat Inc., Rogue Wave Software, SAP AG, Siebel Systems, Software AG, Sun Microsystems, Sybase, TIBCO Software Inc., "Service Component Architecture", Nov. 2006, online at http://www.ibm.com/developerworks/library/specification/ws-sca/

[BEDM02] Benatallah, B., Dumas, M. and Maamar, Z., "Definition and Execution of Composite Web Services: The SELF-SERV Project", Data Engineering Bulletin, IEEE Computer Society Press, 25(4), 2002.

[BEDS05] Benatallah, B., Dumas, M. and Sheng, Q.Z., "Facilitating the Rapid Development and Scalable Orchestration of Composite Web Services", Distributed and Parallel Databases, Springer, Jan. 2005.

[BGKL04] Blow, M., Goland, Y., Kloppmann, M., Leymann, F., Pfau, G., Roller, D. and Rowley, M., "BPELJ: BPEL for Java Technology", 2004, online at http://www.ibm.com/developerworks/library/specification/ws-bpelj/

[BHKN04] Bunter, P., Hertlein, R., Khalaf, R. and Nadalin, A., "An Approach to Moving Industry Business Messaging Standards to Web services", IBM DeveloperWorks, Dec. 2004, online at http://www-106.ibm.com/developerworks/webservices/library/ws-move2ws.html

[CADI00] Casati, F. and Discenza, A., "Supporting Workflow Cooperation Within and Across Organizations", ACM Symposium on Applied Computing (SAC 2000), Como, Italy, ACM Press, March 2000.

[CCMG04] Chafle, G.B., Chandra, S., Mann, V. and Gowri Nanda M., "Decentralized Orchestration of Composite Web Services", Proc. of the World Wide Web Conference (WWW 2004), Alternate Track Papers & Posters, New York, NY, USA, ACM Press, 2004.

[CHKM07] Charfi, A., Khalaf, R. and Mukhi, N., "QoS-aware Web Service Compositions Using Non-Intrusive Policy Attachment to BPEL", Proc. of the Internation Conference on Software Oriented Computing (ICSOC 2007), Industry Track, Vienna, Austria, Springer LNCS, Sep. 2007.

[CHME04] Charfi, A. and Mezini, M., "Aspect-oriented Web Service Composition with AO4BPEL", Proc. of the European Conference on Web Services (ECOWS 2004), Erfurt, Germany, Springer LNCS, Volume 3250, Sep. 2004.

[COFI05] Courbis, C., Finkelstein, A., "Towards Aspect Weaving Applications", Proc. of the 27th International Conference on Software Engineering, ICSE 2005, IEEE Computer Society Press, pages 69–77, May 2005.

[CDKL07] Curbera, F., Duftler, M., Khalaf, R. and Lovell, D., “Bite: Workflow Composition for the Web”, International Conference on Service Oriented Computing (ICSOC 2007), Vienna, Austria, Springer LNCS, Sep. 2007.

[CDKM04] Curbera, F., Duftler, M., Khalaf, R, Mukhi, N., Nagy, W. and Weerawarana, S., “BPWS4J”, online at http://www.alphaworks.ibm.com/tech/bpws4j

[CDKN05] Curbera, F., Duftler, M., Khalaf, R., Nagy, W. A., Mukhi, N. and Weerawarana, S., “Colombo: Lightweight Middleware for Service Oriented Computing”, IBM Systems Journal, Volume 44, Number 1, 2005.

[CGKL03] Curbera, F., and Goland, Y., Klein, J., Leymann, F., Roller, D., Thatte, S. and Weerawarana, S., “Business Process Execution Language for Web Service v1.1”, online at http://www.ibm.com/developerworks/library/ws-bpel, May 2003.

[CKLW03] Curbera, F., Khalaf, R., Leymann, F. and Weerawarana, S., “Exception Handling in the BPEL4WS Language”, Proc. of the Conference on Business Process Management, BPM 2003, Eindhoven, the Netherlands, Springer LNCS, Volume 2678, June 2003.

[CUKM08] Curbera, F., Khalaf, R. and Mukhi, N., “Quality of Service in SOA Environments: An Overview and Research Agenda”, it – Information Technology Journal, Special Issue on Service-Oriented Architectures, Oldenbourg Wissenschaftsverlag, Munich, Germany (to appear)

[CKNW06] Curbera, F., Khalaf, R., Nagy, W. and Weerawarana, S., “Implementing BPEL4WS: The Architecture of a BPEL4WS Implementation”, International Journal of Concurrency and Computation: Practice and Experience, Special Issue on Grid Workflow, John Wiley & Sons, Volume 18, Number 1, January 2006.

[CUWD00] Curbera, F., Weerawarana, S. and Duftler, M. J., “On Component Composition Languages”, Proc. of the International Workshop on Component–Oriented Programming, Springer LNCS, May 2000.

[DAMO04] Damodaran, S., “B2B Integration over the Internet with XML: RosettaNet Successes and Challenges”, Proc. of WWW 2004, Alternate Track Papers and Posters, New York, NY, ACM Press, pp. 188–195, May 2004.

[DAMT99] Dashofy, E. M., Medvidovic, .N., and Taylor, R. N., “Using off-the-shelf middleware to implement connectors in distributed software architectures”, Proc. of the International Conference on Software Engineering, Los Angeles, California, USA, IEEE Computer Society Press, pp. 3–12, May 1999.

[DKLW07] Decker, G., Kopp, O., Leymann, F., and Weske, M., “BPEL4Chor: Extending BPEL for Modeling Choreographies”, Proc. of the IEEE International Conference on Web Services (ICWS 2207), Salt Lake City, Utah, USA, IEEE Computer Society Press, July 2007.

[DMCS05] Desai, N., Mallya, A. U., Chopra, A. K. and Singh, M. P., “Interaction Protocols as Design Abstractions for Business Processes”, IEEE Transactions on Software Engineering, Volume 31, Issue 12, pp. 1015-1027, Dec. 2005.

[DIDU04] Dijkman, R. and Dumas, M., “Service-oriented Design: A Multi-viewpoint Approach”, International Journal of Cooperative Information Systems, Volume 13, Number 4, World Scientific, Dec. 2004.

[DFMV05] Dumas, M., Fjellheim, T., Milliner, S. and Vayssiere J., “Event-based Coordination of Process-oriented Composite Applications”, Proc. of the International Conference on Business Process Management (BPM 2005), Nancy, France, Springer Verlag, Sep. 2005.

[DUHA05] Dumas, M., ter Hofstede, A. And van der Aalst, W.M.P. (Ed.) “Process Aware Information Systems: Bridging People and Software Through Process Technology”, John Wiley and Sons, 2005

[ECLI07A] Eclipse.org, The Eclipse BPEL Project, online at http://www.eclipse.org/bpel

[ECLI07B] Eclipse.og, The Eclipse Modeling Framework Project (EMF), online at http://www.eclipse.org/modeling/emf/

[EDLI96] Eder, J. and Liebhart, W., “Workflow Recovery”, Proc. of the International. Conference on Cooperative Information Systems (CoopIS’96), Bruxelles, Belgium, 1996.

[FAGV05] Farahbod, R., Glässer, U. and Vajihollahi, M., “A Formal Semantics for the Business Process Execution Language for Web Services”, Proc. 3rd International Workshop on Web Services: Modeling, Architecture and Infrastructure at the International Conference on Enterprise Information Systems (ICEIS'05), Miami, FL, May 2005.

[FIDG91] Fidge, C., “Logical Time in Distributed Computing Systems”, Computer Volume 24, Number 8, Aug. 1991.

[FRIE05] Friedman, T., “The World is Flat: A Brief History of the Twenty-First Century”, Farrar, Straus and Giroux, 2005.

[FRWK02] Fremantle, P., Weerawarana, S. and Khalaf, R., “Enterprise Services in the Communications of the ACM: Special Issue on Enterprise Computing”, 45(10), pp. 77-82, ACM Press, Oct. 2002.

[FUBS03A] Fu, X., Bultan, T. and Su, J., “A top-down approach to modeling global behaviors of web services”, Requirements Engineering for Open Systems Workshop (REOS 2003), Monterey, California, Sep. 2003.

[FUBS03B] Fu, X., Bultan, T. and Su, J., “Conversation specification: A new approach to design and analysis of e-service composition”, Proc. of the International World Wide Web Conference (WWW2003), Budapest, Hungary, May 2003.

[FUBS04] Fu, X., Bultan, T., Su, J., “Analysis of interacting BPEL web services”, Proc. of the World Wide Web Conference (WWW 2004), New York, NY, ACM Press, May 2004.

[FUGR02] Furniss, P. and Green, A., “WS-C+T and BTP: Comments and Comparisons”, 2002, online at http://lists.oasis-open.org/archives/business-transaction/200211/msg00013.html

[GASC99] Gallo, G. and Scutella, M.G., “Directed Hypergraphs as a Modeling Paradigm”, University of Pisa, Technical Report No. TR-99-02, Feb. 1999.

[GLAB90] van Glabbeek, R. J., “The Linear Time-Branching Time Spectrum”, Proc. of CONCUR 90, Springer-Verlag LNCS, Number 458, 1990.

[GODL06] Godlinski, A., “Static Analysis to Discover Communication Relevant Data Flows in BPEL-Processes”, Diploma Thesis, Friedrich Schiller University, Jena, October 2006.

[GOCS04] Gowri Nanda, M., Chandra, S. and Sarkar, V., “Decentralizing Execution of Composite Web services”, OOPSLA 2004, Vancouver, British Columbia, Canada, ACM Press, pp.170-187, Oct. 2004.

[HAAL98] Hagen, C. and Alonso, G., “Flexible exception handling in the OPERA process support system”, Proc. of the International Conference on Distributed Computing Systems, pp. 526–533, 1998.

[HOKA03A] Hondo, M. and Kaler, C., “Web Services Policy Attachment (WS-PolicyAttachment)”, 2003, online at http://www-106.ibm.com/developerworks/library/ws-polatt/

[HOKA03B] Hondo, M. and Kaler, C., “Web Services Policy Framework (WS-Policy)”, 2003, online at http://www-106.ibm.com/developerworks/library/ws-polfram/

[IBSA05] IBM, SAP AG, “WS-BPEL Extension for Sub-processes – BPEL-SPE”, Sep. 2005, online at http://www-128.ibm.com/developerworks/library/specification/ws-bpelsubproc/

[JOHA02] Jouvin, D. and Hassas, S., “Role Delegation as Multi-Agent oriented Dynamic Composition”, Proc. of NetObjectDays, 2002.

[KABR04] Kavantzas, N., Burdett, D. and Ritzinger, G. (ed), “Web Services Choreography Language (WS-CDL1.0)”, online at http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427

[KKSP06] Karastoyanova, D., Khalaf, R., Schroth, R., Paluszek, M. and Leymann, F., “BPEL Event Model”, University of Stuttgart, Technical Report 2006/10. Nov. 2006.

[KLNW06] Karastoyanova, D., Leymann, F., Nitzsche, J., Wetzstein, B. and Wutke, D., “Parameterized BPEL Processes: Concepts and Implementation”, Proc. of the International Conference on Business Process Management (BPM 2006), Short Papers, Vienna, Austria, Sep. 2006.

[KHAL05A] Khalaf, R., “From RosettaNet PIPs To BPEL Processes: A Three Level Approach for Business Protocols”, Proc. of the International Conference on Business Process Management (BPM 2005), Industry Track, Springer LNCS, Nancy, France, Sep. 2005.

[KHAL07A] Khalaf, R., “From RosettaNet PIPs To BPEL Processes: A Three Level Approach for Business Protocols”, International Journal on Data and Knowledge Engineering (DKE), Elsevier, Volume 61, Issue 1, pp. 23-38, Apr. 2007.

[KHAL07B] Khalaf, R., “Note on Syntactic Details of Split BPEL-D Business Processes”, University of Stuttgart Technical Report 2007/02, July 2007.

[KHKL07A] Khalaf, R., Karastoyanova, D. and Leymann, F., “Pluggable Framework for Enabling the Execution of Extended BPEL Behavior”, Proc. of the ICSOC

International Workshop on Engineering Service-Oriented Application: Analysis, Design and Composition (WESOA 2007), Vienna, Austria, Springer LNCS, Sep. 2007.

[KHKL07B] Khalaf, R. and Kopp, O., and Leymann, F., "Maintaining Data Dependencies Across BPEL Process Fragments", Proc. of the International Conference on Service Oriented Computing, ICSOC 2007, Vienna, Austria, Springer LNCS, Sep.2007.

[KHLE07] Khalaf, R. and Leymann, F., "Coordination Protocols for Split BPEL Loops and Scopes", University of Stuttgart, Technical Report 2007/01, March 2007.

[KHLE03] Khalaf, R. and Leymann, F., "On Web Services Aggregation", Proc. of the VLDB Technologies for e-Services Workshop (VLDB T-ES 2003), Berlin, Germany, Springer LNCS, Sep. 2003.

[KHLE06] Khalaf, R. and Leymann, F., "Role Based Decomposition of Business Processes using BPEL", Proc. of the IEEE International Conference on Web Services (ICWS'06), Industry Track, Chicago, IL, Sep 2006.

[KHLK06] Khalaf, R., Leymann, F. and Keller, A., "Business processes for Web Services: Principles and applications", IBM Systems Journal, Special Issue Celebrating 10 Years of XML, Volume 45, Number 2, 2006.

[KHMW03] Khalaf, R. and Mukhi, N., and Weerawarana, S., "Service-Oriented Programming in BPEL4WS", Proc. of the World Wide Web Conference (WWW 2003), Web Services Track, Budapest, Hungary, Kluwer, May 2003.

[KKLP04] Kloppmann, M., Koenig, D., Leymann, F., Pfau, G. and Roller, D., "Business process choreography in WebSphere: Combining the power of BPEL and J2EE", IBM Systems Journal, Volume 43, Number 2, 2004.

[KOEN06] Koenig, D., "Issue R26: Default Compensation Order Conflict", WS BPEL public review issues list, Oct. 2006, online at http://www.choreology.com/external/WS_BPEL_review_issues_list.html#IssueR26

[KOKL07] Kopp, O., Khalaf, R., and Leymann F., "Determination of Explicit Data Links in BPEL Processes", University of Stuttgart, Technical Report 2007/04, Nov. 2007

[LEMP97] Lee, J., Midkiff, S.P., Padua D.A., "Concurrent Static Single Assignment Form and Constant Propagation for Explicitly Parallel Programs", Proc. of the 10th International Workshop on Languages and Compilers for Parallel Computing, LNCS, Volume 1366. Springer-Verlag, 1997.

[LEYM95] Leymann, F., "Supporting Business Transactions via Partial Backward Recovery in Workflow Management Systems", Proc. of the Conference on Database Systems for Business, Technology, and Web (BTW'95), Dresden, Germany, Springer, Mar. 1995.

[LEYM01] Leymann, F., "Web Services Flow Language (WSFL) 1.0", IBM Corp., May 2001.

[LEYM03] Leymann, F., "Web services: Distributed applications without limits - an

outline", Proc. of the Conference on Database Systems for Business, Technology, and Web (BTW 2003), LNCS, Springer–Verlag, Feb 2003.

[LEAL94] Leymann, F. and Altenhuber, A, "Managing business processes as information resources", IBM Systems Journal, Volume 33, Number 2, pp. 326 – 348, 1994.

[LEPO05] Leymann, F. and Pottinger, S., "Rethinking the coordination models of WS-Coordination and WS-CF", Proc. of the European Conference on Web Services (ECOWS 2005), 2005.

[LERO00] Leymann, F. and Roller, D., "Production Workflow", Prentice Hall, 2000.

[LIFR03] Little, M. and Freund, T., "A comparison of Web Services Transaction Protocols", online at http://www-128.ibm.com/developerworks/webservices/library/ws-comproto, Oct. 2003.

[LITT03] Little, M., "Transactions and Web Services: Helping to Make E-Commerce a Reality", CACM, Oct. 2003.

[LUMA07] Lucchi, R. and Mazzara, M., "A pi-calculus based semantics for WS-BPEL", Journal of Logic and Algebraic Programming, Web Services and Formal Methods, Elsevier, Volume 70, Issue 1, pp. 96-118 Jan. 2007.

[LKDK03] Ludwig, H., Heller, A., Dan, A., King, R.P. and Franck, R., "A Service Level Agreement Language for Dynamic Electronic Services", Journal of Electronic Commerce Research, 3, March 2003.

[MEHA05] Mendling, J. and Hafner, M., "From Inter-Organizational Workflows to Process Execution: Generating BPEL from WS-CDL", Modeling Inter-Organizational Systems Workshop (MIOS), Proc. of OTM 2005 Workshops, Agia Napa, Cyprus, Springer LNCS 3762, pp. 506-515, Nov. 2005.

[MAMC03] Mandell, D. and McIlraith, S., "Adapting BPEL4WS for the Semantic Web: The Bottom-Up Approach to Web Service Interoperation", Proc. of the International Semantic Web Conference (ISWC2003), Sanibel Island, Florida, 2003.

[MART05] Martens, A., "Consistency between Executable and Abstract processes", Proc. of the IEEE Conference on e-Technology, e-Commerce and e-Service (EEE 2005), Hong Kong, Mar. 2005.

[MATK97] Magee, J., Tseng, A., and Kramer, J., "Composing Distributed Objects in CORBA", Proc. of the International Symposium on Autonomous Decentralized Systems (ISADS97), 1997.

[MIET06] Mietzner, R., "Extraction of WS-Business Activity from BPEL 1.1", University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Diploma Thesis 2444, 2006.

[MILN99] Milner, R., "Communicating and Mobile Systems: The π-Calculus", Cambridge University Press, 1999.

[MMGA07] Moser, S., Martens, A., Görlach, K., Amme, W. and Godlinski, A., "Advanced Verification of Distributed WS-BPEL Business Processes Incorporating CSSA-based Data Flow Analysis", SCC 2007,IEEE International Conference on Services Computing, July 2007, Salt Lake City,

Utah.

[MUCH97] Muchnick, S., “Advanced Compiler Design and Implementation”, Morgan Kaufmann Publishers, 1997.

[MWWK98] Muth, P., Wodkte, D., Wiessenfels, J., Kotz, D.A. and Weikum, G., “From Centralized Workflow Specification to Distributed Workflow Execution”, Journal of Intelligent Information Systems, Volume 10, Number 2, 1998.

[NITZ06] Nitzsche, J., “Entwicklung eines Monitoring-Tools zur Unterstützung von parametrisierten Web Service Flows”, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Diploma Thesis 2388, 2006.

[OASI03] OASIS, “Web Services for Remote Portlets Specification Version 1.0”, Mar. 2003, online at http://www.oasis-open.org/committees/download.php/3343/oasis-200304-wsrp-specification-1.0.pdf

[OASI07A] OASIS, “Web Services Atomic Transaction (WS-AtomicTransaction) Version 1.1”, online at http://docs.oasis-open.org/ws-tx/wstx-wsat-1.1-spec.pdf

[OASI07B] OASIS, “Web Services Business Activity (WS-BusinessActivity) Version 1.1”, online at http://docs.oasis-open.org/ws-tx/wstx-wsba-1.1-spec.pdf

[OASI07C] OASIS, “Web Services Coordination (WS-Cooridnation) Version 1.1”, Apr. 2007, online at http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.1-spec.pdf

[PALU07] Paluszek, M., “Coordinating Distributed Loops and Fault Handling, Transactional Scopes using WS- Coordination protocols layered on WS-BPEL services”, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Diploma Thesis 2586, 2007.

[PAAR98] Papadopoulos, G. and Farhad, A., “Coordination Models and Languages”, Advances in Computers Journal, 46, Academic Press, 1998.

[PAOS03] Patil, A., Oundhakar, S. and Sheth A., “Semantic Annotation of Web Services (SAWS)”, Technical Report, LSDIS Lab, U. of Georgia, Mar. 2003.

[PITT06] Pitts, Andrew, “Glossary of Mathematical Notation and Terminology”, University of Cambridge, 2006, online at http://www.cl.cam.ac.uk/teaching/2006/CompTheory/comt09.pdf

[REIS85] Reisig, Wolfgang., “Petri Nets’’ Springer-Verlag. 1985.

[RUGG96] Rudolph, E., Grabowski, J. and Graubmann, P., “Tutorial on Message Sequence Charts (MSC'96)”, Tutorials of the Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification (FORTE/PSTV), 1996.

[SAME05] Sauter, P. and Melzer, I., “A Comparison of WS-BusinessActivity and BPEL4WS Long-Running Transaction”, KiVS 2005, pp.115-125, 2005.

[SASI93] Sarkar, V. and Simons, B., “Parallel Program Graphs and their Classification”, Proc of the 6th International Workshop on Languages and

Compilers for Parallel Computing, Portland, Oregon, USA, Springer LNCS, Aug. 1993.

[SCST04] Schmidt, K. and Stahl, C., "A petri net semantic for BPEL4WS - validation and application", Proc. of 11th Workshop on Algorithms and Tools for Petri Nets, 2004

[SCHR06] Schroth, R., "Specification, Design and Implementation of a BPEL Engine with AOP Support and an Aspect Weaver for BPEL Processes", University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Diploma Thesis 2523, 2006.

[SHDF03] Sheshagiri, M., desJardins, M. and Finin, T., "A Planner for Composing Services described in DAML-S", Proc. of the Conference on Autonomous Agents and Multi–Agent Systems(AAMAS03), Workshop on Web Services and Agent–based Engineering, Melbourne, Australia, July 2003.

[SREE02] Sreedhar, V. C. "Mixin'up components", Proc. of the International conference on Software engineering (ICSE2002), Orlando, Florida, 2002.

[SUMS99] Sullivan, K., Marchukov, M. and Socha, J., "Analysis of a Conflict between Aggregation and Interface Negotiation in Microsoft's Component Object Model", IEEE Transaction on Software Engineering, Volume 25, Number 4, 1999.

[SWSS03] Schuler, C., Weber, R., Schuldt, H. and Scheck, H.J., "Peer-to-Peer Process Execution with OSIRIS", Proc. of the International Conference on Service Oriented Computing (ICSOC 2003), Trento, Italy, Springer LNCS, Dec. 2003.

[TAIS05] Tai, S., "Composing web services specifications: Experiences in implementing policy-driven transactional processes", Proc. of the Conference on Database Systems for Business, Technology, and Web (BTW 2005), LNI, Volume 65, pp. 547–559, 2005.

[TAKM04] Tai, S., Khalaf, R. and Mikalsen, T., "Composition of Coordinated Web Services", Proc. of ACM/IFIP/USENIX International Middleware Conference (Middleware 2004). Springer LNCS, Volume 3231, Toronto, Canada, 2004.

[TMWD04] Tai, S., Mikalsen, T., Wohlstadter, E., Desai, N. and Rouvellou, I., "Transaction Policies for Service-Oriented Computing", Data and Knowledge Engineering Journal, Special Issue on Contract-based Coordination and Collabortation, 2004.

[TTVX02] Tan, Y.S., Topol, B., Vellanki, V. and Xing, J., "Manage web services and grid services with service domain technology", 2002, online at http://www.ibm.com/developerworks/library/gr-servicegrid/

[THAT01] Thatte, S., "XLANG", Mirosorft Corp., 2001.

[VAVL07] Vanhatalo, J., Völzer, H., and Leymann, F., "Faster and more focused control-flow analysis for business process models though SESE decomposition", Proc. of the International Conference on Service-Oriented Computing (ICSOC 2007), Vienna, Austria, Springer LNCS, Volulme 4749, Sep. 2007.

[VAZQ07] Vazquez Fernandez, J., “BPEL with Explicit Data Flow: Model, Editor, and Partitioning Tool”, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Diploma Thesis 2616, 2007.

[VETT06] Vetter, T., “Anpassung und Implementierung verschiedener Transaktionsprotokolle auf WS-Coordination”, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Diploma Thesis 2386, 2006.

[W3CW04] W3C, “Web Services Addressing (WS-Addressing)”, online at http://www.w3.org/Submission/ws-addressing/ , Aug. 2004.

[W3CS00] W3C, “Simple Object Access Protocol (SOAP) 1.1", http://www.w3.org/TR/SOAP , May 2000.

[W3CW02A] W3C, “Web Services Description Language (WSDL)”, online at http://www.w3.org/2002/ws/desc/ , June 2006.

[WCDE01] Weerawarana, S., Curbera, F., Duftler, M. J., Epstein, D. A., and Kesselman, J., “Bean Markup Language: a Composition Language for JavaBeans Components”, Proc. of the USENIX Conference on Object-Oriented Technologies and Systems, San Antonia, Texas, Jan. 2001.

[WCLS05] Weerawarana, S., Curbera, F., Leymann, F., Storey, T. and Ferguson, D. (Ed.), “The Web Services Platform Architecture”, Addison Wesley. Apr. 2005.

[WEVO02] Weikum, G. and Vossen, G., “Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery”, Morgan Kaufmann Publishers, 2002.

[WTMR04] Wohlstadter, E., Tai, S., Mikalsen, T., Rouvellou, I. and Devanbu, P., “GlueQoS: middleware to sweeten quality-of-service policy interactions”, Proc. of the International Conference on Software Engineering (ICSE 2004), Scotland, UK, IEEE Computer Society Press, May 2004.

[WUTK06] Wutke, D., “Erweiterung einer Workflow-Engine zur Unterstützung von parametrisierten Web Service Flows", Diploma Thesis 2401, University of Stuttgart, 2006.

[YILD07] Ustun Y., “On Dead Path Elimination in Decentralized Process Executions”, INRIA Research Report, RR-6131, 2007.

[ZBDK03] Zeng, L., Benatallah, B., Dumas, M., Kalagnanam, J.and Sheng, Q., “Quality Driven Web Services Composition”, Proc. of the World Wide Web Conference (WWW2003), Budapest, Hungary, May 2003.