

Universität Stuttgart

# Throughput Improvements for BPEL Engines: Implementation Techniques and Measurements Applied to SWoM

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

Vorgelegt von  
Dieter Roller  
aus Maichingen

**Hauptberichter:** Prof. Dr. Frank Leymann

**Mitberichter:** Prof. Dr. Bernhard Mitschang

Prof. Dr. Cesare Pautasso

**Tag der mündlichen Prüfung:** 26. Juli 2013

Institut für Architektur von Anwendungssystemen

2013



[1] *op-ti-mi-za-tion*: an act, process, or methodology of making something (as a design, system, or decision) as fully perfect, functional, or effective as possible; specifically : the mathematical procedures (as finding the maximum of a function) involved in this.

*Merriam-Webster Online Collegiate Dictionary*

[2] The three most important items in database research are:  
performance, performance, performance

*Prof. Dr. Andreas Reuter*



# ZUSAMMENFASSUNG

In den vergangenen zwei Jahrzehnten hat sich die Workflow-Technologie zu einem wesentlichen Bestandteil moderner Anwendungssysteme entwickelt, insbesondere für solche Systeme, die auf der service-orientierten Architektur (SOA) beruhen. Mit Hilfe der Workflow-Technologie können Geschäftsprozesse erstellt werden, die schnell an geänderte Umgebungen angepasst werden können. Sie bildet zudem die Basis des zweistufigen Programmierparadigmas.

Workflow Management Systeme, die die Workflow-Technologie implementieren, sind eine Kernkomponente der Middleware, deren Performance wesentlich die Ausführungsgeschwindigkeit der Anwendungen bestimmt, die mit ihnen gebaut werden.

Die vorgestellte Dissertation entwickelt einen Satz von Optimierungstechniken für ein modernes Workflow Management System, das Robustheit mit Performance kombiniert. Dieses Workflow Management System wird Stuttgarter Workflow Maschine (SWoM) genannt, nach dem Ort der Entwicklung. Folgende neuartige Ansätze wurden entwickelt, um das Ziel zu erreichen.

Erstens, die Entwicklung des Konzeptes von Transaktionsflüssen. Transaktionsflüsse bestimmen die Menge der Transaktionen, die die Workflow-Engine bei der Ausführung eines Geschäftsprozesses durchführt. Ein Flussoptimierer erstellt optimierte Transaktionsflüsse, die es der Workflow-Engine ermöglichen, Geschäftsprozesse erheblich effizienter abzuarbeiten.

Zweitens wurde Cachingfunktionalität in allen Komponenten der SWoM, soweit wie möglich, implementiert. So werden alle Prozessmodelle und Prozessinstanzen während der Ausführung einer Transaktion im Speicher der Workflow-Engine gehalten. Bei schnelllaufenden Geschäftsprozessen werden die Prozessdaten zwischen zwei aufeinanderfolgenden Navigationsschritten im Speicher der Workflow-Engine gehalten werden.

Drittens benützt die SWoM, um die notwendige Robustheit zur erzielen, IBM WebSphere als Laufzeitumgebung und IBM DB2 zur Speicherung der Daten. Mehrere Techniken wurden entwickelt, um die Infrastruktur zur Laufzeit optimal auf die Arbeitsweise der SWoM abzustimmen.

Viertens: Konfigurationsoptionen erlauben es dem Prozessmodellierer, vordefinierte Optimierungstechniken, wie Optimierung des Datenbankzugriffes oder Verbesserung von Aufrufen von Web Services, zu nutzen.

Fünftens: Ein Flussoptimierer optimiert die Ausführung der Transaktionsflüsse bezüglich Cache-, Datenbank- und CPU-Nutzung durch den Einsatz von Datenfluss- und Transaktionsanalysetechniken. Die Optimierung basiert auf Eingaben des Prozessmodellierers sowie Statistiken, die die SWoM während der Ausführung der Geschäftsprozesse sammelt.

Sechstens: Die aufgeführten Optimierungstechniken, die die erforderliche Robustheit gewährleisten, werden durch einen Satz von Optimierungstechniken ergänzt, die Robustheitsanforderungen mit dem Ziel der Performance-Verbesserungen reduzieren.

Alle Techniken wurden auf Basis eines einfachen, jedoch sehr aussagekräftigen Benchmarks validiert. Der Benchmark fokussiert sich auf schnelle, nachrichten-basierte Interaktionen mit Web Services, synchrone Aufrufe von Web Services, Nachrichtentransformationen sowie paralleler Ausführung von parallelen Pfaden eines Geschäftsprozesses.

Die Resultate des Benchmarks zeigen, dass die SWoM fast linear bezüglich CPU-Last sowie der Anzahl der gleichzeitigen Anfragen skaliert. Die SWoM ist in der Lage auf einer Vierkern-CPU mit Windows (64bit) mehr als 100 Prozessinstanzen in der Sekunde abzuarbeiten. Dies übertrifft signifikant alle Workflow Management Systeme, für die entsprechende Performance-Daten vorliegen.

Weitere Performance-Verbesserungen können durch den Einsatz von Cluster-Technologie erzielt werden. Die Architektur der SWoM erlaubt dies mit geringfügigen Änderungen.



# ABSTRACT

Workflow Technology has become over the last two decades the cornerstone of modern application systems, in particular those built upon Service Oriented Architecture (SOA). It helps implement business processes that can be easily adapted to the changing needs of a dynamic environment and provides the base for the two-level application development paradigm.

Workflow management systems (WfMS) deliver the functions of workflow technology; they have become a critical middleware component whose performance characteristics are significantly impacting the overall performance of the applications that have been built.

The purpose of this thesis is to develop a set of optimization techniques for a state-of-the-art WfMS that delivers the required robustness with the best achievable performance. This WfMS has been labeled Stuttgarter Workflow Maschine (SWoM) to emphasize its birth place. Several novel approaches have been developed to achieve the desired goal.

First, the concept of transaction flows has been developed as the base for a flow optimizer that significantly improves the performance of the workflow engine. A transaction flow defines the set of transactions that the workflow engine needs to carry out when processing a particular process instance.

Second, the notion of caching is driven into virtually all areas of the different components that make up the SWoM, such as the caching of process

models, process instances, or even the caching of process information between subsequent processing steps of a particular process instance.

Third, the exploitation and tie-in into the underlying infrastructure for optimal resource exploitation and load balancing, where the infrastructure, IBM WebSphere for the application server and IBM DB2 as the database environment, provides for the necessary robustness.

Fourth, the support of flow configuration options helps the process modeler to improve performance by telling the SWoM to exploit for a particular process model a set of pre-defined optimization approaches, such as the optimization of database accesses or the improvement of Web Service invocations.

Fifth, a flow optimizer optimizes the execution of the transaction flows with respect to cache, database, and CPU cycle usage through the exploitation of data flow and transaction analysis techniques. Optimization is done based on user recommendations and statistical information that the SWoM collects during execution.

These optimization techniques, that maintain middleware robustness, are complemented by a set of optimization techniques that improve performance by relaxing some of the stringent robustness requirements.

All techniques have been validated using a simple, yet expressive benchmark that focuses on high-speed, message-based interactions, synchronous invocation, message transformation, and parallel execution.

The benchmark results show that the SWoM scales almost linearly with respect to CPU load and the parallelism of requests. The maximum number of requests that the SWoM could obtain on a quad core CPU running Windows 7 64bit was more than 100 process instances per second, outperforming by a factor those WfMSs for which performance data is available.

The architecture of the SWoM provides improved performance through the transparent exploitation of cluster technology that IBM WebSphere provides; very few optimization techniques required specific adaptations to cope with cluster characteristics.

# ACKNOWLEDGEMENT

It has been a long journey with workflow technology; many people have helped me to understand it's great potential.

The first person with whom I worked on workflow was *Prof. Dr. Frank Leymann*; I was a planning manager at IBM's PPC laboratory responsible for case processing, and Frank was developing with a small team the initial ideas behind workflow. Fifteen years of working together followed this initial encounter. And the journey continued: after retiring from IBM I fulfilled an old promise and started this thesis in Frank's department. It gave me the chance and motivation to focus on something I always wanted to understand in detail: what needs to be done to get the most out of a WfMS.

Shortly after these initial thoughts were written, IBM started development of a WfMS, called IBM FlowMark. It strictly separated the three orthogonal aspects of workflow technology: the process model describing the different tasks that need to be carried out (the *what* dimension), the people that need to carry out the individual tasks (the *who* dimension), and the computer programs that are used to carry out the assigned tasks (the *which* dimension). Many discussions with the head of the IBM laboratory in Böblingen, *Dr. Edwin Vogt*, helped us to come up with this fundamental structure of workflow technology, commonly shown as a three-dimensional space (the  $W^3$  space) with the executing business process as a curve in that three-dimensional space.

IBM FlowMark suffered to some extent from the missing robustness of the underlying technology. It was *Steve Mills*, the head of IBM's software business, who firmly believed in the importance of workflow technology, who gave Frank and me the chance to re-implement the WfMS using a new architecture we had developed. The fundamental concept was the usage of stateless servers that used a relational database system for storing information, message queuing for communication between the different components, and transactions for obtaining the necessary robustness. The development was done by a team at the IBM laboratory in Böblingen, and the product launched in 1998 under the name MQSeries Workflow. It was quite successful and showed the applicability of workflow technology to areas we had never envisioned. I'm proud to have been the lead architect for the product and would like to thank all the colleagues who have helped turn this vision of a workflow management system into reality: *Matthias Kloppmann*, *Gerhard Tonn*, and *Claudia Zentner*.

The success of workflow technology and the appearance of more and more WfMSs mandated the need for a standard. This coincided with the advent of Web Services, so it was natural to develop the standard as a part of the Web Services stack. I was part of the IBM architecture team that, together with the architecture lead from Microsoft *Satish Thatte*, developed the first version of Business Process Execution Language for Web Services (WS-BPEL). It was a one of a kind experience in my professional life to work together in a team of competing companies and become friends.

There are many other IBM people that helped shape my understanding of workflow: *Sanjiva Weerawarana* and *Francisco Curbera*, who were the WSDL, SOAP, and XML experts in the WS-BPEL group, *Dieter König*, who helped tremendously in making WS-BPEL a standard, and *Gerhard Pfau*, the IBM performance expert for workflow.

Discussions with people from the academia also helped to shape the understanding of what workflow technology could do in other areas, such as database technology. In particular, I would like to mention *Prof. Dr. Bernhard Mitschang* with whom I had many discussions; I particularly remember those we had during preparation of a patent that the University of Stuttgart and IBM filed together.

I also would like to thank all my students who either worked directly on the SWoM, which has started as a student project, or provided answers to a number of questions, in particular in the area of infrastructure deployment.

I'm particularly grateful to *Prof. Dr. Cesare Pautasso* for taking the time to serve as an evaluator for the thesis.

In all these many years, working hard for IBM or working on the thesis, it was my family that helped me tremendously. Without their assistance and particularly their patience it would have been impossible.



# CONTENTS

<b>1. Introduction</b>	<b>1</b>
1.1. A Short History	2
1.1.1. The Document Routing Age	3
1.1.2. The People-Facing Age	3
1.1.3. The Dawn of Business Process Reengineering	3
1.1.4. The Production Age	4
1.1.5. The Automation Age	4
1.1.6. The Services Age	5
1.2. Application Development	5
1.2.1. Two-level programming	5
1.2.2. Recursive Composition	7
1.2.3. Web Services Application Structure	8
1.3. Software Stack	9
1.4. Requirements	10
1.5. Engine Architecture	11
1.6. Basic Implementation Structures	12
1.7. Objectives	13
1.8. Approach	14
1.9. Achievements	15

1.10.	Key Contributions . . . . .	19
1.10.1.	Architecture Contributions . . . . .	19
1.10.2.	Flow Optimizer . . . . .	19
1.10.3.	Caching . . . . .	20
1.10.4.	Exploitation of Application Server Features . . . . .	21
1.10.5.	Exploitation of Database Features . . . . .	21
1.10.6.	Relation between Quality of Service Reduction and Performance Improvements . . . . .	22
1.11.	Structure of the Document . . . . .	22
<b>2.</b>	<b>Architectures, Benchmarks, Optimization</b>	<b>25</b>
2.1.	Benchmarks . . . . .	26
2.2.	Workflow Management Systems . . . . .	27
2.2.1.	Apache ODE . . . . .	27
2.2.2.	JBoss Community jBPM . . . . .	30
2.2.3.	ActiveEndpoints ActiveVOS . . . . .	31
2.2.4.	IBM Process Server . . . . .	33
2.3.	Performance Optimization Techniques . . . . .	34
2.3.1.	Intra Engine Binding . . . . .	35
2.3.2.	Service Request Caching . . . . .	35
2.3.3.	Audit Trail Granularity . . . . .	35
2.3.4.	Process Persistence . . . . .	36
2.3.5.	Message Validation . . . . .	36
2.3.6.	Data Bases and Tables Allocation . . . . .	36
<b>3.</b>	<b>Base Architecture</b>	<b>37</b>
3.1.	Basic Architecture . . . . .	38
3.1.1.	Administration Component . . . . .	39
3.1.2.	Runtime Component . . . . .	41
3.1.3.	Buildtime Component . . . . .	42
3.1.4.	Databases . . . . .	42
3.1.5.	System Profile . . . . .	42

3.2.	Infrastructure . . . . .	43
3.2.1.	Infrastructure Choice . . . . .	43
3.2.2.	Stateless Execution and Transaction Chaining . . . . .	43
3.2.3.	Hot Pooling . . . . .	45
3.3.	The Life of a Process and its Activities . . . . .	46
3.3.1.	Process Life Cycle . . . . .	47
3.3.2.	Activity Life Cycle . . . . .	48
3.4.	Basic Processing . . . . .	49
3.5.	Synchronous Request Processing . . . . .	51
3.6.	Service Invocation . . . . .	52
3.7.	Base Configuration . . . . .	54
3.7.1.	System Deployment Descriptor . . . . .	54
3.7.2.	Process Deployment Descriptor . . . . .	55
3.8.	Databases . . . . .	56
3.8.1.	System Database . . . . .	56
3.8.2.	Buildtime Database . . . . .	58
3.8.3.	Runtime Database . . . . .	64
3.9.	Navigation . . . . .	67
3.9.1.	Basic Navigator Structure . . . . .	68
3.9.2.	Types of Request Messages . . . . .	69
3.9.3.	Internal Request Processing . . . . .	70
3.9.4.	External Requests . . . . .	76
3.10.	Activity Processing . . . . .	80
3.10.1.	Receive Processing . . . . .	80
3.10.2.	Assign Processing . . . . .	81
3.10.3.	Time Dependent Processing . . . . .	84
3.11.	Service Invocation Processing . . . . .	85
3.12.	Object Identifier Generation . . . . .	85
3.13.	Application Reliability . . . . .	87
3.14.	Deployment of a Process . . . . .	88
3.15.	Architecture Verification . . . . .	89

<b>4. Performance Testing</b>	<b>91</b>
4.1. Performance-Relevant Factors	92
4.1.1. Process Model Factors	92
4.1.2. General Factors	93
4.2. Performance Criteria	94
4.3. Performance Benchmark	95
4.4. Benchmark Setup	97
4.4.1. Client Setup	97
4.4.2. Server Setup	98
4.5. Benchmark Limitations	99
4.6. Benchmark Execution	100
<b>5. Base Caching</b>	<b>101</b>
5.1. Model Cache	102
5.2. Instance Cache	105
5.3. System Deployment Descriptor Cache	108
<b>6. Transaction Flows</b>	<b>111</b>
6.1. Transaction Flow Types	112
6.1.1. Processing	113
6.1.2. Short Transaction Flow Type	114
6.1.3. Medium Transaction Flow Type	115
6.1.4. Long Transaction Flow Type	116
6.1.5. Ultimate Transaction Flow Type	117
6.2. Test Results	120
6.3. Execution Characteristics	121
6.3.1. Logical and Physical Execution	122
6.3.2. Architectural Changes	123
6.3.3. Processing Algorithm	124
<b>7. Flow Configuration</b>	<b>129</b>
7.1. Internal Processing	129
7.1.1. Intra Engine Binding	130

7.1.2.	Inline Invocation . . . . .	132
7.1.3.	Microflow . . . . .	135
7.1.4.	Correlation Caching . . . . .	136
7.2.	Databases . . . . .	139
7.3.	Referential Integrity . . . . .	139
7.4.	Primary Keys . . . . .	140
7.5.	Large Objects Usage . . . . .	141
7.6.	Correlation Handling . . . . .	142
7.7.	Instance Deletion Strategy . . . . .	148
7.7.1.	Completion Deletion . . . . .	149
7.7.2.	Deferred Deletion . . . . .	149
7.7.3.	Ongoing Deletion . . . . .	152
7.8.	Flow Compilation for Microflows . . . . .	153
7.9.	Performance Improvements . . . . .	156
7.10.	Statistics Manager . . . . .	157
<b>8.</b>	<b>Flow Optimization</b>	<b>161</b>
8.1.	Flow Optimizer . . . . .	162
8.2.	Flow Execution Plan . . . . .	162
8.3.	Transaction Flow Construction . . . . .	164
8.3.1.	Transactions Construction . . . . .	164
8.3.2.	Multi-Phase Activity Link Creation . . . . .	171
8.3.3.	Join Links Creation . . . . .	173
8.3.4.	Standard Link Creation . . . . .	175
8.3.5.	Join Condition Creation . . . . .	176
8.3.6.	Transaction Chaining . . . . .	176
8.3.7.	Deadlock Freeness Analysis . . . . .	177
8.4.	Navigation . . . . .	182
8.4.1.	Enhanced Navigation Architecture . . . . .	182
8.4.2.	Transaction Flow Navigator . . . . .	183
8.4.3.	Transaction Internal Processing . . . . .	185
8.5.	Variable Usage Optimization . . . . .	190
8.6.	Variable Load Optimization . . . . .	195

8.7.	Correlation Set Usage Optimization . . . . .	196
8.8.	XPath Processing Improvement . . . . .	198
8.9.	Intra Transaction Cache . . . . .	204
8.9.1.	Multiple Cache Execution . . . . .	208
8.9.2.	Single Cache Execution . . . . .	211
8.10.	Cache Persistence . . . . .	212
8.11.	Execution Linearization . . . . .	216
8.12.	Performance Improvements . . . . .	221
8.13.	Flow Execution Plan Management . . . . .	223
8.14.	Road Map . . . . .	223
8.14.1.	Cost-Based Transaction Flow Type Selection . . . . .	224
8.14.2.	Statistics-Supported Flow Optimization . . . . .	224
8.14.3.	Feature Completion . . . . .	225
8.14.4.	Dynamic Transaction Boundaries . . . . .	226
8.14.5.	Execution History Based Optimization . . . . .	226
8.14.6.	Application-Specific Optimizations . . . . .	226
8.14.7.	Business Goals Optimizations . . . . .	227
<b>9.</b>	<b>Infrastructure Specific Optimizations</b>	<b>229</b>
9.1.	System Optimizer . . . . .	230
9.2.	System Statistics Manager . . . . .	231
9.3.	IBM WebSphere . . . . .	232
9.3.1.	Connection Handling . . . . .	232
9.3.2.	Scheduler Table . . . . .	233
9.3.3.	JMS Message Engine . . . . .	236
9.3.4.	JVM Setting . . . . .	236
9.4.	IBM DB2 . . . . .	237
9.4.1.	Basic Configuration . . . . .	238
9.4.2.	Databases . . . . .	241
9.4.3.	Table Spaces . . . . .	242
9.4.4.	Bufferpools . . . . .	243
9.4.5.	Indices . . . . .	244
9.4.6.	Optimizing the Indices . . . . .	246

<b>10. Quality Reduction Optimizations</b>	<b>249</b>
10.1. Persistence Options for Messages . . . . .	250
10.2. Transaction-Less Execution . . . . .	251
10.3. Service Request Result Caching . . . . .	253
10.4. Memory-Only Execution . . . . .	255
<b>11. Feature Optimizations</b>	<b>257</b>
11.1. Audit Trail . . . . .	257
11.1.1. Audit Trail Table . . . . .	258
11.1.2. Audit Trail Manager Architecture . . . . .	259
11.1.3. Basic Audit Trail Control . . . . .	259
11.1.4. Batching Audit Trail Records . . . . .	261
11.1.5. Event Type Selection . . . . .	262
11.1.6. Context Based Selection . . . . .	262
11.1.7. Multiple Audit Trails . . . . .	263
11.1.8. Audit Trail Caching . . . . .	264
11.2. Process Queries . . . . .	265
11.3. Process Instance Monitor . . . . .	266
<b>12. Topologies</b>	<b>269</b>
12.1. Single Server Structure . . . . .	270
12.2. Two-Tier Structures . . . . .	270
12.2.1. Shared Database . . . . .	271
12.3. IBM WebSphere Cluster . . . . .	272
12.4. Advanced JMS Messaging . . . . .	273
12.5. Cluster-Level Caching . . . . .	274
12.5.1. Shared Intra Transaction Cache . . . . .	275
12.5.2. Node Spheres . . . . .	275
<b>13. Summary and Outlook</b>	<b>281</b>
13.1. Summary . . . . .	281
13.2. Outlook . . . . .	283

<b>A. Benchmark</b>	<b>285</b>
A.1. Structure . . . . .	285
A.2. Process A . . . . .	286
A.2.1. WSDL Definition . . . . .	287
A.2.2. BPEL Definition . . . . .	291
A.2.3. Process Deployment Descriptor . . . . .	297
A.2.4. Flow Execution Plan . . . . .	300
A.3. Called Processes . . . . .	311
<b>B. Implementation</b>	<b>315</b>
B.1. Implementation Scope . . . . .	315
B.2. Infrastructure Exploitation . . . . .	316
B.2.1. IBM WebSphere Setup . . . . .	316
B.3. Project Structure . . . . .	318
B.3.1. Administration . . . . .	318
B.3.2. Administration Interface . . . . .	320
B.3.3. Process Execution . . . . .	322
B.3.4. Shared . . . . .	323
B.3.5. Shared EJBs . . . . .	324
B.3.6. Statistics . . . . .	325
B.4. Development Environment . . . . .	326
<b>Bibliography</b>	<b>327</b>
<b>List of Figures</b>	<b>341</b>
<b>List of Tables</b>	<b>345</b>

# INTRODUCTION

Over the last three decades, workflow technology has evolved into a key technology that is part of the software stack, the service oriented architecture, and the foundation for modern applications. Workflow now plays in the same league as database management systems, message queuing systems, transaction monitors, or application servers. This mandates that state-of-the-art WfMSs provide the same level of execution characteristics, such as robustness, continuous operation, high performance, and scalability.

This chapter presents information that helps to understand the usage of workflow technology, the technologies it is built upon, the requirements it must address, the approach the SWoM has taken to address these requirements, and the results that have been achieved. In particular, the following items are addressed:

- A short history of workflow technology to help understand the requirements that a workflow management system must address for various application environments and demands.
- The way workflow has changed how new applications are constructed. Those applications are no longer monolithic from a programming per-

spective, but are built according to the two-level programming paradigm with the workflow language providing the programming-in-the-large aspect.

- The advent of service-oriented architectures and their incarnation in the form of Web Services. The result is a set of standards for the various building blocks within the Web Service space, including the workflow language standard.
- The operational requirements that a WfMS must implement as a key middleware component, such as reliability, availability, and serviceability.
- The performance objectives a WfMS must address so that the constructed workflow-based applications live up to the performance demands of mission-critical applications, such as the management of account information in a bank.
- The approach that has been taken to construct the SWoM, a state-of-the-art high-performance workflow management system.
- The results of benchmark tests that have been carried out to evaluate the individual performance improvements and to relate the achieved performance to the performance of other workflow management systems.

The chapter ends with an outline showing what can be found in each of the remaining chapters of the thesis.

## 1.1. A Short History

Despite the relatively short time of a little more than twenty years since workflow debuted, six distinct phases can be identified in the evolution of workflow management systems. Each phase brought up a new set of requirements that the workflow management systems needed to address.

### 1.1.1. The Document Routing Age

The late 1980s saw the advent of the first systems that helped people to manage and route documents and folders electronically. A document could be anything, a folder, a scanned image, or a text document. A user of the system had an electronic inbasket which served as a container for the documents a user had to work on. The potential flow of documents was prescribed in advance with routing conditions defined in terms of document content or document properties. The actual routing was then carried out using the content of those routing properties. Since in “paper factories” (banking, insurance) work mainly equates to processing documents, this type of processing was called *workflow* [Ley01].

### 1.1.2. The People-Facing Age

Users soon discovered that the handling of documents is quite often just one small part of the overall business process. Additional functions were required, such as the invocation of functions provided by the enterprises’ application system. Workflow management systems grew the simple in/out baskets into work lists that the user could tailor to its needs, and that provided additional functions to manage individual work items within work lists, such as prioritization or life-cycle management. In addition, each work item could be associated with an executable, that when selected, caused the executable to be carried out; the work list has evolved into a launchpad for executables [LR94].

### 1.1.3. The Dawn of Business Process Reengineering

The early 1990s saw the advent of Business Process Reengineering (BPR) whose goal was the significant improvement of the efficiency of enterprises through radical changes and streamlining the internal business processes[HC94]. Thus workflow management systems had to be improved to cope with the increased spectrum of processes and the desire to speed up business processes and reduce costs.

The workflow management systems addressed the speedup requirement with a set of new functions: parallelism in workflows, deadline processing, workflow monitoring, and the processing of activities without any human intervention. The cost reductions requirement was indirectly addressed through a set of functions that helped to better understand the execution characteristics of the processes, such as the auditing of significant events, providing the capability of analyzing the process behavior.

#### 1.1.4. The Production Age

By the mid 1990s , workflow-based applications had become the state-of-the-art application structure used for enterprise applications: business processes were implemented via a workflow management system; the actual business functions were implemented “traditionally” using TP-monitors, such as CICS (CICS) [IBM11a], or application servers.

Enterprises have now become as dependent upon WfMSs as they had become decades ago on middleware components, such as TP-monitors, or DBMSs. The term *production workflow* has been coined to indicate that the WfMS is driving operational aspects of an enterprise.

To play in this league, WfMSs had to implement the same quality of service that characterized TP-monitors and DBMSs, such as high/continuous availability, scalability, robustness, and performance.

#### 1.1.5. The Automation Age

As more and more enterprises were moving forward with workflow technology, integration of applications became more and more important. Different application types, such as legacy applications, or newly written applications made the integration in a business process more cumbersome not to mention the plethora of new invocation mechanisms, such as message queuing and publish/subscribe, which were appearing.

To cope with this conglomerate of application types and invocation mechanisms, workflows needed to be understood as a business oriented “logical units of work” (LUW). The support of those LUWs required the introduction of

advanced transaction management: forward recovery of workflows as well as workflow-based applications and backward recovery (spheres of atomicity and compensation).

### 1.1.6. The Services Age

In the new millennium, the service oriented architecture started changing the way applications were constructed. The two-level programming became even more prominent, as applications and functions were exhibiting their external behavior as services using Web Services Description Language (WSDL)[W3C]. The creation of an application became the scripting together of different services using an orchestration language, such as Business Process Execution Language for Web Services (WS-BPEL)[OAS07].

## 1.2. Application Development

Over the last twenty years, workflow technology has been the driving force for a new application style following the two-level programming style as introduced by DeRemer and Kuhn [dK76]. This new application style was introduced by Leymann and Roller [LR97] as *workflow-based applications*. In addition the same article proposed an appropriate development methodology, called *process-based case*, for the construction of this type of application starting from a high-level design down to the final implementation, putting it into production, and monitoring the deployed applications.

### 1.2.1. Two-level programming

As pointed out, these new applications were characterized by their two-level programming structure, as shown in Figure 1.1 (adapted from a figure in [Ley10]).

DeRemer and Kuhn [dK76] introduced the notion of programming-in-the-large versus programming-in-the-small. They stated that programming-in-the-large requires a special language that is quite distinct from the languages that are used for programming-in-the-small.

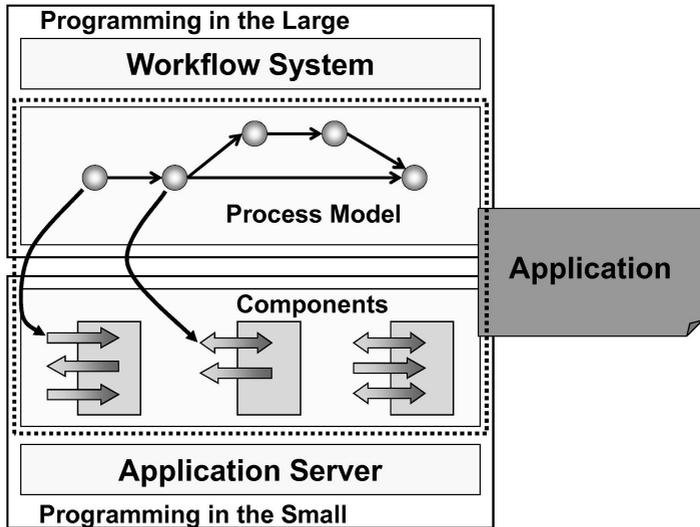


Figure 1.1.: Two-Level Programming

Workflow languages, such as the Flow Definition Language (FDL) used in IBM FlowMark[LR94], were exactly the language that provided the functions needed for programming-in-the-large. They were particularly interesting since they allowed to shift the boundary between programming-in-the-large and programming-in-the-small at will.

The figure illustrates the two level programming using the service oriented architecture, where the WfMS provides programming-in-the-large via the facilities to model and execute process models and the application server to execute software components that have been constructed using the standard programming-in-the-small Java programming language.

### 1.2.2. Recursive Composition

Another advantage of workflow-based applications is the recursive composition model that the WS-BPEL language provides. Figure 1.2, taken from [Ley10], shows that a process model not only invokes Web Services but represents its interfaces as Web Services.

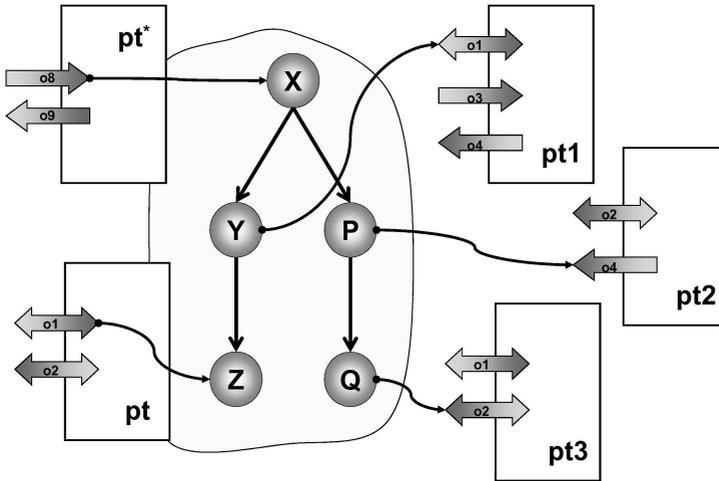


Figure 1.2.: Recursive Composition

This recursive composition model allows the assembly of large applications from components which are also workflow-based applications. As the WS-BPEL language contains specifications that are usually part of programming-in-the-small languages, such as if-then-else clauses or parallel case clauses, the boundary between programming-in-the-large and programming-in-the-small starts blurring, possibly moving the boundary further down to the programming-in-the-small area.

The net result of this change is the increase of resources that are consumed by the WfMS, which requires that the WfMSs increase their performance, that means carry out more process instances in less time. This is kind of a repetition

of the story that DBMSs went through after their initial success.

### 1.2.3. Web Services Application Structure

The application structure was further refined with the advent of Web Services. An application now consists, as shown in Figure 1.3, adapted from [Ley10], of three parts: the application structure is augmented with the application deployment descriptor.

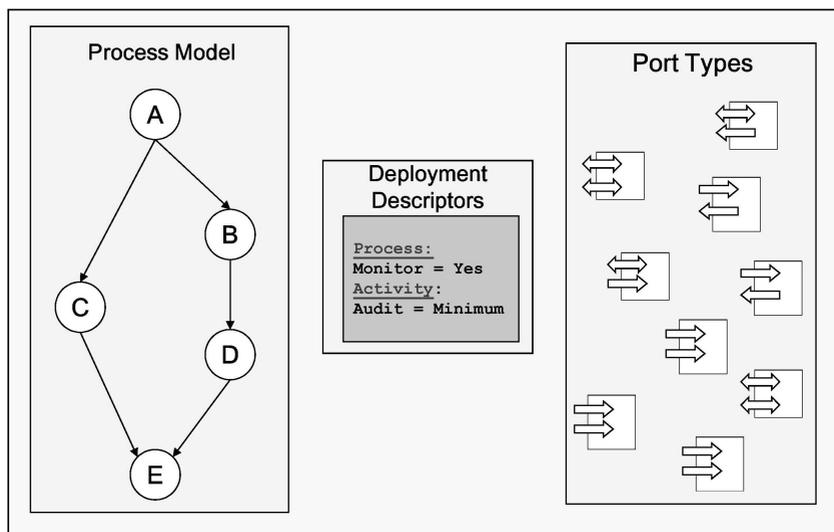


Figure 1.3.: Web Services Application Structure

The deployment descriptor provides additional information that is specific to the environment to which the application is deployed to as well as requirements that the application must deliver or meet, for example, whether an audit trail, which provides a history of the execution of a process instance, should be written for a particular activity, or whether additional information should be kept for a process instance for process monitoring purposes.

The deployment descriptor has not been standardized so far, and most likely, will not, since the information maintained in the deployment descriptor quite

often reflects the peculiarities of the workflow management system and the infrastructure the workflow management system is running in. It should be noted that the SWoM makes heavy use of the deployment descriptor: the *process deployment descriptor*.

### 1.3. Software Stack

Figure 1.4, a modernized version of a corresponding figure in [LR00c], shows the software stack that has become state-of-the-art. As usual, the base is the operating system. The database management system provides for the storage of data and is exploited by all layers above.

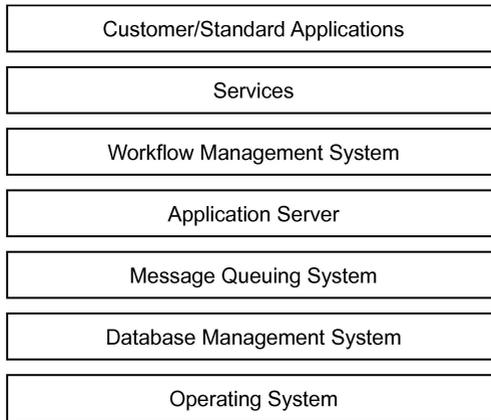


Figure 1.4.: Software Stack

The message queuing system on the next level provides enterprise-wide message queuing. Note that the standard communication support is provided by the operating system.

The application server provides a platform for applications through appropriate facilities such as transaction monitoring, component management, and request handling; it conceptually delivers the same type of functions for applications that the operating system provides on a lower level; one could perceive

the application server as an application operating system.

The workflow management system on the next level, running in the application server, is the foundation for applications that are constructed as Web Service applications as shown in Figure 1.3. Services that are used by the local workflow management system or by any other client via appropriate invocation mechanisms are also hosted on the application server.

On the top of the stack are the set of applications that users typically work with, either standard applications that software vendors deliver, such as book-keeping systems, or customer-specific applications that have been written to address customer specific business requirements.

## 1.4. Requirements

Leymann and Roller [LR00c] group the requirements on a production workflow system into two categories: operational requirements and enterprise requirements. Enterprise requirements, such as security, administration standards, and platforms, are of limited interest, so only operational requirements are presented.

1. **Reliability** A WfMS needs to be able to gracefully recover from failure and start where execution left off. After failures, completed work must not be lost and partial work must be undone and restarted.
2. **Availability** A WfMS must provide high availability. Sometimes it might even be required that it provides continuous operation, i.e. to still provide service when the system is upgraded or parts of it are down for maintenance.
3. **High Capacity** A WfMS needs to support many concurrent users and process instances at the same time. Ideally, the system itself does not impose a limit on those resources.
4. **Scalability** A scalable WfMS increases throughput (e.g. the number of requests satisfied) if additional resources are added. Scalability is a

property of the structure of the workflow system, allowing to flexibly add additional resources and leveraging them upon availability.

- 5. **Traceability** Traceability is a requirement of a WfMS to provide and persist information about the actions performed by the engine with regards to the processes executed, e.g. to monitor actions being performed on the level of the process. This can be later used for statistical analysis of completed process instances, for calculating costs related to the process, monitoring key performance indicators (KPI) [Par08], and checking if business-level goals are met.

### 1.5. Engine Architecture

When stripped of all peculiarities, one detects the basic engine structure of a WfMS in a SOA environment, shown in Figure 1.5. The core of the WfMS is the navigator that carries out a process instance according to the processing description in the process model. In a nutshell, it manages the life cycle of processes and activities, such as starting and terminating process instances, determines which paths within the process model have to be taken, and handles the activities.

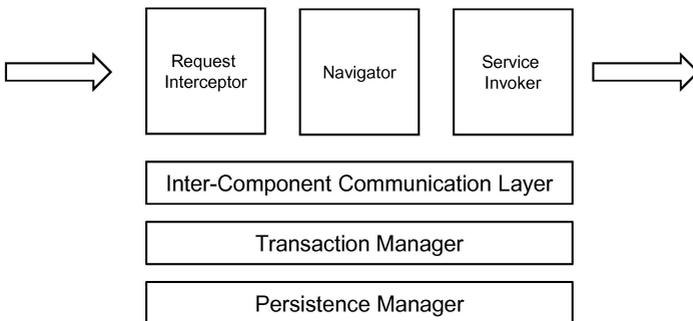


Figure 1.5.: Base Architecture

The request interceptor is the gateway between clients requesting the execution of the Web Service that a process represents and the navigator that carries out a process instance. It translates the external message into a format understandable by the navigator and supplements the message with information that the navigator needs to locate the associated process model and the activity which is the target of the request.

The invocation of Web Services that are associated with appropriate invoke activities in the process model are handled by the service invoker component, which takes the requests delivered by the navigator and invokes the specified Web Service.

The inter-component communication layer is the vehicle that the navigator, service invoker, and request interceptor use for communication. The actual implementation varies significantly between the different WfMSs; the SWoM uses a mixture of Java RMI calls and message queuing.

A WfMS can only address the previously specified requirements if the complete processing is done under transactional control; this mandates the use of a transaction manager that is able to handle transactions across multiple resource managers. This function may be provided by a separate transaction manager, the WfMS itself, or in the case of SWoM, by the JEE application server IBM WebSphere®.

Finally, a persistence manager is required that the WfMS uses for storing information between subsequent processing steps of a process instance. This function is typically provided by a relational database management system; SWoM uses IBM DB2® as the persistence mechanism.

Obviously, this engine architecture is not the only one that a WfMS can use. For example, the distributed engine proposed by [Wut10] uses Tuple Spaces [Gel85] for providing persistence, transaction, and communication support.

## 1.6. Basic Implementation Structures

Conceptually, one can differentiate between two WfMS architectures: non-distributed and distributed.

A non-distributed WfMS keeps all state information in a single location managed by the persistence layer of the WfMS, either on the same server as the WfMS or on a different one. The invocation of the specified Web Services is handled by the service invoker component of the WfMS, which may reside on the same server or even a different one. Depending on the location of the service invoker and the location of the Web Service, the appropriate requests may either be local or need to be carried out over the network. A WfMS in this sense, that runs on a cluster using a shared database, is still a non-distributed WfMS.

A distributed WfMS, in contrast, is made up of several autonomous workflow engines, each managing the state of process instances in its own persistence storage. A particular process instance is then carried out by several workflow engines, with the objective to carry it out as efficiently as possible. For example, an activity is carried out by the workflow engine that can handle the activity with the best performance and the least amount of resources required. Several implementations exist, such as OSIRIS [SWSS03, SWSS04], or the system proposed by Wuttke [Wut10].

A completely different approach for the distributed execution of a process instance is the splitting of a particular process model into a set of process models/process fragments. [Kha08, KL06] describe a method that creates the different process fragments and provides an appropriate execution environment. All approaches try to minimize the changes that are needed to the workflow management systems that participate in the execution of the different process fragments. In fact, the ideal situation would be one in which standard-compliant WfMSs are sufficient. Since only complete process models are processed by the individual WfMSs, these WfMSs can be distributed or non-distributed WfMSs. Information about several WfMS can be found in Section 2.2.

## 1.7. Objectives

The objective of the thesis is to develop a non-distributed WfMS that not only supports the requirements specified earlier but delivers the functionality with

good performance for all types of processes. This is achieved through a set of optimization techniques applied to a state-of-the-art WS-BPEL engine.

The main goal of optimization of the WfMS is for throughput rather than response time. The reasons for selecting throughput are manifold:

- If the WfMS executes parallel paths really in parallel, then the response time for the complete process mainly depends on the time that the invoked Web Services need for execution.
- Almost all optimization techniques, such as caching which reduces CPU cycles used by the WfMS and the DBMS, reduce the resource consumption of the WfMS, which improves throughput as well as response time.
- Optimization for response time calls for additional optimization techniques, such as improving the execution of parallel paths or the dynamic changing of the thread priority [LR05, LR06]. Most likely, these optimization techniques require support from the underlying infrastructure. For example, functions such like thread prioritization are not available in JEE application servers.

It should be noted that no optimization techniques have been developed that rely on extensions to the WS-BPEL specifications, such as special activity types [IBM05b, IBM05a].

The actual achievement of the objectives, the creation of a high-performance workflow management system, is measured using a benchmark that focuses on fast, asynchronous message exchange, parallel processing, and efficient data manipulation.

## 1.8. Approach

The development of the SWoM is carried out using a phased approach.

- The base engine is implemented using the system structure described in [LR00c], using the notion of a stateless server and exploiting model

and transaction level caching. The correctness of the design and implementation on IBM WebSphere and IBM DB2 is verified for scalability with respect to CPU exploitation and number of requests through the benchmark introduced later.

- The next step is the addition of configuration options that allows the process modeler to tell the workflow management system various properties of the process model so that the workflow management system can carry out the process model significantly more efficiently. In particular, the notion of transaction types and the support of correlation caching contribute to the performance improvements.
- The use of the process configuration options requires, at least for some of the options, a deep understanding of the execution of a business process. This problem is alleviated through the introduction of a flow optimizer. It not only automatically generates the correct process configuration settings but provides additional improvements, such as intra transaction caching, life cycle management for variables, or advanced persistence options.
- The performance improvements identified in the previous phases are applicable to all kinds of infrastructures. Other performance improvements exploit peculiarities of the underlying infrastructure, in particular the application server and the database management system.
- Finally, performance optimizations can also be applied to features, such as auditing, that are not required, for example, by the WS-BPEL standard, but are necessary for any complete WfMS implementation.

## 1.9. Achievements

The base version of the SWoM is first calibrated using the simple benchmark used in [BBD10a]. The calibration is necessary to have an understanding about the starting point of optimization. It is obviously significantly easier to achieve decent performance improvements when starting on a very low level.

The benchmark consists of a simple sequential process with five invoke activities; each invokes synchronously a simple Web Service that does nothing.

The published benchmark is run on six cores of a large server, equipped with four Two-Core Intel Xeon chips with 2.4 GHz, 32 GB memory and RAID disks and running Linux as the operating system. The same benchmark is run for the SWoM using a Quad Core Intel Chip with 3.0 GHz, 8 GB memory and running Windows 7 64 bit. The benchmark does not perform any significant I/O operations, so it can be assumed that the differences in the capabilities of the different I/O sub systems do not play a role. Both servers have enough main memory, so memory should not make a difference; only light paging was observed on the Windows server. It can therefore be assumed that CPU speed is the main differentiating factor. As the Linux server provides approximately twice the CPU power of the Windows server (based on the CPU performance benchmarks published by PassMark Software [Pas13]), the actual figures measured for the SWoM were adjusted by the factor 2.

<b>WfMS</b>	<b>Requests/sec</b>
ActiveVOS	20
jBPM	2
ODE	NRA
SWoM	27

Figure 1.6.: Calibration Performance Results

Figure 1.6 shows the appropriate benchmark results. The figures for the first three WfMSs (ActiveVOS, jBPM, Apache ODE) were published as response times; the SWoM results are throughput figures using soapUI [Sma12]. The following versions have been used : Active Endpoints ActiveVOS V 5.0.2 Server Edition, Apache ODE V 1.3.3. and V 2.0-beta2, and Red Hat JBoss jPBM-Bpel.

The benchmark collected for a set of parallel executing clients the average response time. As the benchmark was driving the CPU to saturation, the

response time numbers can be converted to throughput by dividing the number of clients through the average response time. It should be noted that no benchmark figures are available for Apache ODE, since the authors were unable to run the benchmark even with minimal work load.

The numbers given for the SWoM are conservative. The invoked Web Services in the original benchmark are implemented as servlets, whereas the Web Services in the SWoM case are standard WS-BPEL processes, consisting of a receive and a reply activity.

The benchmark is very simple; no XPath processing, no asynchronous message exchange, no parallel paths. A more complex benchmark, described in detail in Chapter 4, has been developed for testing the various optimization techniques introduced in the thesis. It consists of a main process that interacts with four Web Services which are implemented as business processes as well. It should be noted that the benchmark has also been used in testing the design of a multi-core workflow engine [PPBB13]. A summary of running the benchmark process with the different optimization techniques proposed in the thesis is shown in Figure 1.7.

	<b>Processes/ min</b>	<b>Total Processes/ min</b>	<b>Absolute Improve- ment (%)</b>	<b>Relative Improve- ment (%)</b>
Basic	876	4380		
Ultimate TFT	1100	5500	26.7	26.7
Configured	1195	5975	36.4	8.7
Optimized	1400	7000	59.8	17.1
Memory	1673	8365	91	19.5

Figure 1.7.: Performance Achievements for Benchmark Process

Each row in the figure represents the results obtained for a set of optimization techniques. The individual results are ordered by the efficiency of the optimization techniques with the first row, labelled `Basic` showing the results of running the SWoM unoptimized (the processing used for the number shown

in Figure 1.6). It should be noted that a particular set of optimization techniques includes the previous sets; for example, the `Optimized` set includes the optimization techniques used in the `Configured` and `Ultimate TFT` sets of optimization techniques.

The first column identifies the type of optimization; the second column shows the throughput that has been achieved in terms of processes/minute for the main process. The third column shows the total number of processes/minute that have been carried out: the main process and the four called processes. The fourth column shows the performance improvement achieved relative to the unoptimized execution shown in the first row. For example, the performance improvement of the set of optimization techniques, subsumed under `Optimized`, is 59.8% over the unoptimized version shown in the first row. The fifth column shows the relative performance improvements with aspect to the previous row.

As pointed out, the first row shows the number of processes that are carried out with the `SWoM` executing in base mode, where each activity is carried out as a separate transaction. The navigator and service invoker transactions are chained together using messages.

Row `Ultimate TFT` reflects the performance improvements that are achieved through running the benchmark with transaction flow type `ultimate`, which gives the most advanced transaction flow structure.

Row `Configured` shows the performance improvements that are achieved by having the process modeler specify process configuration options, such as the length of variables or the caching of correlation information. It should be noted that not all options have been used, such as intra engine binding.

Row `Optimized` shows the results of running the flow optimizer.

Row `Memory` provides the performance improvements that are achieved by carrying out the process instances in memory-only mode that means without any database accesses.

It is obvious that the same level of performance improvements can not be achieved for all types of process models. It can be expected, however, that most process models benefit from the set of performance improvements built into the `SWoM`.

## 1.10. Key Contributions

The SWoM advances the state of the art of workflow technology with a plethora of new architectural concepts and implementation techniques, which can be grouped into a set of categories:

### 1.10.1. Architecture Contributions

The architectural concept of transaction flows is the base for the execution and optimization of a workflow engine. A transaction flow consists of a set of transactions, whose execution sequence is described using the standard WS-BPEL flow and link constructs. Transactions consist of a set of activities that are carried out within a transaction. The structure of the transaction flow is either defined by the process modeler via appropriate configuration options or even automatically determined by a flow optimizer that uses additional statistical and cost information.

### 1.10.2. Flow Optimizer

- A flow optimizer that generates flow execution plans that assist the engine components to optimally carry out process instances. The flow optimizer uses statistical information collected by the various caches and statistics managers in addition to information supplied by the process modeler. Appropriate mechanisms, including versioning, are put in place to make sure that the flow execution plan is re-generated when the processing characteristics of a process model changes.
- The minimization of the SQL calls for retrieving variables from the persistence store by having the flow optimizer analyze the life cycle for variables via data flow analysis.
- Variables and correlation sets need to be stored in SQL data types CLOB/BLOB, as the size of variables and correlation sets is normally not defined. IBM DB2 handles LOB data types less efficiently than other basic types, such a CHAR or VARCHAR. The flow optimizer uses statistical

information about the length distribution to select appropriate database structures to eliminate the usage of LOB data types as much as possible.

- XPath is extensively used in transition conditions, assign activities, and correlation processing. The standard approach of using the appropriate Java implementation is not very efficient. New technologies are developed that replace the XPath processing with simple String operations as much as possible.
- The process instance is typically stored in the database in a de-composed form, each object type in its own table, for example, activity instances in the activity instance table. The SWoM introduces a new persistence structure by persisting the transaction-level cache as a CLOB within the database. This approach provides significant performance improvements if the user does not need to query for fields of the individual object types.

### 1.10.3. Caching

- The notion of a transaction-level cache that manages all process instance data within a particular transaction. The cache is cleared at the beginning of the transaction and persisted at the end of the transaction using the SQL batch update facility significantly reducing the number of SQL calls needed for updating the process instance in the database.
- A system properties cache manager so that all components that need system property information can obtain up-to-date information. Appropriate caching by the individual components is facilitated by the system properties cache manager providing an indicator when one of the system properties has changed.
- The notion of an intra transaction cache that keeps process instances between two subsequent processing steps of a particular process instance. This is particularly efficient for fast interactions between a process instance and the invoked Web Services as it eliminates the retrieval of

the information from the runtime database and the conversion into the memory representation.

#### 1.10.4. Exploitation of Application Server Features

A set of optimization techniques that improve the processing of the SWoM with respect to the functions provided by IBMWS. The techniques are relying on the SWoM statistics manager and the SWoM system manager that collect appropriate information from SWoM internal processing and IBM WebSphere.

- The optimization of connection handling through appropriate definition for the properties of the connection pool and prepared statement cache.
- The usage of multiple scheduler tables to optimize the processing of time-related WS-BPEL activities.
- The selection of the proper JMS messaging implementation.
- The tuning of the settings of the IBM WebSphere JVM.

#### 1.10.5. Exploitation of Database Features

A set of optimization techniques that improve the processing of the SWoM with respect to IBM DB2 optimizations. The techniques are relying on the SWoM statistics manager and a set of tools provided by IBM DB2.

- The placement of the different pieces of system data on different disk drives.
- The proper setting of the configuration parameters that control the execution of IBM DB2. The SWoM assists in the correct setting by collecting appropriate information as input to the DB2 Configuration Advisor.
- The proper support for IBM DB2's query optimizer by determining when new statistics information should be generated for the query optimizer

- The proper placement of the databases and their tables, bufferpools, and table spaces.
- The optimization of indices by having the SWoM collect information for the DB2 Design Advisor tool that determines the optimal indices.

#### 1.10.6. Relation between Quality of Service Reduction and Performance Improvements

A set of optimization techniques that improve performance by reducing the quality characteristics of the SWoM. In a worst case scenario, the process instance is not carried out at all or hangs around in some undefined state. Appropriate utilities are supplied to determine at least the ones which hang around and have them removed.

- The usage of non-persistent queues for communication between the different components of the SWoM.
- The execution of the SWoM in non-transactional mode.
- The memory only execution of a process instance.
- The caching of Web Service requests and memorizing the returned result for some time, so that subsequent requests can be serviced out of the cache eliminating the need to carry out the actual Web Service request.

#### 1.11. Structure of the Document

The remaining chapters present related work, the benchmark that has been developed to test the impact of different optimization techniques, and the different optimizations techniques with their resulting performance improvements.

**Chapter 2 Architectures, Benchmarks, Optimization** presents three areas that are connected to the thesis: the architectures of workflow management systems, workflow specific benchmarks, and optimization techniques. It should

be noted right away that the information is rather limited; in particular, the vendors of commercial systems provide minimal, if any, information.

**Chapter 3 Base Architecture** presents the base architecture of the SWoM. This includes the structure of the SWoM components, the deployment into the infrastructure, and the storage of information in databases.

**Chapter 4 Performance Testing** discusses performance objectives and presents a simple, yet expressive benchmark that helps to measure the performance of the basic functions of a workflow management system, such as handling the receive, invoke, assign, and reply activities and the related XPath processing.

**Chapter 5 Base Caching** presents how the SWoM caches basic information, such as process model information, process instance data, and system information.

**Chapter 6 Transaction Flows** represents the transaction execution structure that the SWoM follows when navigating through a process instance. Each transaction consists of one or more activities, based on the transaction flow type, which controls the granularity of transactions. The proper selection by the process modeler results in significant performance improvements.

**Chapter 7 Flow Configuration** introduces a set of configuration properties that can be set by the process modeler and if set properly result in major performance improvements.

**Chapter 8 Flow Optimization** represents the main chapter of the thesis. It introduces the notion of a flow optimizer that performs an analysis of the process model and uses information collected by a statistics manager to create a flow execution plan that the navigator uses for optimally processing process instances that are created from the process model.

**Chapter 9 Infrastructure Specific Optimizations** discusses optimization techniques that are specific to the infrastructure the SWoM is running in. In particular, it presents techniques for improving the resource usage of IBM WebSphere and IBM DB2.

**Chapter 10 Quality Reduction Optimizations** introduces performance improvement techniques that improve performance by lowering service quality, for example through the usage of transient queues or caching service requests.

**Chapter 11 Feature Optimizations** presents the optimization of a set of

features that are not mandated by the WS-BPEL specifications but that no workflow management system can live without. The most prominent one is the support of an audit trail to which all execution relevant information is written.

**Chapter 12 Topologies** provides an overview of the possible topologies into which the SWoM as a non-distributed workflow management system can be installed and discusses the performance improvement that can be obtained.

**Chapter 13 Summary and Outlook** discusses the achieved results and outlines future work.

**Appendix 1 Benchmark** provides detailed information about the process models that make up the benchmark, including appropriate WSDL and WS-BPEL definitions, process deployment descriptors, and flow execution plans.

**Appendix 2 Implementation** provides information about the SWoM implementation including code sizes.

# ARCHITECTURES, BENCHMARKS, OPTIMIZATION

Three aspects need to be looked at in the context of the thesis: (1) the types of benchmarks and benchmark results available, (2) architecture of other workflow management systems and their performance, and (3) performance optimization techniques implemented by other workflow management systems. It should be noted beforehand that the information is fairly scarce for a number of reasons:

- The structure of commercial systems can only be deduced by reading the appropriate product documentation ( which is quite often silent about technical details).
- Only limited performance information is available for both commercial and non-commercial systems. It can be assumed that the vendors of commercial systems have sufficient performance data available; however, the information is not publicly available and is used most likely only in customer engagements. Furthermore, benchmarks with commercial

systems can only be carried out with some consent of the vendor to avoid any unpleasant legal implications. For non-commercial systems (publicly available) one can only speculate that those systems have not matured to the point where performance figures could be obtained or any decent ones ever will.

## 2.1. Benchmarks

The number of benchmarks for workflow management systems is rather low, only lately some new developments have emerged. One could speculate that the vendors of commercial systems are running internal benchmarks to come up with performance figures that they publish in their product documentation or provide to prospective customers upon special requests. However, none of them has published the process models that they are using.

The first benchmark, called LabFlow-1, [BSR95] that has been published is studying the impact of the database on the workflow performance; in fact, the authors see it as a database benchmark rather than a workflow benchmark, comparing different database management systems. The benchmark was motivated by the statements made in [GHS95] that commercial workflow management systems can not support applications with high-throughput workflows and can not meet the needs of the associated genome research center for high-performance workflows. Unfortunately, the structure of the workflows is not given, so that no judgment can be made how the benchmark would perform on a state-of-the-art workflow management system.

The next benchmark [GMW00], conducted in 2000 by the database group of Gerhard Weikum at the University of Saarland, was comparing the performance of a commercial workflow management system with the one that has been developed by members of the group. The base was a rather simple e-commerce workflow that was described using state charts. The actual benchmark measured the throughput of each of the systems as well as the impact of the database work that was forwarded to an extra server. The maximum throughput of the systems was measured at 400 processes/hour running on a dedicated SUN Sparc 10.

SOABench[BBD10b] is a framework for the automatic generation, execution and analysis of testbeds for evaluating the performance of service-oriented middleware. It has been used in [BBD10a] to compare the performance of several WS-BPEL engines. The goal of the benchmark was to test the efficiency of the individual structural activities, such as <sequence>, <flow> with and without links, and <while>. Each activity type was used in a process model to run five invoke activities, whose implementation was a simple servlet. ActiveVOS [Act09], jBPM [JBo11], and Apache ODE [Apa11] were compared using a different number of clients and different think times between subsequent requests. The benchmark was carried out on a virtual machine with six dedicated CPU cores running at 2.4 GHz; memory was 34 GB. ActiveVOS was leading the pack with around 20 requests/second, followed by jBPM with around 3 requests/second. Apache ODE failed to complete even the lightest workload, which was 10 clients issuing parallel requests (see Section 1.9 for details).

## 2.2. Workflow Management Systems

This section presents the architecture and system structure of four WfMSs to the extent that the information can be obtained from external documentation, that means without going through an installation, monitoring the system execution, or even inspecting the code (which is not an option for commercial systems).

The WfMSs are the three ones that were used in the calibration benchmark: Apache ODE, jBPM, and ActiveVOS, plus IBM WebSphere Process Server.

### 2.2.1. Apache ODE

Apache Orchestration Director Engine (Apache ODE) [Apa11] is an open-source WS-BPEL engine hosted and developed under the umbrella of the Apache Software Foundation. A commercial variant is available under the name Intalio|Server [Int11] from the company that originally contributed the ODE code base to the Apache foundation.

Figure 2.1, published by Apache ODE [Apa12] and slightly adopted, illustrates the architecture of Apache ODE. It is composed of four distinct parts: (i) the process deployment module shown in the upper part of the figure, (ii) the ODE integration layer for interaction with the outside world on the right-hand side, (iii) the persistence layer depicted on the left-hand side and (iv) the ODE BPEL BPEL Runtime.

The deployment of a process is handled by the ODE BPEL Compiler which accepts a process model and compiles it into a form that can be handled by the ODE BPEL Runtime.

The ODE BPEL Runtime provides for the execution of the compiled BPEL processes by providing implementations of the various WS-BPEL constructs. The runtime also implements the logic necessary to determine when a new instance should be created, and to which instance an incoming message should be delivered.

ODE does not specifically require an application server to run; all services it depends on, such as a Java Transaction API (JTA) implementation, are provided by the engine. It provides communication channels for the process engine, such as handling incoming and outgoing Web service requests. Current implementations provide SOAP/HTTP connectivity directly through Apache Axis2 [The06] or indirectly through a Java Business Integration (JBI) [Sun05] implementation that implements this binding.

ODE Data Access Objects (DAO) abstract from a concrete database management system by providing an object model of the data to be persisted. Currently, there are two implementations of ODE DAOs: one for Hibernate3 [JBo12], a popular object-relational mapper, and the other for Open JPA [The12], Apache's JPA compliant Java persistence solution.

Instead of relying on a separate message-oriented middleware system, Apache ODE has implemented its own MOM-like system called Java Concurrent Objects (JACOB). The main goal of JACOB is to provide a persistent and asynchronous programming model that avoids the need to block execution while waiting for operations to complete. It furthermore allows to transactionally persist execution state for robustness and recovery. JACOB provides the following functions:

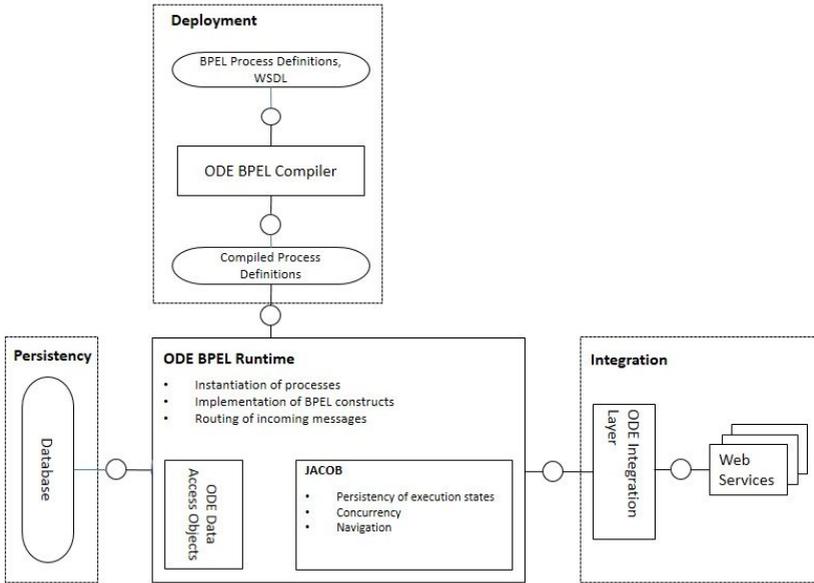


Figure 2.1.: ODE Architecture

- Provides a persistent and asynchronous programming model for application-level concurrency. It does not make use of operating system threads (which are claimed to be expensive); it provides its own task abstraction.
- Provides persistence of execution state via DAOs.
- Manages tasks in an execution queue which represents the current state of execution. Tasks may create other tasks, which are then appended to the execution queue.
- Supports communication between tasks through channels. Note that these channels are not queues in the sense of MOM; they are implemented as a Java class, sending a message means invoking a method on that class.

### 2.2.2. JBoss Community jBPM

jBPM is a WfMS that, in contrast to most other WfMSs, can carry out process instances of different process meta models by mapping a particular process meta model, such as WS-BPEL, to a generic process meta model. Figure 2.2 shows how this process meta model looks; Process Virtual Machine (PVM) [BF07] provides the appropriate execution environment.

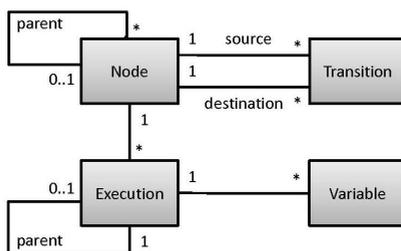


Figure 2.2.: PVM Meta Model

An activity is represented in the PVM meta model as a node. A node can be related to another node in two different ways. The first relationship establishes via transitions control flow dependencies between predecessor or successor nodes; the second relationship allows the specification of parent-child relationships.

Executions represent instances of process models at execution time. If a process is instantiated, an execution object is created that corresponds to the start activity. An execution implements the method *proceed* which transfers control (via a synchronous call) to the next activity.

The data that is generated in the course of the processing of a process instance is stored in PVM instance variables, that are assigned to executions. *Actions* realize application logic that is carried out at specific exit points of an execution, such as the entering of a transition into a node, the exit of a node, or even the transition itself.

For processes that specify parallel paths, a separate execution is maintained

for each parallel path. The relationship between the node, that realizes the fork [RHAM06], and the executions that are started is expressed via a parent-relation between the different execution objects. The fork node remains inactive during the execution of the different parallel executions. After the appropriate join has been evaluated (assuming that all enclosed nodes have executed without error), the fork node is reactivated and its execution is continued.

If processing of the application logic associated with the node or the transitions themselves is CPU-intensive or requires significant time, the transfer of control is not practicable; PVM solves this problem through *asynchronous continuations*. They are based on the transaction controlled insertion of a job request into a message queue [BHL95] and the asynchronous processing of the request through so-called *job executors*.

### 2.2.3. ActiveEndpoints ActiveVOS

ActiveVOS from ActiveEndpoints is a standard WfMS that provides the typical features, such as tools for process modeling, process simulation, and appropriate run time support. In addition to WS-BPEL, ActiveVOS supports the WS-BPEL Extension for People [AAD<sup>+</sup>07a] and WS-Human Task [AAD<sup>+</sup>07b]; an appropriate framework extends the support of additional query languages in addition to the ones coming with the system, which are XPath 1.0 [Cda99], XPath 2.0 [BBC<sup>+</sup>07] and JavaScript [E. 99].

Figure 2.3 shows the overall architecture of ActiveVOS, obtained from [Act09]. It is built on top of an application server that provides the platform for the actual WS-BPEL run time as well as the Web Service interfaces for the administration, task management, and monitoring components.

The ActiveVOS BPEL Engine implements the core of ActiveVOS; it provides the necessary functions for deployment and execution of processes. Persistence is supported through an abstraction layer of managers, so that different technologies can be exploited. Other components centered around the core handle the processing of expression, the addressing of the Web Service that the process interacts with, and the actual processing of the interactions themselves.

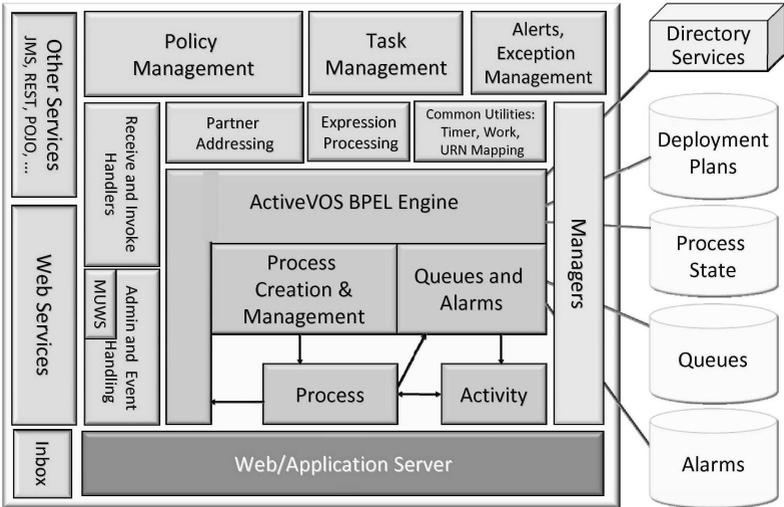


Figure 2.3.: ActiveVOS Architecture

Additional components implement task or inbox management, functions that are needed for support of WS-BPEL Extensions for People. The Exception Management component helps to correct errors that occur during process execution.

The deployment of process is carried out by first combining all relevant information, such as the process, the WSDL of the invoked Web Services, and a deployment descriptor, into a deployment container, and second, by importing the deployment container into the ActiveVOS environment. ActiveVOS uses, as do for example Apache ODE or SWoM, a deployment descriptor to specify process specific information, such as partner bindings or the persistence characteristics of the process instances. During deployment, each BPEL process model is transformed into an XML-DOM representation, which is traversed during process instance execution using the visitor pattern [GHJV95].

For all messages that create a process instance from the process model, a receive handler is registered with the external communication interface of

the WfMS. If a client now sends such a message to the WfMS, the receive handler intercepts the message and starts a new process instance. The instance generation is handled by an instance-specific implementation object, which is generated during process deployment. During the process instance generation, an implementation object is created for each of the definition objects.

Navigation is realized through an instance-specific navigation queue, called the execution queue. Activities are entered into the navigation queue by the parent activities as soon as all pre-conditions for the activity have been met. A little example, whose primary activity is a SEQUENCE, helps to illustrate the navigation. After the process instance has been generated successfully, the implementation object for the activity is inserted into the navigation queue. The navigator consumes the object and runs the execute method of the object. This method implements the application logic of the WS-BPEL activity. The logic in the case of a SEQUENCE activity is the insertion of the first child activity and the SEQUENCE activity into the navigation. After the child activity has completed processing, it signals its completion to the SEQUENCE activity. If the SEQUENCE activity is now carried out again, it determine if another child activity needs to be carried out and if so puts it into the navigation queue.

#### 2.2.4. IBM Process Server

IBM Process Server implements the architecture shown in Figure 2.4 (adopted from [FLO8]), which is mainly defined by three layers: the infrastructure layer, the runtime layer, and the function layer.

The infrastructure provides the basic functionality for the layers above: persistence delivered via a database, such as IBM DB2 and messaging based on IBM WebSphere Platform Messaging.

The runtime layer provides basic JEE application server capabilities such as JMS, Web Services, JEE container, and JDBC database access through WebSphere application server. The SCA runtime provides IBM's implementation of the SCA standard [Ope].

The Business Process Choreographer is the execution container for business processes (using the SCA WS-BPEL implementation model) and human tasks

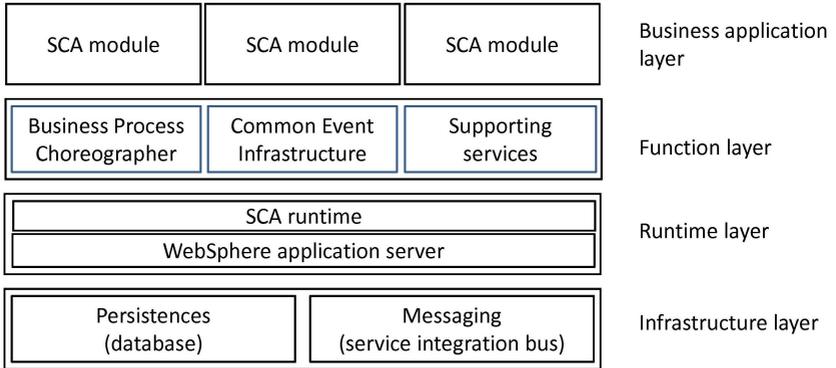


Figure 2.4.: WebSphere Process Server Architecture

(using WS-BPEL for people). It should be noted that most SCA modules make use of WS-BPEL as a result of the business centric view of SOA, and thus get in touch with Business Process Choreographer. A set of additional functions is provided in the function layer, that can be leveraged by integration developers, such as business rules. The Common Event Infrastructure (CEI) is a framework for logging business events, and auditing/monitoring the execution of SCA components (and even business processes).

It seems that no explicit information has been published by IBM about the internal working of Business Process Choreographer. However, given the additional information provided in [FLF09] about the usage of the infrastructure layer, namely database and messaging, it can be safely assumed that IBM Process Server uses the same or at least similar navigation techniques as the SWoM, both based on [LR00c].

### 2.3. Performance Optimization Techniques

Several commercial and non-commercial workflow management systems offer performance optimization techniques, either by supplying information on how to set up the system properly or by providing configuration parameters

for tailoring the system or individual processes. It should be noted in passing that none of these optimization techniques, which are part of the SWoM implementation, have been activated in the calibration test (see Section 1.9).

### 2.3.1. Intra Engine Binding

Intra Engine Binding, introduced in Section 7.1.1, eliminates the need to go through the SOAP stack if the target Web Service is also maintained by the SWoM and both processes execute in the same IBM WebSphere application server or cluster.

Oracle BPEL Process Manager [ORA08] provides a parameter `optSoapShortcut` that controls the bypassing of SOAP processing for local Web Service calls. The parameter applies to the complete process model, which somewhat limits its applicability; the SWoM provides for configuration each activity individually. Furthermore, the actual implementation has not been disclosed nor are any performance comparison figures available.

### 2.3.2. Service Request Caching

The response time of a business process as well as the throughput can be improved through the caching of service request results in the service invoker. [SHLP05] proposes such a cache for the enterprise service bus (ESB)[Cha04]; an appropriate cache mediation pattern for service invocation is presented in [RFT<sup>+</sup>05]. Xue [Xue10] shows how the dynamic cache of IBM WebSphere can be exploited for service request caching in IBM Process Server.

### 2.3.3. Audit Trail Granularity

Most workflow management systems offer simple options for specifying the amount of events and data that is written to the audit trail. Oracle BPEL Process Manager [ORA11] allows the process modeler to specify several options that control the amount of data that is written. The approach is slightly different from the basic audit control used by the SWoM; the Oracle BPEL Process Manager controls the amount of data that is written with each event, whereas

the SWoM controls which events are written. Furthermore, the SWoM provides additional fine-grained control capabilities, such as context-dependent audit trailing.

#### 2.3.4. Process Persistence

ActiveVOS provides a configuration option to disable process persistence completely to improve performance [Act11]. This seems to be equivalent to running the SWoM in memory-only execution mode (see Section 10.4).

#### 2.3.5. Message Validation

ActiveVOS provides a configuration option to disable the validation of input and output messages to improve performance [Act11]. No information is obtainable how invalid messages are being handled.

#### 2.3.6. Data Bases and Tables Allocation

Commercial systems typically provide information on how to set up the databases and their tables so the optimal exploitation is achieved; for example, IBM Process Server provides an appropriate manual [IBM05c]. SWoM takes this step further by collecting statistical information that can be used by appropriate IBM DB2 tools to optimize not only the configuration of the databases but also of IBM DB2 itself.

## BASE ARCHITECTURE

This chapter presents the base architecture, design, and implementation of the SWoM. It provides the necessary robustness, performance, and scalability that a middleware component must deliver. The chapter discusses, in particular, the following aspects:

- The *basic architecture* of the SWoM and the components that make up the SWoM, such as the administration, buildtime, and runtime components and their respective sub-components.
- The deployment of the SWoM into an appropriate execution environment that is delivered by IBM WebSphere and IBM DB2.
- The *life cycle* of the different objects of a business process, such as the process itself, activities, and variables.
- The *basic processing* of the sub-components of the *runtime* component, such as navigator, service invoker, and a set of service components that provide information to the runtime component and its sub-components.
- The structure and contents of the *Buildtime Database* which holds all information for process models and its parts, the associated process

deployment descriptor information as well as the information of the Web Services that are invoked by the process or that the process model represents.

- The structure and contents of the *Runtime Database* that holds the information for all process instances that are currently being carried out or for process instances that have been completed; however, the appropriate information has intentionally been left in the database (for example, to support process queries for some time after the process instance has completed).
- The concept of *navigation* and how the SWoM uses the Buildtime Database as well as the Runtime Database for carrying out business processes.
- The *processing of assign activities* to illustrate the data manipulation via appropriate XPath expressions.
- The *processing* that is carried out when *Web Services* are *invoked*.
- The details of the *deployment of processes*, which includes the activities needed to make a particular process model ready for the creation of appropriate process instances.

The presented information provides the base for understanding the architecture of the SWoM as a high-performance WfMS and the additional architectural decisions that are presented in the following chapters.

### 3.1. Basic Architecture

Figure 3.1 shows the basic architecture of SWoM with its five major pieces: the administration component, the buildtime component, the runtime component, a set of databases, and the system profile. It follows to a large extent the architecture presented in [LR00c].

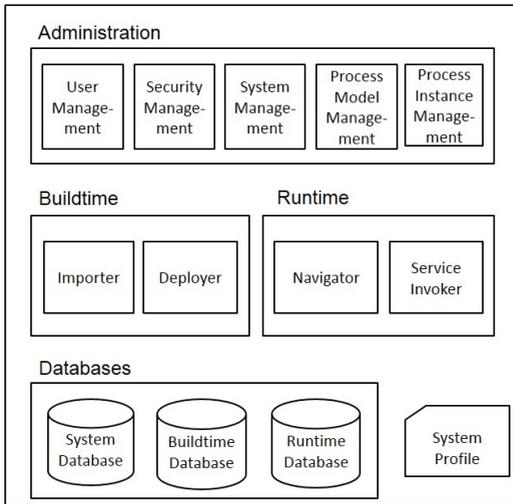


Figure 3.1.: Basic Architecture of SWoM

### 3.1.1. Administration Component

The administration component consists of five major sub-components: the user administration component that manages users and the relationships between users, the security management component that maintains the access rights of users to process models and process instances, the system management component that assists system administrators in controlling the execution of the WfMS, the process model management component that assists in controlling process models, and the process instance management component that helps in controlling process instances. Only the last component is important with respect to the performance of the SWoM, so that only it warrants a more detailed description.

### 3.1.1.1. Process Instance Management

The process instance management component is made up of four smaller components, each of them addressing a particular category for managing business processes.

- *Process Management* provides the user with the capabilities to control the execution of process instances.
- *Process Queries* provide the user with the capabilities to query the status of a process instance or a set of process instances.
- *Process Repair* is the set of tools that help to correct situations where a process instance is being carried out incorrectly or has entered a state that should never happen.
- *Process History Maintenance* provides the necessary instruments to manage the audit information that the SWoM generates when carrying out process instances.

### 3.1.1.2. Process Management

The category *process management* includes all functions that help manage individual or even a set of process instances. The major functions are the termination of process instances as well as suspending and resuming of process instances.

Termination means killing the process instance by aborting all running activities and, when completed, stopping the process.

Suspending a process instance is the stopping of the execution of a process instance; that means letting all running activities finish and then stopping navigation through the process instance. The process instance remains stopped until either the suspend time supplied with the suspend function has expired or an explicit resume function has been issued against the suspended process instance.

### 3.1.1.3. Process Queries

The category *process queries* includes all functions that are provided to locate particular processes and query their current state and processing history.

Locating a particular process or a set of processes that have the same property is a common task in WfMSs. For each request, the user specifies selection criteria which are then used to filter out the appropriate processes. There are two types of properties which the user can use as selection criteria: operational properties, such as the state or start date of a process, and business properties, such as the name of the customer.

Queries that deal with the operational properties are typically used by process or system administrators to monitor the SWoM or repair processes that have not been carried out correctly.

Queries that deal with the business properties are typically used by call center people or customers using a Web browser, for example. Those properties are managed by the WfMS in modeler-specified key data containers that are associated with the individual process models.

Both types of queries return the appropriate process identifiers, which allow the user to query the details of a process instance. Appropriate functions return the current state of the various constructs of the associated process model, such as activities or variables.

### 3.1.2. Runtime Component

The *runtime component* is SWoM's main component. It manages the life cycle of process instances, including the creation, the navigation through, and finally the termination of process instances. The runtime component consists of the *navigator* that navigates through the process graph and the *service invoker* that invokes the Web Services that are defined in a process model. A detailed discussion of the runtime component parts, and in particular their execution characteristics, is provided later in Section 3.4.

### 3.1.3. Buildtime Component

The *buildtime component* is responsible for importing process models and any associated information and making the process models available for the runtime component. A detailed discussion of the buildtime component, in particular the structure of the buildtime database is provided in Section 3.8.2

### 3.1.4. Databases

The SWoM maintains three databases for its processing:

The *system database* contains all information that the SWoM needs for its own operation, such as the settings of different execution options provided via the system deployment descriptor. In addition, the system database contains tables that hold error information that the SWoM generates; this information can be viewed by process administrators via the administration interface to take appropriate corrective actions.

The *buildtime database* contains all information about process models, the WSDLs, and the associated process deployment descriptors. The information is stored in several tables, with each table containing information about a particular process model element, such as the activity table for activities.

The *runtime database* contains all information about all process instances that are active (or kept around for a while if completed). The information is stored in several tables, conceptually mirroring the tables in the buildtime database.

### 3.1.5. System Profile

The system profile contains information for the different components, such as the names of databases; it is loaded by the various components when initialized by IBM WebSphere. The information is created by the installer during the installation of the SWoM.

## 3.2. Infrastructure

The system structure outlined so far can be mapped to several infrastructures that would provide the appropriate execution environment. The basic requirement for such an execution environment is that it (1) supports the execution of the different components, in particular the runtime environment, as transactions, (2) provides a persistent store for storing information between different processing steps of a process instance, and (3) provides for the robust communication between the different components. If an infrastructure meets these requirements, the appropriate robustness for mission-critical applications is achieved through proper implementation. The transactional execution provides the SWoM with forward recoverability [CARW97]; backward recoverability is provided by WS-BPEL by means of compensation spheres [KRL09].

### 3.2.1. Infrastructure Choice

We have decided to use a combination of a JEE application server and a relational database management system (RDBMS) as the infrastructure for the SWoM. The JEE application server provides support for the execution of the SWoM components, for transactions and messaging, whereas the RDBMS provides the necessary persistence support.

In particular, we have decided to use IBM WebSphere V 7.0 as the JEE application server and IBM DB2 V 9.7 as the RDBMS. We have implemented and tested SWoM on Windows 7; however, it can be expected, since both systems are available on a plethora of operating systems, that the SWoM can be ported without major difficulties to other operating systems. Furthermore, the port is made easy by strictly staying within the Java language, not using any operating system level facilities, and not exploiting IBM WebSphere proprietary extensions, such as asynchronous beans [Mis07].

### 3.2.2. Stateless Execution and Transaction Chaining

The processing of a process instance is the execution of a set of transactions that are chained together, where the output of a transaction is, as shown in

Figure 3.2, adopted from [LR00c], made persistent in a queue from which the next transaction reads it. The net result is that the complete process instance is carried out as a transaction, an approach that has been named *stratified transactions* [Ley97] [LR00b].

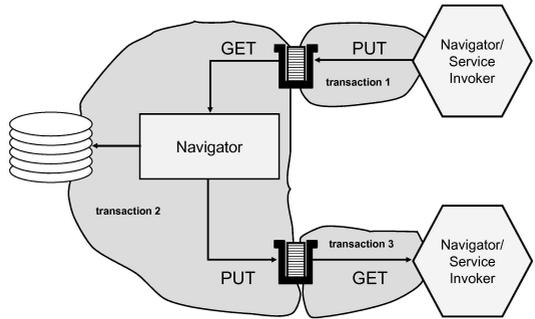


Figure 3.2.: Transaction Boundaries

All components, the navigator, the service invoker, and the façade beans (that represent the Web Services that the process model exhibits) are carrying out their processing as transactions and communicate with each other via messages that are put into the input queue of corresponding component. The different input queues form, as shown in [LR00c], a kind of software bus. Each component on the bus talks to another component on the bus by inserting a message using the input queue of the targeted component.

When a component receives a request from its input queue (or by being called directly as shown later), the request contains the identifier of the process instance that is processed; this allows the component to fetch the appropriate process instance from the database so that the request can be processed. The component carries out the necessary processing; upon completion it inserts an appropriate request message either into its own input queue or into the input queue of another component and updates the process instance state information in the runtime database. Since all information is stored in the runtime database, all components are stateless. For example, when the navigator receives a

message for the execution of an invoke activity, it fetches the appropriate process instance from the runtime database, creates an instance of the invoke activity, and executes the invoke activity instance. The invoke activity instance inserts a message for the service invoker into the service invoker's input queue requesting the invocation of the Web Service associated with the invoke activity. The navigator completes processing by writing the invoke activity instance information to the runtime database.

It is the combination of having stateless components and carrying them out as transactions that provides for the necessary robustness of the SWoM. If the transaction is aborted for whatever reason, it is just restarted providing the desired forward recoverability.

### 3.2.3. Hot Pooling

The implementation of the various components, such as the navigator or the service invoker, as message driven beans helps implement another important architectural concept, the concept of *hot pools* as shown on Figure 3.3 [LR00c]. Multiple instances of the same component are actively obtaining requests from the same input queue.

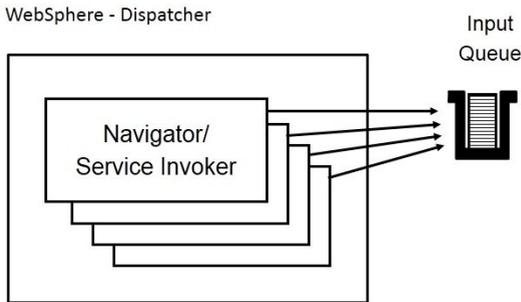


Figure 3.3.: Hot Pooling

Each of the active instances is waiting on the queue until a message is received, in which case it is processed, or until the instance is destroyed by IBM WebSphere. The number of active instances can be specified using IBM WebSphere settings. IBM WebSphere uses the concept of *session pooling* to manage bean instances. When a new instance is needed, it is activated from the session pool and when no longer needed, it is deactivated into the session pool. It is claimed that this approach is superior to creating and destroying session/message driven beans upon request. Incidentally, the same pooling is also supported for components invoked via a Web Service request or via an appropriate Java call, such as façade beans.

It should be noted in passing that multiple navigator instances may access the same process instance. This situation eventually occurs if a process has defined multiple paths. In this case, the execution of these navigator instances is serialized by having the navigator acquiring a lock on the process instance. This should not impose any problems on the throughput as navigator transactions are rather short-lived ( see the performance numbers given in Figure 1.7). In addition, the techniques described in Section 9.3 help adapting the number of active navigator instances to the actual demand.

### 3.3. The Life of a Process and its Activities

Each construct in a process model has its own life cycle; that means each object goes through a set of states. The WS-BPEL standard does not define the states of the different objects; it is left to the individual implementations which state model they want to implement and whether they would like to expose the appropriate interface or not.

The following sections illustrate the life cycles of process and activity instances; they follow to some extent the life cycle of the objects presented in [KKS<sup>+</sup>06].

### 3.3.1. Process Life Cycle

A process instance typically goes through many states and state transitions during its lifetime. Figure 3.4 shows the most important states. The transition from one state to another state is performed either by the SWoM or by some explicit request from an authorized user, such as the request to suspend the execution of a process instance.

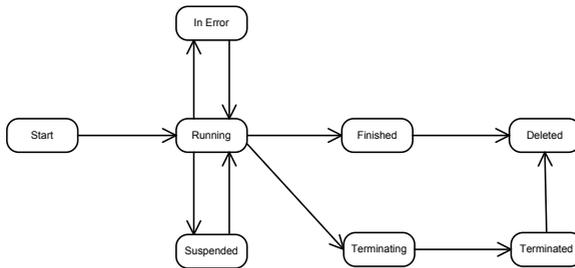


Figure 3.4.: Process Instance Life Cycle

The main state of a process instance is the *running* state. The process instance is put into this state upon arrival of a new request for the initial receive activity (note that each WS-BPEL process starts with at least one receive or pick activity). In the running state, the process is carried out, and the navigation through the process takes place. A process stays in this state until the process instance finishes, and then it goes into the *finished* state. The process instance stays in this state until either explicitly deleted by a user or after a user-specified time has expired, which results in the process instance being automatically deleted by the SWoM.

A process instance goes into the *suspended* state when a user requests this state explicitly. In this state, navigation has stopped, and no more activities are carried out. The process instance stays in that state until it is explicitly put back into the running state by a user's resume request or after the time specified with the suspend request has been exceeded. Any normal requests, such as completion of a Web Service, received during this time, are stored in

the runtime database. They are processed as soon as the process instance is resumed.

There are situations where further processing of a process instance is no longer meaningful. In this case, the process instance can be terminated by an authorized user. Termination causes all activities associated with the process instance, such as navigating through the process instance, to be stopped. Immediately after the request has been received, the process instance is put into the *terminating* state. After all activities have stopped, the process instance is put into the *terminated* state.

In certain situations, carrying out a process instance cannot be continued (even with appropriate fault handling defined in the process). In this case, the process instance is put into the *in Error* state. In this state, navigation through the process instance is stopped; an authorized user can perform appropriate recovery actions. After appropriate action, the process instance can be put back into the running state for continuation.

Note that the figure just reflects the major states and state transitions of a process instance. Many more states are available to cope with subtleties of process instance execution.

### 3.3.2. Activity Life Cycle

Activities also assume many different states and go through many state transitions. Figure 3.5 shows the main states that the activity can assume and the major transitions from state to state.

When a process is created, all activities are put into the *inactive* state. No actions are performed for activities that are in this state. However, they are shown in appropriate queries.

If an activity is not carried out because the appropriate path is not being taken or the join condition failed, the activity is skipped and goes into the *skipped* state.

When navigation reaches an activity, the activity is put into the *running* state, where it stays until the activity completes, either by entering the *finished* state after successful execution of the activity, the *terminated* state when the activity

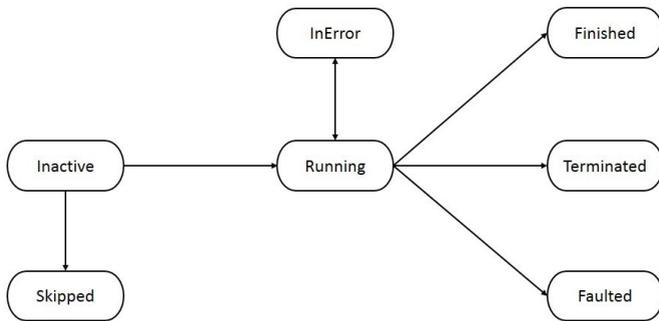


Figure 3.5.: Activity Life Cycle

was terminated by a termination request, the *faulted* state when the activity raised a fault.

The *error* state is entered when the activity terminates as the result of an error that can not be handled through the fault mechanism of WS-BPEL. Administrative commands are available for correcting the situation and bringing the activity back into the running state for continuing execution.

Note that the SWoM does not implement compensation, so the appropriate activity states are not listed.

### 3.4. Basic Processing

Figure 3.6 illustrates how the SWoM processes a request that comes from the outside; that means some client has issued a SOAP/HTTP request using one of the endpoints that is associated with a particular process model. Note that the request can also come from the SWoM itself when carrying out an invoke activity.

When a SOAP/HTTP request enters at the specified service endpoint, IBM WebSphere processes the message and hands it over to the respective façade bean. The façade bean is generated when a process model is deployed and im-

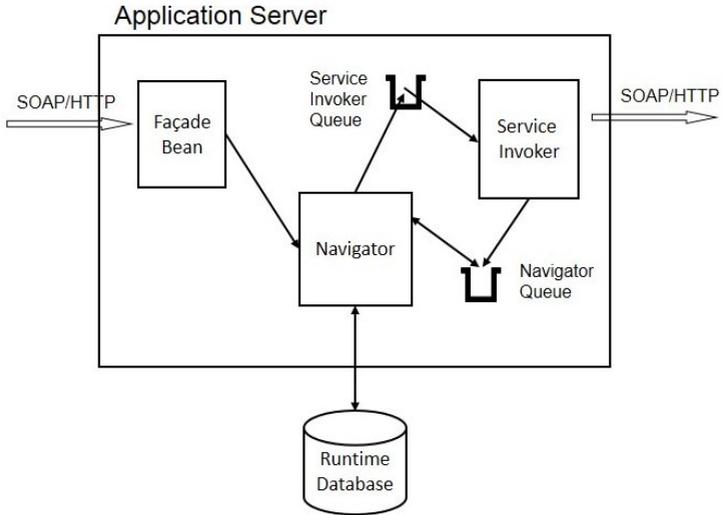


Figure 3.6.: Basic Processing Within The SWoM

plements all operations that the process model understands; more information about façade beans is provided in Section 3.14. The façade bean then invokes the navigator via the navigator façade interface. The navigator then performs the actual navigation through the process instance. To do this it fetches the appropriate process model from the buildtime database and process instance information from the runtime database. When the navigator locates an activity that invokes a Web Service, it prepares the appropriate invocation information and puts this information into the service invoker queue for processing by the service invoker.

An alternate approach of having the façade bean inserting a message into the navigator input queue has been discarded for a number of reasons:

- The solution requires an extra transaction (the façade bean transaction and the navigator transaction) and an extra message (the one that the façade bean puts into the navigator queue and that the navigator reads).

- The implementation of synchronous microflows (see Section 7.1.3) is significantly easier. Furthermore it is more robust since the microflow is carried out as a single transaction.

The service invoker carries out all service invocations. It uses the data supplied by the navigator to construct the message to be sent and to execute the appropriate call according to the binding specifications associated with the port type and the service definition.

After the service has been processed, navigation continues depending on the invocation type of the invoked Web Service, as explained in the following sections.

### 3.5. Synchronous Request Processing

The processing shown so far illustrates the processing of an asynchronous client request. Figure 3.7 explains the slightly different processing that is needed for synchronous client requests.

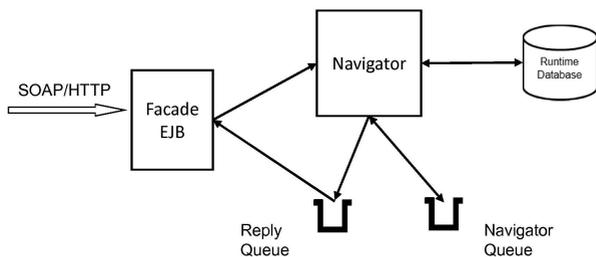


Figure 3.7.: Basic Synchronous Processing Within The SWoM

When the façade bean receives control, it invokes the Object Identifier (OID) generator component presented later (see Section 3.12) to obtain a unique OID, which it hands over to the navigator in the RMI navigator call and then waits on the navigator’s reply queue using the identifier as a JMS selector.

The navigator creates a new process instance and stores the received identifier together with the process instance information in the runtime database. The process instance is carried out as usual, possibly including several steps, requiring the usage of the runtime database and the navigator's input queue. When the navigator finishes the process instance, it inserts a message with the result and the identifier as selector into the reply queue.

Control is now returned to the waiting façade bean, which reads the message, constructs the appropriate SOAP message to be sent back, and returns control to the client.

### 3.6. Service Invocation

The service invoker, as mentioned earlier, is responsible for invoking the Web Service specified in an invoke activity. It needs to support synchronous and asynchronous invocation of a Web Service. Figure 3.8 shows the flow of control for the synchronous invocation of a Web Service.

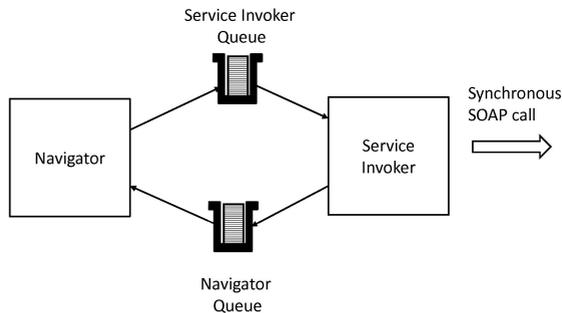


Figure 3.8.: Synchronous Invocation

The navigator inserts an appropriate request into the input queue of the service invoker and eventually finishes the current transaction. When the transaction is finished, the message is delivered to an instance of the service invoker. The service invoker extracts the information from the navigator message, creates the SOAP call, and then calls the Web Service. After completion,

it takes the response, maps it into a navigator message, inserts the message into the navigator input queue, and then finishes the transaction. Eventually the navigator reads the message, obtains the response and process instance information that has been exchanged between the navigator and the service invoker, and continues navigation through the associated process instance.

Figure 3.9 illustrates the processing of an asynchronous request. The SWoM internally differentiates for optimization reasons between two types of asynchronous processing: the one shown in Figure 3.9, which demonstrates the processing of a message exchange between an invoked Web Service and the invoking process instance, and the fire-and-forget type, where the Web Service is just invoked without calling back the invoking process instance.

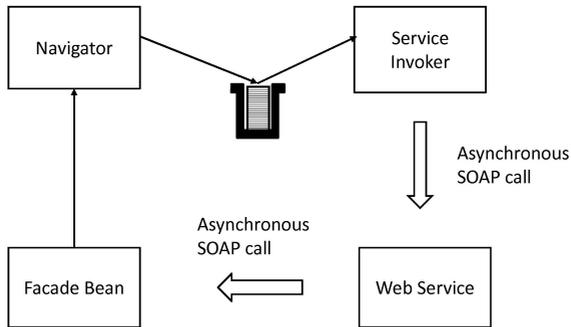


Figure 3.9.: Asynchronous Invocation

The navigator, as in the synchronous request, inserts an appropriate request into the input queue of the service invoker. The service invoker also constructs the SOAP call, invokes the Web Service, and then finishes the transaction. In the case of a fire-and-forget pattern, processing has completed. In a message exchange pattern, the invoked Web Service invokes the calling process instance again via another asynchronous SOAP call. If so, the façade bean gets control and then calls the navigator for continuing navigation of the process instance, identified via correlation information provided by the calling Web Service.

## 3.7. Base Configuration

The capability to adapt a piece of standardized software to the individual needs of the users of the software is usually called configuration or system configuration. In the case of the SWoM, configuration means adaptation of the SWoM itself and of the processes models.

The SWoM uses the notion of deployment descriptors to provide the appropriate information. The system-wide information for the SWoM is provided via the *System Deployment Descriptor* (SSDD); process-related information is defined via the *Process Deployment Descriptor* (SPDD). Both deployment descriptors use XML for easy readability. The SSDD is imported via the system management functions provided by the administration component. The SPDD is zipped together with the WS-BPEL file and the associated WSDL files into a SWoM Process Archive (SPAR) file and imported by the process model administration functions offered by the administration component.

### 3.7.1. System Deployment Descriptor

The SSDD provides the capability to overwrite the global processing default values of the major domains of the SWoM, such as administration, execution, debugging, IBM DB2 and IBM WebSphere exploitation, or caching strategies. Each SWoM domain is identified via a corresponding XML element. For example, the administration properties are identified via the `administrationOptions` element.

```
1      <?xml version="1.0" encoding="UTF-8"?>
2      <tns:SystemDeploymentDescriptor
3          xmlns="http://shared.swom.iaas/ssdd/"
4          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
5
6          <executionOptions>
7              <statisticsOptions>
8                  <sqlCallStatistics>YES
9                  </sqlCallStatistics>
10                 <transactionStatistics>NO
11                 </transactionStatistics>
12                 <messageStatistics>NO
13                 </messageStatistics>
14                 <variableStatistics>NO
```

```

14         </variableStatistics>
15         <correlationSetStatistics>NO
16         </correlationSetStatistics>
17         <responseTimeStatistics>NO
18         </responseTimeStatistics>
19         <executionTimeStatistics>NO
20         </executionTimeStatistics>
21         <soapCallStatistics>NO
22         </soapCallStatistics>
23     </statisticOptions>
24 </executionOptions>
25 </tns:SystemDeploymentDescriptor>

```

Listing 3.1: System Deployment Descriptor

Listing 3.1 demonstrates how a system administrator may define that the statistics manager of the SWoM keeps track of SQL call statistics. Line 7 shows the activation of the SQL statistics counter. These statistics, available via the administrative component, can be used by a system administrator to make appropriate adjustments either in the SWoM itself or in the underlying infrastructure components, such as the database. Furthermore, this information is used, as discussed later, by the flow optimizer to automatically adjust the SWoM processing or even dynamically modify the underlying infrastructure components.

Some of the information in the system deployment descriptor, for example, the `statisticOptions`, can be further overwritten via the process deployment descriptor, so that the settings only apply to the particular process model.

### 3.7.2. Process Deployment Descriptor

The SPDD supports the configuration of three different aspects of process models:

- The definition of endpoints for the Web Services that are invoked from the process. The process administrator can use this facility to either override the endpoint specified in the referenced WSDLs or provide endpoint references in the case that no endpoint reference could be obtained.

- The overwriting of SSDD properties for the processing of process instances of the associated process model, for example, by enabling SQL call statistics for the process model.
- The modification of properties of the process model, such as values in transition conditions. This allows process modelers to develop process models that can be dynamically modified.

Examples of process deployment descriptors are given later when discussing the different optimization techniques.

## 3.8. Databases

The SWoM maintains three databases: the system database, the buildtime database, and the runtime database. The databases are all normalized; each table has a primary key and uses referential integrity. LOB fields are typically used for WS-BPEL properties that are defined as `String`. The primary key is, apart from a few exceptions, always a `CHAR(12)` field containing an OID, which uniquely identifies each object in the databases. Whenever a component creates a new object, it calls the OID generator component (see Section 3.12 to generate an OID using information provided in the system database.

### 3.8.1. System Database

The system database holds all administrative information for the SWoM, such as:

- Information that the OID generator component needs to generate an OID when a component requests one. It holds the last issued OID in the table and fetches this one when new OIDs are requested and the internal batch of available OIDs has been used.
- Information used for the execution of the SWoM, such as the SSDD.
- User information, such as user name, password, and role that the user holds. This information is used by the SWoM to control access to the

administration functions. It should be noted that this information is normally stored in enterprise directories, such as an Lightweight Directory Access Protocol (LDAP) directory [LDA06].

- System activity information that is written when the SWoM encounters an error and that can be used by a system administrator to carry out appropriate corrective actions.

The most important table, the system information table, is shown in Listing 3.2.

```
1 CREATE TABLE SWOM.SYSTEM_INFO (
2     INFO_IDENTIFIER    INTEGER          NOT NULL
3     INFO_VALUE         CLOB (2000K)    NOT NULL
4 ) ;
```

Listing 3.2: System Information Table

The table holds all information that is required for managing the basic behavior of the SWoM. The system deployment descriptor, for example, is stored in this table using an information identifier of 1 as the key and the actual system deployment descriptor stored in the field `INFO_VALUE`. The information in the system information table is loaded whenever a component, such as the navigator, starts. Note that this is one of the few tables, whose primary key is not generated by the object identifier generator presented in Section 3.12.

Another important table is the system activity table, whose structure is shown in Listing 3.3.

```
1 CREATE TABLE SWOM.SYSTEM_ACTIVITY (
2     SYSTEM_ACTIVITY_ID CHAR(12)          NOT NULL
3     TIME               TIMESTAMP         NOT NULL,
4     SEVERITY           SMALLINT          NOT NULL,
5     AREA              SMALLINT          NOT NULL,
6     MESSAGE_ID        CHAR (8)          NOT NULL,
7     MESSAGE           VARCHAR (256) ,
8     PIIID             CHAR (12) ,
9     PMID             CHAR (12) ,
10    AIID              CHAR (12) ,
11    AID              CHAR (12) ,
12    MESSAGE           CLOB (2000k) ,
```

```
13         ABEND_INFORMATION      CLOB (2000k)
14     );
```

### Listing 3.3: System Activity Table

When a WfMS encounters a situation, either an internal processing error or an error that is the result of some user action, such as the submission of a request that can not be processed, one must make sure that sufficient information is available for a process administrator to take the appropriate corrective actions. A WfMS that does not collect this information can hardly claim that it supports mission-critical applications; just leaving the process instance hanging around in an *in Error* state or simply discarding a request is certainly not a viable option.

The SWoM solves this problem by writing all relevant information as an entry into the system activity table. The fields in Line 3 through Line 5 provide common information about the error, such as the time the error occurred, the area (navigator, service invoker, façade bean) that reported the error, and the severity of the error. The field `MESSAGE_ID` in Line 6 identifies the message that the SWoM issued; the field `MESSAGE` in Line 7 contains the appropriate message text. The fields in Line 8 through Line 9 identify the process instance and its associated process model; the fields in Line 10 and Line 11 identify the activity which is processed when the error occurs. The field `ABEND_INFORMATION` in Line 13 contains further information that helps the process administrator to diagnose the problem.

#### 3.8.2. Buildtime Database

The buildtime database contains two distinct, yet related sets of tables, one for holding process model information and one for holding the WSDL information of the Web Services that are provided/consumed by the processes. The information is put into the buildtime database by the importer shown in Figure 3.10. The importer accepts as input a SWoM Process Archive (SPAR) file. This file contains all information for a process model that the SWoM requires: the WS-BPEL process definition, the WSDL definitions for the process as well as the invoked Web Services, and the SPDD.

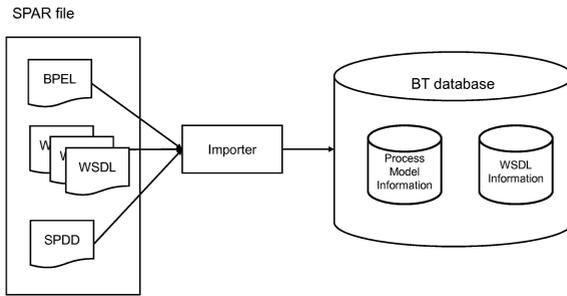


Figure 3.10.: Import Processing

The importer takes the information and stores it into the appropriate tables: the WS-BPEL information into the process model tables, the WSDL information into the WSDL tables, and the information contained in the Process Deployment Descriptor either into the process model or into the WSDL information. After the import has been completed, the two sets of tables contain all necessary information.

In the following sections only those constructs are presented, that are needed to understand how the navigator exploits the information stored in the build-time database and help comprehend the performance improvements.

### 3.8.2.1. Process Tables

Each WS-BPEL construct is held in an appropriate table: activities are stored in the activity table, the process model itself in the process model table. Each construct in a process model as well as the process model itself becomes a tuple in the appropriate table.

The anchor point for all process models is the *process model table*, whose most important fields are shown in Listing 3.4. Each process model is represented by a tuple in the table with the PMID field in Line 2 uniquely identifying the process model; the field is all keys generated during import.

```

1 CREATE TABLE PROCESS_MODEL (
2     PMID                CHAR (12)                NOT NULL
                        PRIMARY KEY ,

```

```

3      NAME          VARCHAR (255) ,
4      STATE        INTEGER          NOT NULL
5      ) ;

```

Listing 3.4: Process Model Table

The STATE holds the state of the process; it is used to control the handling of the process model, for example, whether the process model can be used to create process instances, or whether the process model still contains errors. Note that the size of the NAME field is an implementation restriction and does not comply with the WS-BPEL specification.

Each activity within a process model is reflected via an appropriate entry in the activity table shown in Listing 3.5, with the field AID uniquely identifying the activity.

```

1      CREATE TABLE ACTIVITY
2      (
3          AID          CHAR (12)          NOT NULL
4          INPUT_VID    CHAR (12) ,
5          OUTPUT_VID   CHAR (12) ,
6          NR_INCOMING_LINKS INTEGER          NOT NULL ,
7          PMID         CHAR (12)          NOT NULL
8          REFERENCES PROCESS_MODEL
           ON DELETE CASCADE
           ) ;

```

Listing 3.5: Activity Table

The PMID field points to the appropriate process model entry in the process model table. It is defined as a foreign key, so that all activities are deleted when the process model is deleted. The two fields INPUT\_VID and OUTPUT\_VID point to the appropriate input and output variable definitions associated with the activity; the variables are stored in the variable table. Obviously the fields are only meaningful for activities, such as receive or invoke activities, that have variables associated with them.

The NR\_INCOMING\_LINKS specifies how many links are entering the activity. The information is constructed from other information and is used here for better illustration of navigation (despite the fact that it helps improve performance as illustrated in Section 3.9.3.1).

The actual information about the links between the activities is stored in the link table shown in Listing 3.6. For each link an appropriate instance is created in the table. The source and the target activities of the link are maintained in the fields `SOURCE_ACT` and `TARGET_ACT` respectively; as usual `PMID` points to the process model the link is part of.

```

1 CREATE TABLE LINK
2 (
3     LID                CHAR (12)                NOT NULL
                        PRIMARY KEY ,
4     SOURCE_ACT        CHAR (12)                NOT NULL ,
5     TARGET_ACT        CHAR (12)                NOT NULL ,
6     TRANSITION_COND  CLOB (20K) ,
7     PMID              CHAR (12)                NOT NULL
                        REFERENCES PROCESS_MODEL
                        ON DELETE CASCADE
8 ) ;

```

Listing 3.6: Link Table

The associated transition condition is stored in the `TRANSITION_COND` field. It should be noted in this context, that the actual definition of transition conditions in the `WS-BPEL` specifications is not associated directly with the link definition itself. The `SWoM` extracts this information during import and attaches it directly to the link.

Listing 3.7 illustrates how the information about variables is stored. Each variable is uniquely identified via the `VID` key.

```

1 CREATE TABLE VARIABLE
2 (
3     VID                CHAR (12)                NOT NULL
                        PRIMARY KEY ,
4     SMID              CHAR (12)                NOT NULL ,
5     PMID              CHAR (12)                NOT NULL
                        REFERENCES PROCESS_MODEL
                        ON DELETE CASCADE
6 ) ;

```

Listing 3.7: Variable Table

The `SMID` field points to the appropriate schema definition for the variable. The schema definitions are managed as separate entities in a set of different tables.

Listing 3.8 illustrates how the information about an assign activity is stored. The AID is the appropriate unique identifier of the entry in the activity table; that means the assign table specializes the activity table. Incidentally, all other activity types, such as receive or wait, are also maintained in separate tables to store the data that is unique to the activity type.

```

1 CREATE TABLE ASSIGN
2 (
3     AID                CHAR (12)                NOT NULL
                        REFERENCES ACTIVITY
                        ON DELETE CASCADE ,
4     ORDER_NUMBER      INTEGER                    NOT NULL ,
5     LANGUAGE           INTEGER                    NOT NULL ,
6     FROM_SPEC         INTEGER                    NOT NULL ,
7     FROM_VID          CHAR (12) ,
8     FROM_PARM         CLOB (2000k)
9     TO_SPEC           INTEGER                    NOT NULL ,
10    TO_VID            CHAR (12) ,
11    TO_PARM           CLOB (2000k) ,
12    PRIMARY KEY (AID, ORDER_NUMBER)
13 ) ;

```

Listing 3.8: Assign Table

Each part of the assign activity is stored in a separate entry; the `ORDER_NUMBER` causes the different entries that make up a particular assign activity to be stored in the specified order. When the assign activity is later carried out, the different parts of the assign activity are processed in this order. The rest of the fields are used to represent the different options that WS-BPEL supports for the source as well as the target of the assign operations. The `LANGUAGE` field identifies the language that is used for selecting parts of the variables, such as `XPATH`.

The actual operations are specified via the `FROM_SPEC` and `TO_SPEC` field by identifying the type of data that is used, either as a source or as a target. For example, 0 indicates that the field is a variable, 2 that the field is a partner link. The actual values are stored in the `FROM_PARM` or `TO_PARM` fields or, if variables are referenced, in the `FROM_VID` and `TO_VID` fields.

### 3.8.2.2. Web Services Tables

All information about the invoked as well as provided Web Services is maintained in a set of tables that hold the appropriate WSDL. Listing 3.9 shows the anchor point for all WSDL information.

```
1 CREATE TABLE WSDL
2 (
3     WSDLID          CHAR (12)          NOT NULL
4                     PRIMARY KEY ,
5     TARGET_NAMESPACE VARCHAR (1024)    NOT NULL
6 ) ;
7
8 CREATE UNIQUE INDEX TNS ON WSDL
9     (TARGET_NAMESPACE
10    ) ;
```

Listing 3.9: WSDL Table

Each WSDL document is represented by an entry in the table. The WSDLID uniquely identifies the WSDL; it is generated by the SWoM during import and is used in foreign key references of the other tables that contain the rest of the WSDL information. The TARGET\_NAME\_SPACE field contains the namespace that is associated with the WSDL. Note that a particular WSDL document is not identified via a name but via the target namespace assigned to the WSDL, so an extra unique index on the target namespace has been added. The size of the target namespace is an implementation restriction.

Each construct within a WSDL document is stored in an appropriate table. Listing 3.10 illustrates how the table looks for port types. The NAME field contains the name of the port type.

```
1 CREATE TABLE PORT_TYPE
2 (
3     PTID          CHAR (12)          NOT NULL
4                     PRIMARY KEY ,
5     WSDLID       CHAR (12)          NOT NULL
6                     REFERENCES WSDL
7                     ON DELETE CASCADE ,
8     NAME         VARCHAR(255)        NOT NULL
9 ) ;
```

Listing 3.10: Port Type Table

The operations associated with the port types are held in the operation table shown in Listing 3.11.

```
1 CREATE TABLE OPERATION
2 (
3     OPID          CHAR(12)          NOT NULL
4     PTID          CHAR(12)          NOT NULL
5     NAME          VARCHAR(255)      NOT NULL
6 );
```

Listing 3.11: Operation Table

### 3.8.3. Runtime Database

All process instance information is kept in the runtime database; similar to the description of the buildtime database, only the most important tables with the most relevant fields are presented.

Conceptually, all information in the buildtime database is mirrored in the runtime database; that means that most of the tables in the buildtime database have a corresponding table in the runtime database. For example, the *process model* table in the buildtime database has an appropriate *process instance* table in the runtime database. Similar to the buildtime database, all keys are system-generated unique keys. The different tables are illustrated as being connected using appropriate referential integrity rules. This provides the capability to delete all parts of a particular process instance with a single SQL delete call. This simplifies the code for removing process instances and makes the code less sensitive against potential changes.

When a process instance is started, an entry is created in the *process instance* table shown in Listing 3.12. Each process instance is uniquely identified via the *PIID* field generated during process instance creation. The associated process model is identified via the *PMID*, which points to the appropriate process model in the buildtime database. The *START\_TIME* field contains the time when the process instance starts; the *STATE* field contains the current state of the process instance, such as running, terminated, or finished.

```

1 CREATE TABLE PROCESS_INSTANCE
2 (
3     PIID          CHAR (12)          NOT NULL
4                 PRIMARY KEY ,
5     PMID          CHAR (12)          NOT NULL ,
6     START_TIME    TIMESTAMP          NOT NULL ,
7     STATE         INTEGER            NOT NULL
8 ) ;

```

Listing 3.12: Process Instance Table

The process instance is created when an appropriate request is received via an appropriate `receive` or `pick` activity. The process instance is deleted when the process instance has completed. Whether the actual removal of the finished process instance takes place immediately after completion or some time later is defined through appropriate deletion strategies (see Section 7.7 for further information).

For each activity that is being carried out, an entry is created in the *activity instance table*. The `AIID` field uniquely identifies the entry. The foreign key `PIID` identifies the process instance that the activity instance belongs to; the appropriate delete rules cause the entry to be deleted when the process instance is deleted. The field `AID` points to the appropriate activity in the buildtime database.

```

1 CREATE TABLE ACTIVITY_INSTANCE
2 (
3     AIID          CHAR (12)          NOT NULL
4                 PRIMARY KEY ,
5     AID           CHAR (12)          NOT NULL ,
6     PIID          CHAR (12)          NOT NULL
7                 REFERENCES PROCESS_INSTANCE
8                 ON DELETE CASCADE,
9     ACT_NR_IN_LINKS INTEGER          NOT NULL ,
10    STATE         INTEGER            NOT NULL
11 ) ;
12
13 CREATE UNIQUE INDEX PIIDATIDINDEX ON ACTIVITY_INSTANCE
14 (PIID,AID) ;

```

Listing 3.13: Activity Instance Table

The STATE field contains the actual state of the activity instance such as running, finished, or terminated (see Section 3.3.2 for details about the life cycle of activities). The ACT\_NR\_IN\_LINKS field is used to count the number of links that have entered the activity. When the number of actual incoming links is equal to the number of defined incoming links (see the appropriate field NR\_INCOMING\_LINKS in the ACTIVITY table), navigation can continue; that means all links entering the activity have been evaluated and processing of the join condition associated with the activity can start.

The unique index on the combination of process instance identifier and activity identifier serves several purposes: First, it makes sure that only one activity instance is created for a particular activity within a process model. Second, it allows efficient access to a particular activity instance via those two fields. This access is, for example, needed for the parallel processing of activities with multiple incoming links. Third, it is used to efficiently delete all activity instances of a particular process instance when the process instance is deleted and the activity instances are deleted via the appropriate foreign key rules.

Evaluation of the join condition typically requires the truth values of transition conditions associated with incoming links. Thus one needs to have a table that reflects instances of links as shown in Listing 3.14.

```

1 CREATE TABLE LINK_INSTANCE
2 (
3     LIID          CHAR (12)          NOT NULL
4     PRIMARY KEY ,
5     LID           CHAR (12)          NOT NULL ,
6     PIID          CHAR (12)          NOT NULL
7     REFERENCES PROCESS_INSTANCE
8     ON DELETE CASCADE,
9     STATE         INTEGER            NOT NULL
10 ) ;
11
12 CREATE INDEX LIID_PIID_INDEX ON LINK_INSTANCE PIID ;

```

Listing 3.14: Link Instance Table

The LIID uniquely identifies the link instance. The LID field points to the appropriate definition of the link definition in the link table. The PIID field references the appropriate process instance.

Each variable is stored as an entry in the *variable instance table* shown in Listing 3.15. As a variable is unique within a process model, one can identify the variable via the appropriate VTID field that describes the variable and the PIID that identifies the associated process instance.

The index on the PIID is needed for the efficient deletion of variable instances when the associated process instance is deleted.

```
1 CREATE TABLE VARIABLE_INSTANCE
2 (
3     VIID          CHAR (12)          NOT NULL
4                 PRIMARY KEY ,
5     VMID          CHAR (12)          NOT NULL ,
6     PIID          CHAR (12)          NOT NULL
7                 REFERENCES PROCESS_INSTANCE
8                 ON DELETE CASCADE ,
9     VALUE         CLOB (2000k) ,
10    PRIMARY KEY (VIID)
11 );
12
13 CREATE INDEX VIID_PIID_INDEX ON VARIABLE_INSTANCE PIID ;
```

Listing 3.15: Variable Instance Table

The actual data is stored in the VALUE field. A LOB field type has been chosen since the size of the field is not restricted by the WS-BPEL specification. It is felt that it is not possible to inject an implementation restriction and use a VARCHAR field whose size in IBM DB2 V9 is restricted to 32704 bytes. The disadvantage of using LOBs is the fact that IBM DB2 stores a LOB in a separate file so that an extra I/O operation is needed to access the field. The optimization techniques introduced in Section 7.5 addresses this issue. The Stuttgarter Workflow Maschine stores variables in literal XML format, so a CLOB is used. Other storage possibilities, currently not exploited in SWoM, are serialized Java objects, JavaScript Object Notation (JSON) [JSO13] or DOM object representations.

### 3.9. Navigation

The tables shown in the two previous sections are used by the navigator to carry out a process instance. Processing consists of identifying the execution

path(s) of the process instance and directing the execution of the individual activities on that path. The sequence in which the activities have been carried out is called the *execution history* of the process instance.

The main processing that the navigator performs is very simple: after an activity has been completed, the next set of activities is determined and appropriate activity specific execution processing is initiated. These steps are initiated either by an internal request or an external request, where an internal request has been issued by another SWoM component and an external request has been issued by some source outside the SWoM.

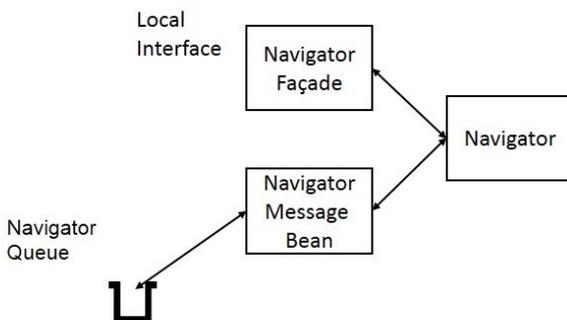


Figure 3.11.: Basic Navigator Structure

### 3.9.1. Basic Navigator Structure

The navigator as shown several times already is invoked in several styles, first by using a standard Java call and second by sending messages to it. To cope with this situation, the navigator implements the structure shown in Figure 3.11.

Both the navigator façade and the navigator message bean are implemented as EJB. The navigator façade is a stateless session bean which implements the navigator's interface; only a local interface needs to be implemented, since the façade beans and the navigator façade are running in the same application

server. A remote interface would only be needed if the various components are running in different application servers. The navigator message bean is a message driven bean that accepts input from the navigator input queue. Both EJBs load the actual navigator as a class during initialization and call this class for initialization.

### 3.9.2. Types of Request Messages

The processing of requests either delivered as a message or, as in the case of the façade bean, passed as a parameter is facilitated through a set of messages, which are all inheriting properties from the abstract message shown in Listing 3.16.

```
1 public abstract class AbstractMessage implements Serializable {  
  
2     private short messageDestination;  
3     protected short queuePersistence;  
4     private short messageType;  
5     private short messageRequestor;  
  
6 }
```

Listing 3.16: Abstract Message

`messageDestination` specifies the component that is the target of the message, `queuePersistence` whether the message should be processed as a persistent or non-persistent message (see Section 10.1 for details about the usage), `messageType` defines the type of message, and `messageRequestor` the creator of the message.

Two messages inherit from this abstract message: the *DeleteServerMessage*, which is sent to the delete server, and the *navigation message*, which is used by the navigator directly but also serves as the base for a number of other messages. The navigation message adds the following three fields: AID which identifies the activity to be processed, PMID which identifies the process model associated with the message, and the variable associated with the request. If available, the associated instances are identified via PIID identifying the process instance and AIID identifying the activity instances. Additional properties, that are specific to a particular processing, are reflected in the additional messages:

the *fromFacadeBeanMessage*, that the façade bean sends to the navigator façade, the *invocationMessage*, that the navigator sends to the service invoker, or the *fromServiceInvokerMessage*, that the service invoker sends back to the navigator.

### 3.9.3. Internal Request Processing

An internal request is processed in the following two cases: (1) the navigator sent itself an internal message requesting the processing of a new activity, and (2) the service invoker has completed processing of the associated Web Service and inserts an appropriate completion message into the navigator's input queue. The navigator determines the type of request by checking the message type in the message. If it is a *standard navigation message*, the navigator knows that it is a message it had sent itself; if it is a *fromServiceInvokerMessage*, it knows that the service invoker has sent the message after completing the synchronous invocation of a Web Service.

It should be noted that the code used for illustration is rather simplified; for example no checking is performed whether the process instance is in state terminating.

#### 3.9.3.1. Navigator Request

The navigation message identifies the activity that should be executed via the AID field and the associated process instance via the PIID field.

Listing 3.17 illustrates the processing that the navigator carries out to locate the appropriate activity, locates an existing activity instance if the activity is a join condition, creates if necessary an appropriate activity instance and determines if the activity instance can be processed.

Line 1 obtains, from the activity table, all necessary information about the activity to be processed; in particular it retrieves the counter that specifies the number of links that enter the activity. Note that this counter has been created when the activity information was stored in the activity table (see Listing 3.5; it eliminates the need to retrieve all links that enter the activity to determine the number of incoming links). Line 2 sets a switch that indicates whether the activity can be processed, to `false`. Line 3 determines whether the target

activity is a join activity (having multiple incoming links entering it) or not by checking the retrieved counter.

```
1  SELECT NR_INCOMING_LINKS
   INTO :nrInLinks
   FROM ACTIVITY
   WHERE AID = :targetActivityId
2  activityProcessable = false
3  if (nrInLinks > 1) then
4      SELECT ACT_NR_IN_LINKS FROM ACTIVITY_INSTANCE
   INTO :actNrInLinks
   WHERE (AID = :targetActivityId AND
   (PIID = :processInstanceId)
5      if notFound then
6          INSERT INTO ACTIVITY_INSTANCE
   (AIID,AID,ACT_NR_IN_LINKS)
   USING (:activityInstanceId,
   :activityId,1)
7      else
8          actNrInLinks = actNrInLinks + 1
9          if actNrInLinks < nrInLinks then
10             UPDATE ActivityInstance
   SET ACT_NR_IN_LINKS = :actNrInLinks
   WHERE (AIID = :activityInstanceId)
11             else
12                 activityProcessable = true
13             end if
14         end if
15     else
16         activityProcessable = TRUE
17     end if
18     if !activityProcessable then
19         return
20     end if
```

Listing 3.17: Navigation - Locate Activity Instance

If the number of incoming control connectors is greater than one, the activity is a join activity. In this case, the activity can not be selected for processing until all incoming links have been evaluated; the appropriate processing is carried between Line 4 and Line 11.

Line 4 checks if an instance of the target activity has already been created previously by accessing the activity instance table using the activity identifier and the process instance identifier. Such an instance of the target activity exists if any one of the other links entering the activity has been processed earlier.

If no activity instance exists (Line 5), Line 6 creates an activity instance (with the number of actual incoming links set to 1) and inserts it into the activity instance table. No further processing is needed in this situation as this is the first link that enters the activity and there is more than one link entering the activity.

If an instance exists already (Line 7), the processing between Line 8 and Line 12 is carried out. First, the number of links that have entered the activity is increased by one (Line 8). If the number of links that have entered the activity (Line 9) is still smaller than the total links that have been defined to enter the activity, the activity instance is updated in Line 10 with the incremented counter.

Line 11 is being carried when all links have entered the activity; processing of the activity is initiated by setting the program variable `activityProcessable` to true. Similarly, the program variable is set to true in Line 12 when the activity has maximal one incoming link.

Line 15 is processed when the activity has only one incoming link; the activity is then by definition processable and appropriately set in Line 16.

If the activity is not processable (Line 18), processing is terminated and the navigator completes processing the request. Otherwise the navigator processes the activity as detailed in Listing 3.18.

```
1      check join condition
2      if true then
3          set :state = running
4      else
5          if suppressJoinFailure then
6              set :state = skipped
7          else
8              set :state = faulted
9          end if
10     end if
11     if nrInLinks > 1 then
12         UPDATE ACTIVITY_INSTANCE
13             SET STATE = :state
14             WHERE (AIID = :activityInstanceId)
15     else
16         INSERT INTO ACTIVITY_INSTANCE
17             (AIID,AID,STATE)
18         USING (:activityInstanceId,
```

```

                                :activityId, :state)
15  end if
16  case (state)
17    when (running) then
18      process activity
19      return
20    end when
21    when (skipped)
22    when (faulted) then
23      start fault handling
24  end case

```

Listing 3.18: Navigation - Process Activity

After an activity is ready for processing, the join condition is evaluated (Line 1). Evaluation of the join condition typically includes checking of the truth values of other incoming links or even links that have been processed several activities before. If the join condition evaluates to true, the state of the activity is set to *running* (Line 2). If the join condition is false (Line 4), the `suppressJoinFailure` setting determines the state that the activity assumes and the actions that are taken. If set, the activity state is set to *skipped* (Line 5), otherwise the state is set to *faulted* (Line 7).

Line 11 and Line 13 update the activity instance with the actual state. If the activity has multiple incoming links, then the activity instance exists already and is updated via Line 11; otherwise an activity instance is created in Line 13.

Line 16 through Line 22 initiate the appropriate processing based on the state of the activity. If the activity is in state *running*, processing of the activity is started. If the activity is an invoke activity, an appropriate request is sent to the service invoker and the activity instance stays in state *running*. Activities that can be processed immediately, for example an assign activity, are carried out right away, and the activity instance goes into state *finished*.

Line 19 finally checks the activity state and, if the state is *running*, processing is complete and the navigator terminates the transaction.

If an activity has completed or is skipped, the navigator determines the next set of activities that need to be processed (or skipped) as shown in Listing 3.19.

```
1  DECLARE outgoingLinks AS CURSOR FOR
    SELECT TRANSITION_COND, TARGET_ACT FROM LINK_TEMPLATE
    WHERE (SOURCE_ACT = :aid)
2  OPEN outgoingLinks
3  FOR ALL outgoingLinks DO
4      FETCH outgoingLinks
        INTO :transitionCondition
            :targetActivityId
5      Evaluate transition condition and
        store state in :linkState
6      INSERT INTO LINK_INSTANCE
        (PIID, LIID, LID, STATE)
        VALUES (:piid, :liid, :lid, :linkState)
7      IF transitionCondition is true THEN
8          insert navigation message with TARGET_ACT as AID
9      ELSE
10         insert navigation message with TARGET_ACT as AID and skip indicator
11     END IF
12 END FOR
13 CLOSE outgoingLinks
```

Listing 3.19: Navigation - Determine Next Activities

Line 1 defines a cursor that, when executed, retrieves all links that leave the completed activity. The cursor is opened in Line 2. Line 3 starts a loop to retrieve all outgoing links, evaluate them, determine the target activities, and inserts an appropriate processing request into the navigator's input queue.

In particular the following actions are taken: Line 4 retrieves the next link that exits from the completed activity to obtain the transition condition and the activity identifier of the target activity. Line 5 evaluates the retrieved transition condition and puts the result in the `linkState` variable for later storage in the database. Line 6 inserts an appropriate link instance in the link instance table. Storing the information, in particular the state, provides the capability to reference the state information in other transition conditions using the appropriate WS-BPEL XPath functions.

If the transition condition evaluates to true (Line 7), an appropriate navigation message for processing the target activity is inserted into the navigator's input queue. It is processed later the same way the current message is pro-

cessed. If the transition condition evaluates to false, then the target activity must be skipped and an appropriate message is inserted. Note that the corresponding processing of skipped activities has been left out for simplicity. The navigator, after closing the cursor, finishes processing.

### 3.9.3.2. Service Invoker Request

When the service invoker inserts a message into the navigation queue, it populates the message with the information that the navigator provided, in particular the activity instance identifier (AIID) of the invoke activity and the process instance identifier (PIID). The navigator uses this information to locate the appropriate activity instance and change its state to *finished* as shown in Listing 3.20.

```
1  SELECT AID
   INTO :aid
   FROM ACTIVITY_INSTANCE
   WHERE AIID = :aiid

2  activityState = finished

3  UPDATE ACTIVITY_INSTANCE
4  SET STATE = :activityState
   WHERE AIID = :aiid
```

Listing 3.20: Navigation - Process Service Invoker Request

Line 1 determines the activity in the process model that is associated with the activity instance that just completed processing. This is done by accessing the activity instance table with the identifier of the activity instance (AIID) and retrieving the identifier of the activity template (AID). Line 3 updates the state of the activity instance to *finished*, which is set in Line 2.

The next step is to locate the next set of activities, as already shown in Listing 3.19 in Section 3.9.3.1.

### 3.9.4. External Requests

The navigator needs to process two types of external requests: requests sent by the timer service of IBM WebSphere and requests received for receive or pick activities through appropriate client calls.

The requests sent by the timer service are obtained from the navigator queue as any other request and are processed the same way as any other navigator internal request, since the request contains the identifier of the process instance that caused the timer to be set.

Requests received from a client fall into two distinct categories depending on the location of the receive/pick activity within the process model. If the receive activity starts the process, processing is simple: the navigator creates a process instance and then processes the receive activity. The situation is slightly more complicated for receive activities that are in the middle of the process model and mostly are intended to process the answer of an earlier invoke activity. The received message does not directly identify the process instance but contains correlation information that can be used to identify the target process instance; Section 3.9.4.2 explains the appropriate processing of the message.

#### 3.9.4.1. Response Processing

As pointed out earlier, each receive activity is complemented by a façade bean that is generated during process deployment and that serves as implementation of the Web Service that represents the receive activity. The façade bean is called by IBM WebSphere when a request is received on the appropriate Web Service endpoint. Listing 3.21 shows the processing that the façade bean carries out when a call is received. Note that this processing is also carried out for receive activities that start a process instance.

```
1   obtain input message from request
2   construct navigator interface parameters
      from input message, process model identifier,
      and activity identifier
3   Invoke navigator using the local Navigator interface
```

Listing 3.21: Façade Bean Processing

The façade bean's first action (Line 1) is the analysis of the request message and the extraction of relevant information from the SOAP body, in particular the variable that is supplied by the caller.

Next, the façade bean constructs (Line 2) the parameters for invoking the navigator. Of particular importance are the activity identifier of the activity associated with the request and the process model identifier. Both pieces of information have been generated into the façade bean by the SWoM deployment process. The navigator uses this information together with the correlation information in the supplied variable to locate the proper process and activity instance, or in the case of an initial receive activity, to create a process instance from the specified process model.

Finally (Line 3), the façade bean invokes the navigator façade using the standard Java call mechanism. The façade bean is defined with transactional properties; so a transaction is established which the navigator joins.

When the navigator receives control, it carries out the actions, shown in Listing 3.22.

```
1   obtain process, activity, and correlation set information
2   call Correlation Manager to obtain
3       process instance
4   locate activity instance via
5       activity identifier and
6       process instance identifier
6   process activity and carry out navigation
```

Listing 3.22: Service Request Handling

Line 1 obtains the information for the process, activity, and correlation set from the appropriate tables in the buildtime database using the supplied process model and activity identifiers that the passed message contains.

Line 2 calls the correlation manager to obtain the identifier of the appropriate process instance that is the target of the request. A detailed description of how the correlation manager locates the appropriate process instance can be found later in Section 3.9.4.2. Line 4 uses the obtained process instance identifier and the activity identifier to obtain the activity instance that is associated with the activity. The found activity instance is now processed as outlined in Section 3.9.3.

### 3.9.4.2. Correlation Processing

Correlation sets are the means to associate a received message with a particular process instance that generated those correlation sets and provided them when calling a Web Service [IBM11b]. Figure 3.12 illustrates how correlation information is managed.

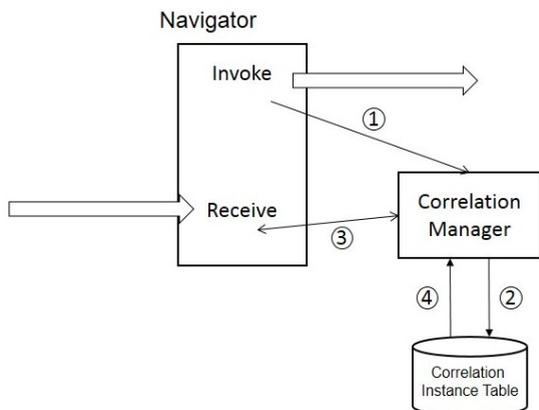


Figure 3.12.: Correlation Manager

When the navigator processes an `invoke` activity, which sets a correlation set, it invokes (1) the correlation manager to put an appropriate entry (2) into the correlation instance table whose structure is shown in Listing 3.23.

```
1 CREATE TABLE CORRELATION_SET_INSTANCE
2 (
3     CSIID          CHAR (12)          NOT NULL ,
4     CSID           CHAR (12)          NOT NULL ,
5     PIID           CHAR (12)          NOT NULL
6     VALUE          CLOB (2000k) ,
7     PRIMARY KEY (VALUE,CSID, PIID)
8 ) ;
9 CREATE UNIQUE INDEX ON CORRELATION_SET_INSTANCE
10 (VALUE,CSID) ;
```

Listing 3.23: Correlation Set Instance Table

The CSIID field uniquely identifies the correlation instance. The appropriate definitions of the correlation sets are found in the buildtime database via the CSID field.

The process instance associated with the entry is maintained in the PIID field; the actual value of the correlation set is maintained in the VALUE field.

The specified unique index ensures the WS-BPEL restriction that at any time only one value can be assigned to a correlation set in a process model. The primary key is used to access the table via the VALUE and CSID fields; the field PIID has been added so that IBM DB2 can obtain all information from the index and does not need to go to the data table itself. It should be noted that the table is a conceptual view; IBM DB2 does not support the indexing of LOB field types. One approach is to inject an implementation limitation restricting the size of correlation values to the maximum size support by the VARCHAR field type. If that is not an acceptable option, one needs to resort to the appropriate optimization techniques introduced in Section 7.5.

When the navigator processes a receive activity with an associated correlation set definition, it calls (3) the correlation manager to obtain the process instance identifier that is identified via the correlation set. The correlation manager looks up (4) the database and performs the processing shown in Listing 3.24. The calls are simplified to highlight the concepts behind correlation processing; for example, only one correlation set is associated with the message that the receive activity processes and no locking of the process instance is performed as is normally required.

```
1      SELECT PIID FROM
          CORRELATION_INSTANCE
          WHERE (VALUE = CorrelationValue AND
                CSID = CorrelationSetIdentifier)
2      if not found then
3          handle not found situation
4          exit
5      end if
6      obtain the process instance
```

Listing 3.24: Correlation Processing

Line 1 selects from the correlation set instance table the process instance identifier of the process instance that qualifies.

Line 2 handles the situation that no entry is found in the correlation set table; that means no process instance can be located that is the target of the external request. The WS-BPEL specifications do not specify which actions are to be taken, so a WfMS may just discard the message, an approach that is certainly not suitable for a piece of middleware; the SWoM stores the received message in the system activity table for later analysis and appropriate corrective actions.

Line 6 locates the process instance and obtains all necessary information.

### 3.10. Activity Processing

Each activity type is supported through an appropriate implementation, called an activity processing handler, that inherits from an interface that defines the requests that an activity needs to understand. For example, the `execute` method requests the execution of the activity; the `complete` method requests that the activity performs all processing after the activity has been carried out. This structure allows to fairly easily add new activity types to the SWoM.

#### 3.10.1. Receive Processing

The processing of a receive activity is carried out in two phase. The first phase, the creation phase, creates an instance of the receive activity in the runtime database; the second phase, the execution phase, carries out the actual processing of the message that is associated with the receive activity. Navigation of the process instance stops after the creation phase and the running transaction is committed. Processing of the execution phase is carried out in new transaction that is established when the message is received by the façade bean. In other words, the creation phase of a receive activity concludes a navigator transaction, the execution phase starts a new transaction.

### 3.10.2. Assign Processing

An important part of the work that is done in a process is the creation and manipulation of data, which is maintained in WS-BPEL variables. A variable is created either by receiving it as part of a message that is consumed by the process, or by creating it within an assign activity. The assign activity also provides the capability of modifying the contents of a variable. Therefore the majority of work being done with variables is in assign activities, the processing usually being quite resource-intensive, both from a database and from a CPU cycle perspective: database expensive because the variables have to be retrieved from the runtime database at the start of the assign activity and written back to the runtime database when the assign activity completes, CPU cycle expensive since it involves XPath processing of the variables.

#### 3.10.2.1. Basic Processing

Listing 3.25 illustrates the database processing that is carried out when an assign activity is processed.

Line 1 declares a cursor that is used to obtain the assign definitions, such as COPY, of the assign activity, that is processed (identified via its AID). The individual assign definitions are maintained in the database in the order in which they have been defined (see Listing 3.8). The cursor is opened in Line 2.

The code enclosed via Line 4 and Line 18 is repeated for each of the individual assign definitions; the appropriate loop is started in Line 3. The individual assign part is fetched in Line 4. For illustration purposes the listing is limited to the copying of a variable (or a part of it) to another variable (or parts thereof); however, the checking for the type of input and output specifications is included for completeness.

```
1  DECLARE selectAssignParts AS CURSOR FOR
   SELECT * FROM ASSIGN
   WHERE (AID = :aid)
   ORDER BY ORDER_NUMBER
2  OPEN selectAssignParts
3  FOR ALL assignParts DO
4      FETCH selectAssignParts INTO
       :fromSpec :toSpec
```

```

        :fromParm :toParm
        :fromVID :toVID
5      if :fromSpec = variable then
6        SELECT VALUE FROM VARIABLE_INSTANCE
          INTO :fromData
          WHERE (VID = :fromVariableVID AND
                PIID = :piid)
7      end if
8      if :toSpec = variable then
9        SELECT VALUE FROM VARIABLE_INSTANCE
          INTO :toData
          WHERE (VID = :toVariableVID AND
                PIID = :piid)
10     if SQL_NOT_FOUND then
11       toVariableExists = false
12     end if
13   end if
14   perform appropriate assign operation
15   if toVariableExists = false then
16     INSERT INTO VARIABLE_INSTANCE
       VTID, PIID, VALUE
       (:toVariableVTID, :piid , :toData)
17   else
18     UPDATE VARIABLE_INSTANCE
       SET VALUE = :toData
       WHERE (VTID = :toVariableVTID AND
             PIID = :ppi)
19   end if
20 end for
21 CLOSE selectAssignParts

```

Listing 3.25: Assign Activity Processing

If the from specification references a variable (Line 5), the appropriate variable instance is retrieved from the variable instance table (Line 6).

The same processing is carried out for the to specification (Line 8), if the to specification references a variable. Line 9 tries to obtain the target variable instance. If no variable instance is found, the programming variable toVariableExists is set appropriately (Line 10), which is used later to trigger the correct SQL processing.

Line 14 then performs the appropriate assign operation by copying, for example, appropriate information from the source variable to the target variable. Line 15 inserts the target variable into the database if the variable did not exist; Line 17 updates the target variable if the variable existed. Line 21 closes

the cursor after all assign parts of the assign activity have been processed successfully.

### 3.10.2.2. Assign Part Processing

Listing 3.26 shows the basic processing carried out when a part of a variable is copied to a part in another variable. The parts are specified by an appropriate XPath query (see the assign statement in the appropriate listing A.13 of the benchmark).

```
1      convert from variable to DOM tree
2      convert to variable to DOM tree
3      perform XPath query on from variable
4      perform XPath query on to variable
5      pupdate to variable query result with from variable query
6      transform to variable from DOM tree to literal XML
```

Listing 3.26: Assign Activity Part Processing

Line 1 converts the from variable, which is maintained in the database as an XML document, into an appropriate DOM representation. This transformation is normally not needed, as the XPath processor automatically converts an XML document into a DOM tree. However, if the same variable is used several times, then this approach provides better performance. Furthermore, the line is shown to indicate what is really going on, even if it runs under the cover. Line 2 does the same for the to variable.

Line 3 runs the associated XPath query against the DOM tree that represents the from variable and returns an appropriate fragment. Line 4 runs the XPath query specified for the to variable and determines the appropriate fragment.

Line 5 copies the from variable XML fragment to the to variable, replacing the selected XML fragment of the to variable. Line 6 transforms the to variable from the DOM representation to a literal XML document.

It is obvious that the assign processor of the navigator can optimize the processing to some extent if multiple assign parts are being processed. For example, all variables are converted to DOM representation when used the first time and transformed back to XML representation after all assign activity parts have been processed.

### 3.10.3. Time Dependent Processing

WS-BPEL defines a set of activities whose processing is time-dependent; that means they are triggered after a certain time has elapsed or a particular time has been reached. A typical activity is the *Wait* activity that suspends the execution of a process instance for a time interval or until a particular time has been reached. Other activities of this type are *pick* or *event handlers*.

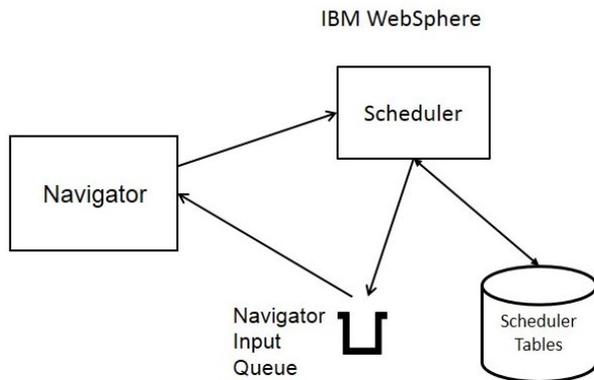


Figure 3.13.: Time Dependent Processing

Figure 3.13 shows the architecture needed in support of those activities. When the navigator reaches such an activity, it tells the IBM WebSphere scheduler when to signal expiration of the time. If, for example, a wait activity requires to wait for 30 minutes, then the timer service would be instructed to signal when 30 minutes have elapsed. When the timer fires, an appropriate message is inserted into the navigation input queue that tells the navigator to continue processing of the identified process instance.

### 3.11. Service Invocation Processing

The service invoker component carries out the processing shown in Listing 3.27. Line 1 triggers the processing by retrieving the next request message from the service invoker queue.

```
1   READ next request from the input queue
2   Construct the invocation message from
   input provided by the navigator
3   carry out appropriate service invocation
4   if error occurs then
5       INSERT appropriate fault message into navigator
   input queue
6   else
7       INSERT activity completion message into navigator
   input queue
8   end if
```

Listing 3.27: Service Invocation Processing

Line 3 then performs the actual service invocation, possibly with the help of an ESB. The return code from the invocation is checked. If an error occurred (Line 4), an appropriate fault message is inserted into the navigator input queue, causing the navigator to start fault processing according to the definitions in the process model. If all goes well (Line 6) an activity completion message is inserted into the navigators input queue. The navigator then processes the message as described in Section 3.9.3.

### 3.12. Object Identifier Generation

All objects in the runtime database, and almost all objects in the other databases, are uniquely identified via 12 byte identifier. This identifier is generated by the OID generator whenever a new object in the database is created.

The conceptually easiest way is to have a counter in the database that is incremented whenever a new OID is needed. Unfortunately, this approach can not be implemented for performance reasons; the sheer number of OID requests within a transaction results in the serialization of all transactions.

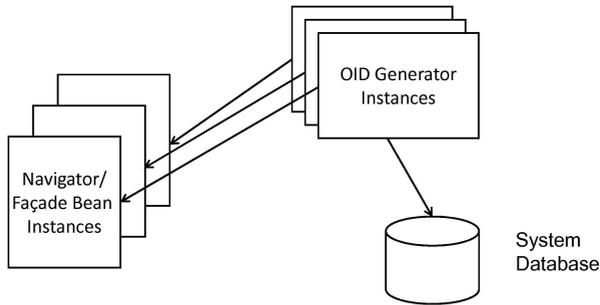


Figure 3.14.: OID Generation

Figure 3.14 shows the SWoM architecture that helps minimize the impact of generating identifiers. The OID generator manages the counter in the database as an SQL type `BIGINT`. The generator, instead of increasing the counter every time an OID is needed, obtains a set of numbers from the database. For example, if the counter in the database is 100.000, then the generator would set the counter to 101.000. Whenever a navigator/façade bean instance needs an OID, it is obtained from the next available number in the internal OID list. The actual OID is constructed by swapping the first four bytes with the last four, so that the OID are not delivered as sequential numbers to avoid any potential impacts on database performance. The final number is then base64 encoded [Bas06] into a 12 bytes fixed-length String to have a human-readable representation of the OID.

As there is no need for the OIDs to be consecutive, there is no problem with the SWoM being shut down. When started anew, the OID generator just obtains the previously stored counter and continues processing.

This design has been chosen in favor of using the appropriate Java UUID generator classes for a number of reasons, such as length of key, uniqueness, and greater flexibility in manipulation of the keys with respect to database performance.

There is no problem with running out of OIDs. The maximum number that a big integer field can hold is  $10^{19}$ . If 100 SWoM components request a batch of 1000 numbers every second, then the OID generator can deliver OIDs for  $10^{14}$  seconds, which amounts to approximately 3 million years.

### 3.13. Application Reliability

The SWoM components navigator, service invoker, and the façade beans carry out their processing as transactions. If a problem occurs, the transactions are rolled back and restarted. If a Web Service was invoked within a service invoker transaction, and the service invoker transaction terminates (for whatever reason), the service invoker transaction is restarted, causing the Web Service to be invoked again. This is no problem if the invoked Web Service is idempotent, the extra invocation is just extra processing.

The SWoM tries to minimize the risk of repeating the execution of a Web Service by reducing the time of the transaction in which the Web Service is invoked by invoking the Web Service in a separate transaction established by the service invoker component. A similar approach is taken by IBM Process Server by carrying out commits before and after the execution of the invoke activity (see [IBM13] for details); ORACLE BPEL Process Manager [ORA08] through the `idempotent` property associated with the invoke activity that causes the transaction to be committed right after execution of the invoke activity

Nevertheless, there are situations when the execution state of the invoked Web Service is unknown. For example, the call has been carried out and the system fails before the response message has been processed the caller. The only solution to this problem for non-idempotent Web Services is the execution of a called Web Service using Web Services Atomic Transaction (WS-AtomicTransaction)[OAS09], which IBM WebSphere implements. If the service invoker transaction fails for whatever reason, the changes made by the Web Service are also rolled back.



- The generated façade bean is handed over to the Java compiler to generate the appropriate class file. The SWoM library provides the appropriate classes that the generated façade bean exploits, such as the OID generator or navigator interfaces.
- Next the WSDL port/service generator generates the appropriate WSDL by using the provided WSDL logical parts and adding the physical parts.
- The WAR/EAR generator uses a set of EAR templates to construct from the façade bean class and the WSDL the appropriate façade bean EAR file.
- Finally, the SWoM deployer calls the IBM WebSphere deployer to deploy the generated EAR file into the IBM WebSphere environment.

### 3.15. Architecture Verification

The architecture of the SWoM is analyzed by performing a set of scalability tests using the benchmark introduced in the following chapter. Figure 3.16 shows how the SWoM scales with respect to CPU load. As can be seen, the CPU load correlates almost linearly with the amount of process instances that the SWoM carries out.

CPU load (%)	28	32	40	52	70
Main Processes/min	270	302	391	595	727
Main Processes/min/CPU load	9.6	9.4	9.8	11.4	10.3

Figure 3.16.: Scalability by CPU load

This finding is fortified by running the same benchmark with a different number of clients. Figure 3.17 shows the results of this test.

<b>Parallel requests/ CPU load</b>	<b>1</b>	<b>10</b>	<b>30</b>
Main Processes/min at 50 % CPU load	577	500	527

Figure 3.17.: Scalability by Number of Parallel Requests

Both test cases show that the SWoM delivers the load scalability that one expects from a piece of middleware. The correctness of the design has been verified via a set of other test cases that were not performance-related.

# PERFORMANCE TESTING

An important aspect of the design and implementation of a high-performance WfMS, as for any software system, is the development of appropriate performance tests that help determine if the architecture is sound and solid and that a particular optimization technique provides the expected performance improvements.

The construction of the performance tests for the SWoM requires that the factors contributing to the performance of the SWoM are well understood so that they can be appropriately factored into the generation of the performance tests.

This chapter discusses the different constructs within a WS-BPEL process model that have a major impact on the performance and shows how this information is used to construct a simple, yet expressive benchmark. This benchmark is then used during the development of the SWoM optimization techniques discussed in the upcoming chapters.

## 4.1. Performance-Relevant Factors

The performance of the SWoM is mainly impacted by two types of performance-relevant factors: (1) the resources necessary to handle the individual constructs in a WS-BPEL process which includes the CPU cycles needed to interpret the constructs and the DBMS related activities, and (2) factors such as the number of process instances in the runtime database that impact the buffering capabilities of the DBMS.

### 4.1.1. Process Model Factors

The WS-BPEL specification define quite a number of constructs ranging from the simple `empty` activity type to complex compensation handlers. This section tries to define basic elements.

#### 4.1.1.1. Synchronous Invocation

A WS-BPEL process can provide its client with a synchronous invocation interface via the use of a `receive` and one or more `reply` activities.

#### 4.1.1.2. Correlation Handling

The more prominent model of interactions with Web Services is the use of messages that are exchanged between the partners. The request of a partner, in this case, contains correlation information that the receiver uses when sending a response to the requester so that the requester can assign the response to the original request. WS-BPEL defines correlation as an appropriate mechanism; Section 3.9.4.2 shows how SWoM implements correlation. The importance of this interaction model makes it imperative that the SWoM handles it extremely efficient.

#### 4.1.1.3. Data Handling

An important facet of business processes is the management of data that a process instance receives as part of appropriate requests, creates from received

data, and sends to the Web Service the process is invoking. WS-BPEL defines that the information is maintained in variables, which are referenced in the appropriate activities. Of particular importance are assign activities that manipulate the contents of the variables with XPath, providing the mechanism to describe the manipulations to be carried out. It can be assumed that assign activities form a substantial part of a WS-BPEL process, so the associated processing should be as minimal as possible from a performance perspective. In addition, join/transition conditions which also use XPath to declare the rules are an important factor in data handling.

#### 4.1.1.4. Long-running Processes

Most processes are carried out in multiple steps. The process context, that means all process instance information, is stored in the runtime database at the end of a processing step and retrieved at the start of the subsequent processing step. It is obvious that the storage/retrieval of the process context and its transformation to and from the internal representation contributes significantly to the resource consumption of the SWoM.

#### 4.1.1.5. Parallelism

The execution time of business processes is typically reduced by carrying out tasks in parallel if possible. A set of parallel paths is started, conceptually via a fork activity, and finished via a join activity. The level of parallelism and the efficiency of processing join activities are important for the performance of non-sequential processes.

#### 4.1.2. General Factors

The performance of the SWoM depends also on the characteristics of the environment, such as the amount and size of process instances maintained in the runtime database and the number of parallel requests. Note that we have not performed any tests on long-running process instances, so the impact

of size of the runtime database is not evaluated (see Section 4.5 about the limitation of the developed benchmark.)

## 4.2. Performance Criteria

We use throughput as the main performance criteria following the performance objective for databases as proposed in [A<sup>+</sup>85] and defined by the Transaction Processing Performance Council in their respective benchmarks, such as the famous TPC-C benchmark. Note that this benchmark sparked significant research activities in the field of databases, resulting in an unexpected dramatic improvement of database technology, as shown in Figure 4.1 (the numbers have been obtained from [Rah10] and information published by the TPC Processing Council [Tra13]). It is some evidence for the approach of using throughput to measure middleware performance.

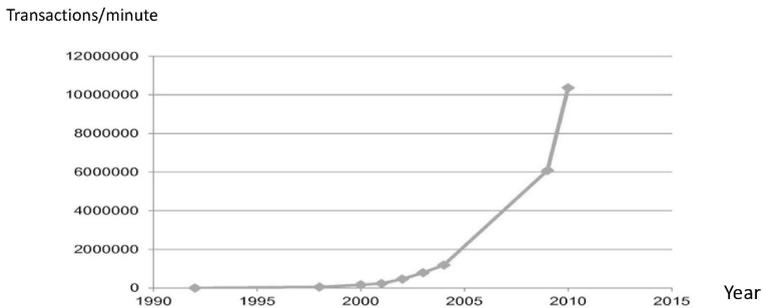


Figure 4.1.: TPC-C Benchmark Improvements

Another performance criteria could be the response/execution time of a business process. Under the assumption that the resource consumption of the WfMS is small compared to the resource consumption of the invoked Web Services, the response/execution time of business processes most likely depends on the response/execution time of the called Web Services. If the response/execution time of the called Web Service is very low, then the overall

response/execution time of the business process is very low, and vice versa. Consequently, response/execution time is not a useful criteria for measuring the performance criteria for a WfMS; at least for the majority of process models.

### 4.3. Performance Benchmark

A benchmark typically contains a set of scenarios that, when carried out, reflect a situation in the real world. Instead of creating a set of process models where each one is designed for testing a particular function and determining appropriate performance measurements, a set of interacting process models has been developed as the benchmark. The core of the benchmark is the process model shown in Figure 4.2. It should be noted in passing that the benchmark has been used by another team to evaluate the performance of JOpera workflow engine[PPBB13].

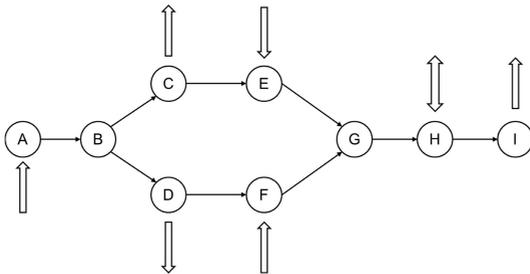


Figure 4.2.: Benchmark

The process is started via the receive activity A which accepts a message consisting of two fields. The assign activity B copies each field of the input message into single field variables, which are used by the two invoke activities C and D to invoke two asynchronous Web Services. The fields in the newly created variables are defined as correlation sets which are used by receive activities E and F. When the called Web Services return an appropriate message, the receive activities extract the correlation information and use the information for locating the appropriate process instance. The subsequent assign activity G

copies the returned fields of the two messages into a two field variable, which is then passed to a synchronous Web Service, and later again to an asynchronous Web Service.

All four Web Services that are invoked by the process are also defined via appropriate process models; that means the complete benchmark is defined and executes as WS-BPEL processes. The two asynchronous Web Services are implemented using a receive activity followed by an invoke activity; the synchronous Web Service is implemented as a receive activity followed by a reply activity; the final asynchronous Web Service just implements a receive activity. All Web Services that return a message basically return the received message.

Note that all process models are explained in detail in Appendix A including WSDL and WS-BPEL definitions, process deployment descriptors, and flow execution plans.

The benchmark addresses almost all of functions discussed in the previous section:

- The different interaction patterns between the different Web Services and the process are implemented as follows: The synchronous Web Service, invoked via activity H, the request-reply, the two Web Services, invoked by activities C and D, the message exchange, and the asynchronous Web Service, invoked by the final activity I, the fire-and-forget pattern.
- Since the Web Services respond right away, the message exchange is rather quickly stressing the correlation capabilities of the SWoM.
- The data handling is simulated with the usage of six different variables and the splitting and joining of the different fields of a variable; the impact of the variable/message size is simulated by starting the process with different message sizes.
- The parallel execution of the invocation of the two Web Services tests the impact of parallel execution.

## 4.4. Benchmark Setup

The performance tests are run using a client-server infrastructure. The server is an Intel Quad Core 3 GHz Processor with 8 GB memory, 5 disk drives, running Windows 7 64bit. The client is an IBM Thinkpad T60p with 2GB memory running Windows 7 32bit. The two processors are connected via a 52 Mbit WLAN .

### 4.4.1. Client Setup

The client machine runs a stand-alone JAX-WS program that starts the processes running on the server. This program reads in an XML file that controls the execution of the client. Listing 4.1 shows a sample XML file used for defining the client execution mode.

```
1    <?xml version="1.0" encoding="UTF-8"?>
2    <tns:TestCase ID="TC1"
3        xmlns:tns=
4            "http://testClient.swom.iaas/TestCase/"
5        xmlns:wSDL=
6            "http://schemas.xmlsoap.org/wSDL/"
7        xmlns:soap=
8            "http://schemas.xmlsoap.org/wSDL/soap/"
9        xmlns:wsa=
10           "http://schemas.xmlsoap.org/ws/2004/08/addressing"
11        xmlns:xsi=
12           "http://www.w3.org/2001/XMLSchema-instance">
13    <numberOfRequests>24000</numberOfRequests>
14    <parallelRequests>30</parallelRequests>
15    <requestDelay>3000</requestDelay>
16    <WSDLLocation>D:\ProcessA.wsdl</WSDLLocation>
17
18    <TCMessage>
19        <TCPart name="longMessage">
20            <field name="field1">
21                <length>100</length>
22                <correlationField>YES</correlationField>
23            </field>
24            <field name="field2">
25                <length>100</length>
26                <correlationField>YES</correlationField>
27            </field>
28        </TCPart>
29    </TCMessage>
```

```
24      <portType>TTProcASPPT</portType>
25      <operation>start</operation>
26    </tns:TestCase>
```

Listing 4.1: Benchmark Driver Data

Each test case is identified via an ID specified via the ID element shown in Line 2. The total number of requests that are executed in the test run is specified via the `numberOfRequests` element. The `parallelRequest` defines the number of clients that are to be simulated; the `requestDelay` element defines the time between two subsequent requests of the same client.

The message to be used in the requests is defined via the `TCMessage` element. Each part of the message is defined via the `TCPart`; the part is defined via the `name` element. For each of the parts, the properties of the individual fields are specified via the `field` element. The `length` element specifies the length that the test client should use for the field. The `correlationField` element specifies that the field is used as a correlation set; this tells the client to vary the contents of the field, so that each time a new value is generated for the field. This makes the correlation sets unique, which is required to avoid any problems when carrying out correlation.

#### 4.4.2. Server Setup

The server runs a standard IBM WebSphere V 7.0 64bit and a IBM DB2 Enterprise Edition V 9.5. No special tuning has been applied to these middleware components; this was done under the assumption that the defaults make sense in most cases. As said before, the system has five disks attached; the disks are assigned the drive numbers C: through G:.

The operating system including the page file resides on C:

WebSphere has been installed on D:, so that all message-related activities are confined to this disk.

DB2 has been installed on E:, so that all log-activities of the database are stored on this disk.

The databases are managed on disk F:. We have not stored the various databases on different disk drives, since only the runtime database is used after the system has started up.

## 4.5. Benchmark Limitations

The benchmark is not perfect for several reasons:

- It underestimates the impact of IBM DB2. The process instances are short-running. It can therefore be expected that IBM DB2 can process all SQL calls through appropriate buffer look-ups and does not need to go to the database. Furthermore, the process instances are deleted after completion, so that it can be assumed that IBM DB2 actually never needs to write to the database and can discard the database logs immediately after receiving the delete request. Further work is needed to construct a benchmark that addresses the impact of long-running process instances, which should result mainly in the additional resources required by IBM DB2.
- The CPU can not be driven to saturation, since the main benchmark process does not respond to the submitter of the requests. So the CPU is driven at a level without starting thrashing. This would have required a much more sophisticated set-up. The load benchmark runs indicate that the SWoM almost scales perfectly with respect to CPU load and hardware exploitation. This validates the assumption that the SWoM would not show any performance degradation when getting close to CPU saturation.
- No advanced functions such as compensation are part of the benchmark. This has deliberately been left out, since the SWoM currently does not support compensation. Note that adding any additional activity types has no impact on the performance of existing functions. The architecture of the SWoM clearly separates the internal handling of the individual activity types.

## 4.6. Benchmark Execution

The different test cases that are performed are all carried out following the same procedure:

- The benchmark to be tested is installed via the SWoM administration.
- The server is re-booted, so that no residues of whatever software are left on the machine.
- All operating system services, that are not needed for running the benchmark are shut down. Only the antivirus system is allowed to be up and running.
- Each test case executes 24.000 instances of the benchmark in about 40 minutes.
- The client emulates 30 requestors. Think time between subsequent requests was adjusted to run at 50-60 % CPU load.
- CPU load is measured using the Windows performance monitor using a sample every second and using a window of 480 seconds for the CPU average. Two readings of the meter are carried out: 36 minutes and 39 minutes after the benchmark started. If the two numbers are fairly close, the average of the two meter readings is constructed. If they are very different, it can be assumed that one meter reading included more garbage collection cycles. In this case, the test case is repeated taking the meter reading at 35 and 38 minutes.
- Since the measured CPU load of the different benchmark tests was different for each of the test cases, normalization was needed to take place so the different test cases can be compared. A CPU load of 97 % has been chosen; the load that has been achieved in the calibration tests.

# BASE CACHING

Chapter 3 described the structure and processing of the SWoM. All processing was carried out by directly going to the database when needed. This approach was chosen to illustrate how the SWoM conceptually works and how it is using the information in the buildtime and runtime databases.

A single call to the database of the SWoM is certainly not that exorbitant; however, the sheer number of SQL calls needed during process execution makes the approach of always interacting with the database prohibitively expensive. Even if the objects are located in the IBM DB2 buffer, each call requires quite a number of CPU cycles for the SWoM to set up the IBM DB2 request, which requires the transformation of internal information into an SQL query, for IBM DB2 validating the request, locating the object in the buffer, preparing the response, sending back the result, and for the SWoM transforming the returned information into the object structure used for internal processing.

The obvious answer to help elevate these performance issues is the introduction of caching. This chapter presents those base caching strategies that the SWoM implements that apply to all process models and that need no specific information to be very effective. The base caching introduced in SWoM includes the following caches:

- *Process Models* are kept in memory in a form suitable for immediate usage. No access to the buildtime database is required for normal navigation, process instance creation, or process model queries.
- *Process Instances* are kept in memory for the time of a transaction to avoid multiple reading and writing of variables and to exploit the capability of using the SQL batch update facility.
- The *System Deployment Descriptor* is kept in memory to avoid loading it when any of the components requires the information.

There are other areas where the SWoM performance can be improved through caching. However, those approaches either need information that a process modeler must supply (more about those caching approaches in Chapter 7) or the information is determined by the flow optimizer as described in Chapter 8.

## 5.1. Model Cache

The caching of process models is fundamental for achieving performance in the SWoM. In fact, even the initial SWoM implementation had a process model cache. So, no performance numbers are available that help understand the performance penalty that would be incurred without process model caching.

The process model cache delivers its performance improvement by eliminating the complete processing that is associated with the storage of the information in the buildtime database and its access through IBM DB2.

First, there is no need to transform the process model from the external representation in the buildtime database into the internal representation that the navigator needs for processing. The associated costs are quite high, as the SWoM uses Java objects for representing the different parts of the process model.

Second, it saves the costs for carrying out the appropriate SQL calls, which includes the costs for moving the exchanged information between the different operating system processes in which the application and the database server

execute, for analyzing the submitted query, the access of the DBMS cache, and preparing of the query result.

The model cache not only helps the navigator but also supports administrative functions. Queries about process models, for example, can be answered without going to the DBMS. If, for example, a process administrator would like to display the structure of a process model, then this information can be obtained from the process model cache.

Most WfMSs, such as IBM Process Server or Oracle BPEL Manager, seem to have a cache for process models. Figure 5.1 illustrates the structure of the process model cache in the SWoM.

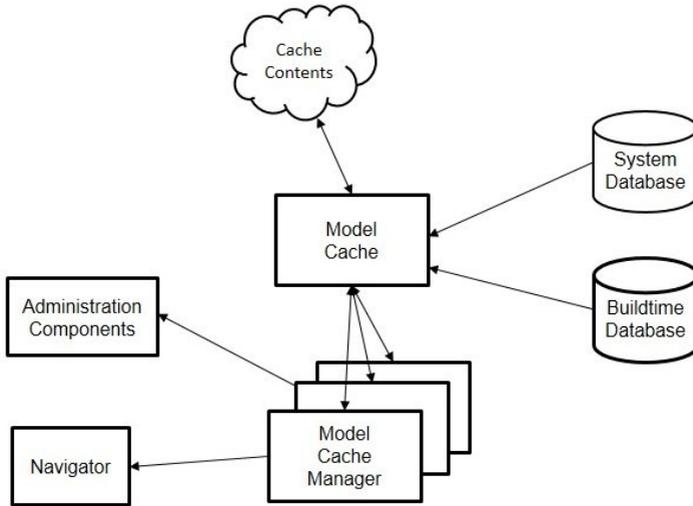


Figure 5.1.: Model Cache

The model cache manager (MCM) provides the access to the model cache information, which is maintained by the model cache (MC). For performance and implementation reasons, several MCM instances exist. The model cache is implemented as a singleton [GHJV95]. When an instance of the MCM is created, it calls the MC for a particular process model. If this call is the first one

for the MC, the MC creates an instance of itself and allocates the appropriate storage for the cache content.

It is the responsibility of the model cache manager to make sure that the most used process models are kept in memory. Whenever a process model is requested that is not in the cache, the model cache manager retrieves it from the buildtime database. Listing 5.1 shows the settings that can be specified in the system deployment descriptor to help the model cache manage the cache contents.

```
1 <systemDeploymentDescriptor>
2   <modelCache>
3     <technique>PriorityBasedLRU</technique>
4     <size>1024MB</size>
5     <maxEntries>100</maxEntries>
6     <fixModel>
7       <model>MillionDollarLoan</model>
8       <model>MajorDisaster</model>
9     </fixModel>
10  </modelCache>
11 </systemDeploymentDescriptor>
```

Listing 5.1: Model Cache Definitions

The `modelCache` element in Line 3 is used to define the properties of the process model cache. The `technique` element specifies the technique that the cache manager should use for controlling the content of the cache. There are several options for doing this: `PriorityBasedLRU`, as shown in the example, would manage the process models using a least-recently queue which favors process models with a higher priority. The cache priority of a process model is defined using an appropriate definition in the deployment descriptor of a process, as shown in Listing 5.2.

```
1 <processDeploymentDescriptor>
2   <cache>
3     <modelPriority>5</modelPriority>
4   </cache>
5 </processDeploymentDescriptor>
```

Listing 5.2: Process Model Cache Properties

The `cache` element in the deployment descriptor is used to specify the cache properties of the process model. The `modelPriority` indicates the priority of

the process model in the process model caching. The higher the priority, the more important is the process model and causes the process model to be fixed in the process model cache.

The rest of the elements in the process model cache definition is used to control the allocation of the process model cache and the fixing of process models in the cache. The `maxEntries` element specifies the number of process models that the cache should hold.

The `fixModel` provides the capability to define which process models should be fixed in the model cache. The individual process models are identified through appropriate `model` sub elements. In the example, the process models `MillionDollarLoan` and `MajorDisaster` are fixed in the process model cache; that means the process models are never flushed from the process model cache. Process models fixed in the process model cache allow the processing of incoming requests without the need to possibly retrieve the process model information from the database and to create the internal representation: that means it provides the capability to process process instances of the process model most efficiently. The disadvantage of fixing process models is that those process models occupy storage even if no process instances of the process model are processed for some time. Defining the process model to be fixed in the SSDD has the advantage that all information is provided in a single place; the SWoM alternatively allows to specify in the SPDD whether the process model should be fixed.

The model cache maintains statistical information about the contents of the cache, such as the number of times a particular process model was accessed and how often the information has to be read from the database. This information can be queried by the process modeler to further tune the model cache.

## 5.2. Instance Cache

The instance cache manages all objects that are accessed, modified or deleted within a navigator transaction. Figure 5.2 illustrates the basic setup of the instance cache. Incidentally, from a performance perspective, the situation is the same as the one for the model cache; no performance figures are available

for the SWoM without an instance cache.

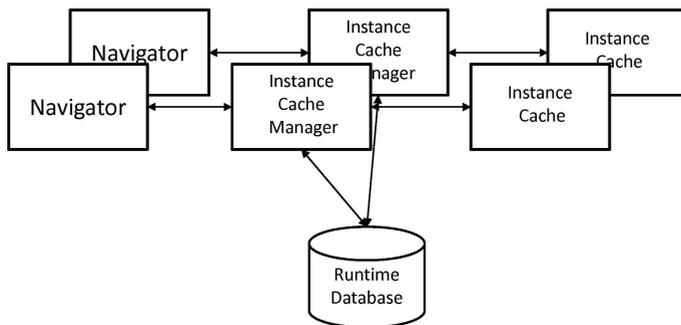


Figure 5.2.: Instance Cache Processing

When a new navigator instance is created, the appropriate *postConstruct* code in the navigator is carried out. This code sets up the necessary environment for the navigator, such as the creation of a process execution context object, that anchors all information that is needed by all modules and loaded classes for carrying out the required task.

The *postConstruct* code also loads, among some other classes, the instance cache manager (ICM). It then passes the process execution context to the initialization code of the ICM for appropriate setup operations.

The ICM consists of three major components, the actual instance cache, the instance cache loader, and the batch update processor, as shown in Figure 5.3.

The ICM provides the appropriate application programming interfaces that allow the navigator to request the typical CRUD (create, retrieve, update, delete) [Mar83] operations on different object instances, such as activity, variable or correlation set instances.

The ICM maintains the information in a tree that reflects the structure of a process instance. The process instance is at the root with object instances hanging off it. Two lists are maintained for each object type: one, the found

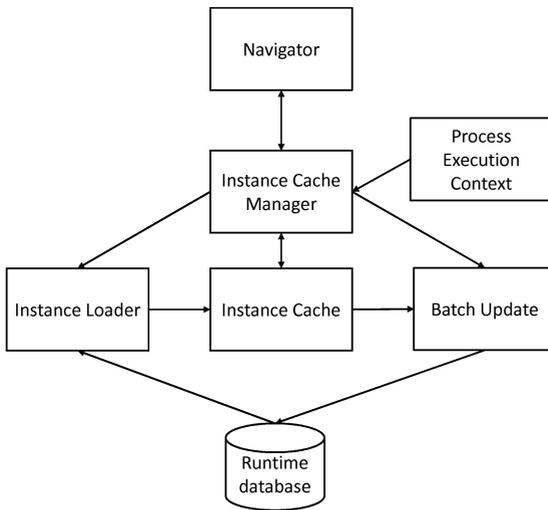


Figure 5.3.: Instance Cache Structure

list, contains the object instances that have been found, and one, the not-found list, contains the objects for which no instance has been found in the runtime database. The not-found list helps the ICM to answer a second navigator request for the same object without going again to the runtime database. They are all anchored in the process instance, so that the cache can be cleared by simply nullifying the process instance, an action performed when the transaction starts. Each object instance is associated with a state marker that tells the ICM whether the object has been created, updated, or should be deleted.

When the ICM can not honor a particular request (the object has never been requested before), it calls the instance loader to obtain the object from the database. If the object can not be found in the database, an appropriate entry is stored in the not-found list. If found, an appropriate object is created in

the found list, if necessary clearing an entry in the not-found list. A particular important task of the instance loader is to find a process instance based on correlation information associated with the message received. This is a change to the base architecture presented where the correlation manager is responsible for locating the process instance; the change has been introduced to consolidate all SQL calls in the ICM.

The batch update facility is called when the navigator reaches the end of processing. The navigator calls the *persist* method of the ICM, which in turn calls the batch update facility to persist the cache contents to the runtime database. The batch update facility uses, for performance reasons, the SQL batch update facility. It runs through the cache, analyzes all objects, and adds, if the object has changed, an appropriate request to the SQL batch. When finished, the SQL batch is executed.

The coupling between the ICM and the navigator is a rather tight one. When the navigator needs to create an instance of an object, it calls the ICM to create such an instance. This instance is then returned to the navigator; from then on, the navigator treats this object as its own; that means it updates fields within the object as required (without making a call to the instance cache manager). This level of handshaking (and trust) provides better performance than a more defensive approach where the objects are solely under the control of the ICM.

### 5.3. System Deployment Descriptor Cache

The System Deployment Descriptor (SSDD) is used by all components within the SWoM to determine the proper processing, so it makes sense to cache the SSDD to avoid that every component needs to go to the database. Figure 5.4 shows the structure of the System Deployment Descriptor Cache Manager (SDDCM).

The System Deployment Descriptor Cache Manager(SDDCM) implements the same structure as the Model Cache Manager using the approach of a singleton so that only one copy of the system deployment descriptor is maintained.

The SDDCM offers a function that allows the navigator or any component to check whether the SSDD has changed. When the SDDCM returns an SSDD,

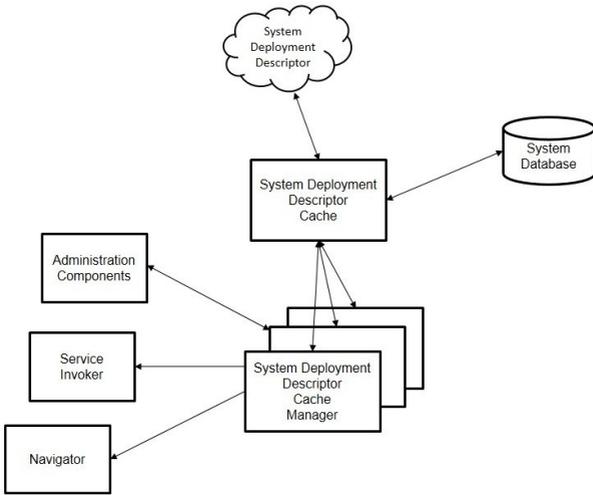


Figure 5.4.: System Deployment Descriptor Cache

it returns a unique ID that is associated with the SSDD. The navigator stores this ID. The SDDCM also supports a function that just returns this ID. When the navigator initially starts, it obtains the system deployment descriptor from the SDDCM together with the associated ID. When the navigator starts a new transaction, it obtains from the SDDCM the ID of the current SSDD. If the stored ID is different from the returned ID, the navigator requests a new copy of the system deployment descriptor and starts internal clean processing with the new SSDD.



## TRANSACTION FLOWS

As discussed earlier, all processing of the SWoM, be it the navigator or the service invoker, is carried out as transactions. The set of navigator transactions and its order, that is associated with a particular process model, is called the *transaction flow* of the process model. The transaction flow is constructed by combining adjacent activities into transactions and relating the different transactions via connecting links. Activities are considered adjacent if they are connected via a link, are part of a sequence activity, or a part of a flow activity and not connected via a link. The connecting links have the semantic of WS-BPEL links; that means they can be associated with a transaction condition. In fact, a transaction flow has conceptually the same structure as a process model with transactions instead of activities and having links between the transaction similar to links between different activities. The granularity of the transactions and thus the structure of the transactions is initially specified by the process modeler using the *transaction flow type*, which allows the navigator to carry out the transaction flow without any further assistance and without constructing the transaction flow at all. The transaction flow concept provides the base of the flow optimizer presented later in Chapter 8, where the flow optimizer actually constructs the transaction flow and uses the constructed

transaction flow to perform appropriate optimizations.

## 6.1. Transaction Flow Types

*Transaction flow types* define the rules by which the different transactions and their boundaries are constructed. A transaction boundary is always necessary for a wait, pick, synchronous invoke, or receive activity because navigation of the process instance must be suspended until the activity can be processed: for a receive or pick activity until the message arrives, for an invoke activity that invokes a synchronous Web Service for completion of the service invoker, and for a wait activity until the specified time has arrived. Additional transaction boundary specifications, as shown in Figure 6.1, result in five basic transaction flow types. The transaction is completed when one of the specified transaction boundaries is processed. For example, in the medium transaction flow type a new transaction is started after a fork or join activity has been processed.

Transaction Flow Type	Requested Boundaries
very short	activities, links
short	activities
medium	fork activity, join activity
long	join activity
ultimate	non-finished join activity

Figure 6.1.: Transaction Flow Types

The difference between the various transaction flow types is the amount of transactions that are being carried out, with the short transaction flow type resulting in the most transactions and the ultimate transaction flow type in the least transactions being executed. Less transactions mean more processing within a transaction and thus the individual transactions take longer. The advantage of less transactions is the reduced processing in terms of database

processing, CPU cycles for managing the transactions in the SWoM, and transaction processing by the application server that provides the transaction support. The disadvantage of less transactions is the prolonged execution of the transactions which results in the need for the transaction manager to hold the resources longer. Another disadvantage is a possible impact on response-time sensitive processes. If all SWoM components are in use, then no response-time sensitive processes can be run until one of the SWoM components has completed processing. The performance tests that have been carried indicate that longer transactions provide better throughput. The final arbitrator is the process modeler that must weight the pros and cons of the different transaction flow types. Note that the flow optimizer introduced in Chapter 8 helps finding a good transaction flow. In addition, the infrastructure support presented in Chapter 9 helps to recognize problems such as running out of component instances.

#### 6.1.1. Processing

The basic processing of the different transaction flow types is almost identical. When the transaction boundary is reached during navigation, the transaction is committed.

The set of next transactions is started as as the result of one of the following events taken place:

- The navigator generates for all outgoing links appropriate navigation messages for processing the target transactions, stores them in the navigation queue, and processes those requests in sub-subsequent transactions.
- The service invoker inserts a service completion message into the navigator queue, which is then processed by the navigator.
- A façade bean processes a request and calls the navigator for handling the request. In this case, the transaction is started by the façade bean, which is then joined by the navigator.

The scope of a transaction encompasses the following resources:

- The process instance or parts of a process instance that is being processed in the transaction.
- The navigation message if the transaction is started via a message obtained from the navigator queue.
- Any messages generated during the transaction, such as service invoker messages to have the service invoker call a Web Service or navigator messages that the navigator is sending itself by putting them into the navigator queue.
- The reply message if the process represents a synchronous Web Service.
- The façade bean if the initial request is handled by the façade bean.

### 6.1.2. Short Transaction Flow Type

In the short transaction flow type, each activity (including the outgoing links) of the benchmark process is processed in a separate transaction; this transaction flow type is used as the base in the calibration measurements shown in Section 1.9.

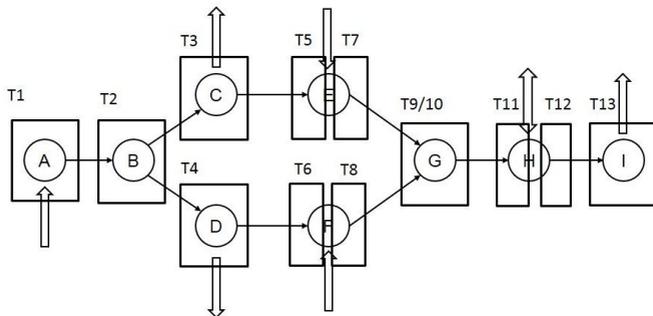


Figure 6.2.: Short Transaction Type

As can be seen in Figure 6.2, 13 transactions are being carried out: one transaction for activities A, B, C, D, and I. Two transactions are required for the receive activities E and F (one for creating the activity and one for processing the message handled in the activity), the join activity G (since two links are entering the activity), and the synchronous invoke activity H (for invoking the called Web Service and for processing the answer from the Web Service).

### 6.1.3. Medium Transaction Flow Type

In the medium transaction flow type, fork and join activities cause the transaction to be completed. Figure 6.3 shows the appropriate transaction flow for the benchmark process. As can be seen, seven transactions are needed to process the benchmark process. The first transaction T1 contains the initial receive activity A and the assign activity B that creates the input variables for the two invoke activities C and D. The activity B concludes the transaction since it is a fork activity (having multiple outgoing links).

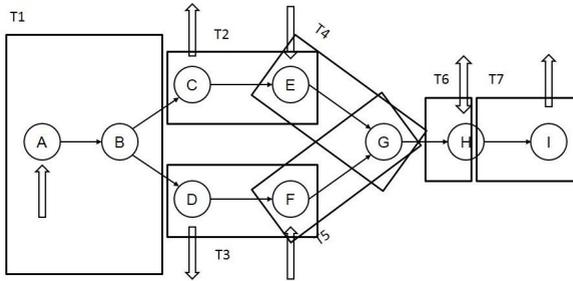


Figure 6.3.: Medium Transaction Type

The transactions T2 and T3 contain the activities C and D, that invoke the defined asynchronous Web Service, and the associate receive activities E and F. As pointed out earlier, receive activities result in a transaction boundary. It should be noted that the receive activities are just created and put into the state running (see Section 3.10.1; the completion of the activities is carried out in the subsequent transactions T4 and T5, when the messages that are to be

processed by the receive activities have been received.

Transactions T4 and T5 are started when the messages for the receive activities E and F have been accepted. The assign activity G is a join activity so that it is part of both transactions. It also causes the transactions to be finished.

The invoke activity H performs a synchronous invocation of a Web Service; it is therefore part of two transactions: transaction T6 in which the activity sends the invocation request to the service invoker and puts the activity into the state running and transaction T7 that processes the completion of activity H.

Transaction T7 completes the processing of the process instance by running the activity I, which asynchronously invokes a Web Service.

#### 6.1.4. Long Transaction Flow Type

In the long transaction flow type, only join activities cause the transaction to be completed. The initial transaction T1 now includes the execution of the invocation activities C and D and the creation of the appropriate receive activities E and F. Note that in this case the parallel paths are carried out sequentially, since the navigator can only process one activity at the time, as the navigator cannot create any new parallel threads (mandated by the EJB specification).

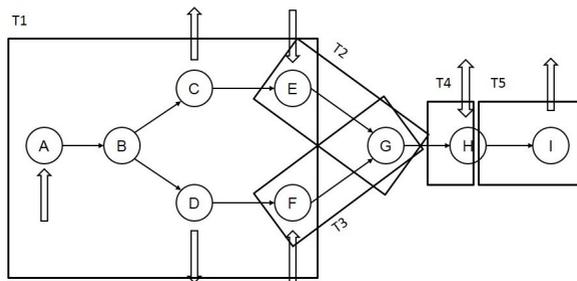


Figure 6.4.: Long Transaction Type

Transaction T2 includes the processing of the message associated with receive activity E and the partial processing of assign activity G. Transaction T3 performs

the same as transaction T2. Activities H and I are processed as in the medium transaction flow type in two transactions.

### 6.1.5. Ultimate Transaction Flow Type

Figure 6.5 shows the execution of the process when carried out using the transaction flow type: the number of transactions is now down to four. It should be mentioned that this is the minimal number of transactions that can be achieved: the two transaction boundaries for the receive activities F and E, one for the transaction with the join activity that is executed first, and one for the completion of the process.

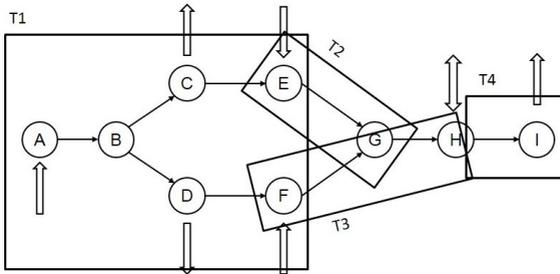


Figure 6.5.: Ultimate Transaction Type

Transaction T2 is carried out the same way as in the long transaction flow type. Transaction T3 however does not stop after processing activity G; since now all links have entered activity G, activity G is processed and the transaction continues including the invocation part of activity H. Note that the processing of T2 may occur before T3, if activity F is carried out before activity E. Transaction T4, as in the medium transaction flow type, concludes the processing of the process.

Figure 6.6 summarizes the number of transactions for each of the different transaction flow types. The number of transactions goes down, as can be seen, quite heavily, in particular from the very short to the short, and again from the short to the medium.

<b>Transaction Flow Type</b>	<b>Navigator Transactions</b>	<b>Service Invoker Transactions</b>
very short	22	4
short	13	4
medium	7	4
long	5	4
ultimate	4	4

Figure 6.6.: Number of Transactions

It is the ultimate transaction flow type that has the minimum amount of transactions that can be achieved without advanced configuration and optimization techniques presented in Chapter 7 and Chapter 8. We have three transaction boundaries: the receive activities, the join activity, and the synchronous invoke activity, so we end up with four transactions.

The workflow engine can carry out the different transaction flow types without any help; that means the engine just needs to know which transaction flow type it should execute and then performs the desired processing.

The reduction in transactions goes hand in hand with the reduction in SQL calls, as shown in Figure 6.7. The reduction can be solely contributed to the instance cache presented in Section 5.2, which reduces the number of SQL calls within a navigator transaction.

The first column `SQL calls` shows the number of non-batch update SQL calls, that means regular SQL calls. Most calls are `SELECT` calls to load the data from the runtime database into the transaction cache. Column `SQL batch update calls` shows the number of SQL batch update calls; column `calls in batch update` lists the number of actual SQL calls handled via the SQL batch update calls. As can be seen, the average number of calls in a batch update call increases from around one for the short transaction flow type to more than four in the ultimate transaction flow type. Column `Total SQL calls`, the

Transaction Flow Type	SQL calls	SQL batch update calls	Calls in Batch update	Total SQL calls	Total SQL requests
short	33	12	14	45	47
medium	22	6	13	28	35
long	18	4	13	22	31
ultimate	16	3	13	19	29

Figure 6.7.: Number of SQL Calls

most important column, shows the number of SQL calls that the SWoM issues against IBM DB2. The last column Total SQL requests shows the number of SQL calls that would be required if the SQL batch update facility is not used.

It is easy to observe that the number of SQL calls and the impact of the SQL batch update facility directly correlate with the number of transactions that are carried out.

The number of messages that are needed shows the same picture, as illustrated in Figure 6.8.

Transaction Flow Type	Nav msgs Read by Nav	Nav msgs put by Nav	SI msgs put by Nav	FSI msgs read by Nav	SI msgs read by SI	FSI msgs put by SI	Total msgs
short	9	9	4	1	4	1	28
medium	3	3	4	1	4	1	16
long	1	1	4	1	4	1	12
ultimate	0	0	4	1	4	1	10

Figure 6.8.: Number Of Messages

The columns (Nav msgs read by nav) and (Nav msgs put by nav) show the number of navigation messages that the navigator reads from and puts into its input queue. These messages are used when the navigator completes processing and needs to initiate the processing of the next transaction. It is interesting to observe that in the case of the ultimate transaction flow type, the navigator is no longer creating messages for itself. In other words, no longer are navigator transactions chained together; all chaining is done between navigator, service invocator, and invoked Web Services.

Column (SI msgs put by nav) shows the number of service invocator messages that the navigator puts into the input queue of the service invoker to have a particular Web Service be invoked; since the process contains four invoke activities, the service invoker must be called four times. Column (FSI msg read by nav) shows the number of finished service invocation messages that the navigator reads from its input queue and have be inserted by the service invoker. This type of message is only needed for the synchronous invocation of a Web Service, where the service invoker needs to provide the navigator with the information returned by the Web Service.

Column (SI msgs read by SI) shows the number of messages that the service invoker reads from its input queue; column (FIS msgs put by SI) shows the number of finished service invoker messages that the service invoker puts into the navigator's input queue after processing a synchronous Web Service.

The last column (Total msgs) shows the total number of messages that are required during the execution of the process. The table shows the same characteristics as the other tables with the number of transaction and SQL calls.

## 6.2. Test Results

Figure 6.9 shows the performance improvements that are achieved using the different transaction flow types. Note that, as already mentioned, no performance figures are available for the very short transaction flow type.

Note that the invoked Web Services are so simple, that there is no significant performance difference between having them run using the medium, long,

Transaction Flow Type	Main Processes (1/min)	Total Processes (1/min)	Improvement (%)
short	876	4380	-
medium	1026	5130	17.1
long	1074	5370	22.6
ultimate	1110	5550	26.7
ultimate/ WSs optimized	1143	5715	30.5

Figure 6.9.: Throughput

and ultimate transaction flow types. It is therefore only the main process that contributes to the performance improvements. The actual performance improvements will be slightly higher if all involved processes can take advantage of the more advanced transaction flow types.

The last figure is obtained by running the main process with transaction flow type `ULTIMATE` and all invoked Web Services optimized with the flow optimizer introduced in Chapter 8. These numbers are used as the base for all configuration and optimization techniques introduced in the subsequent chapters; they are the numbers the SWoM can achieve without any help by the program modeler or the flow optimizer.

### 6.3. Execution Characteristics

This section presents two aspects that are important to understand the execution characteristics of transaction flows and their realization in the SWoM: the logical and physical execution of transactions and the usage of an internal queue to process the activities within a transaction.

### 6.3.1. Logical and Physical Execution

Figure 6.10 shows the transaction flow that is associated with the medium transaction type of the benchmark (see also Figure 6.3).

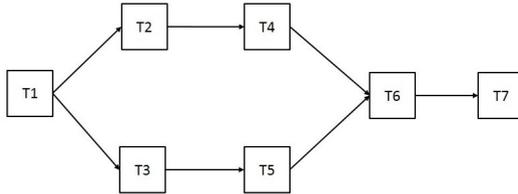


Figure 6.10.: Transaction Flow Structure

This structure represents the logical execution of the process: the parallel transactions T2 and T3 are carried out independently of each other. However, this does not mean that they are physically carried out in parallel; that means are carried out at the same time by two different navigator instances. In fact, they are processed sequentially, as shown in Figure 6.11.

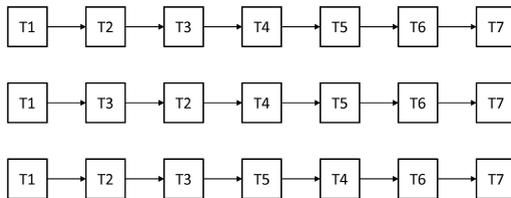


Figure 6.11.: Transaction Flow Physical Execution

The figure shows three different physical execution paths. In the first one, the transaction T3 is carried out after T2 and the join transaction T5 is carried out after T4; in the second one, the transaction T2 is carried out after T3; and in the third one, the execution sequence of the two join transactions T4 and T5 has been reversed.

The SWoM ensures the sequential execution of transactions by obtaining the process instance, when parallel transactions are (potentially) carried out,

with a SELECT SQL call that specifies the WITH RS USE AND KEEP UPDATE LOCKS. This causes IBM DB2 to acquire an update lock on the process instance, and all other navigator transactions that access the same process instance wait until the navigator transaction that acquired the lock has completed the transaction.

The SWoM enforces the sequential execution to avoid deadlocks that may occur if multiple navigator instances update the same process instance object, for example a variable, at the same time.

The sequential execution of a flow has two, most likely minor, disadvantages. First, the navigator may acquire an exclusive lock even in situations where this is not necessary, since the set of process instance objects processed in parallel paths is disjointed. Second, the total execution time of a process instance is higher. Note that the flow optimizer introduced later in Chapter 8 performs an appropriate analysis (Section 8.3.7) to determine if the execution of the parallel paths is deadlock-free, so that parallel paths in fact could be processed in parallel.

### 6.3.2. Architectural Changes

All transaction flow types, except the short and very-short transaction flow types, require a simple architectural change to implement the processing of larger transactions: the addition of an internal queue as shown in Figure 6.12. Conceptually the navigator uses this queue for managing requests within a transaction. It inserts messages into the queue and processes the internal queue until the queue is empty. Messages that leave the transaction are inserted into the navigator queue as done normally.

The following illustrates how the queue is used in transaction T1 of the medium transaction flow type. After the receive activity A has been processed, the outgoing link of A is followed to the assign activity B. Since the transaction boundary has not been reached, a request for processing activity B is inserted into the internal queue. Next, the internal queue is processed, resulting in the execution of activity B. Activity B is a fork activity so that the transaction needs to be completed. The target activities C and D need to be processed in new transactions, so appropriate messages for processing activities C and D are

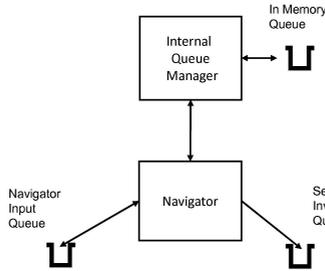


Figure 6.12.: Internal Queue

inserted into the navigator input queue. The internal queue is now empty and the transaction is finished by simply exiting the navigator code, causing IBM WebSphere to complete the transaction.

### 6.3.3. Processing Algorithm

The following set of listings describes in detail the algorithm that is used to carry out a transaction in transaction flow type **ULTIMATE**. Extra processing is carried out to cope with the situation that activities are part of a sequence activity or are not connected in a flow activity; this processing has been left out for simplification of the algorithm.

- 1 start transaction
- 2 process start activity
- 3 **call** DetermineNextSetOfActivities
- 4 **call** HandleInternalQueue
- 5 finish Transaction

Listing 6.1: Navigation - Transaction Main Processing

Listing 6.1 shows the main processing that the navigator carries out when a new transaction is started. Line 1 starts the transaction with Line 2 processing the start activity. The transaction is started either by a façade bean calling the navigator to process a receive or pick activity or by the navigator reading a message from its input queue.

```

1  function DetermineNextSetOfActivities
2    for all outgoing links do
3      if activity is receive activity in creation phase then
4        return
5      end if
6      create message for target activity
7      if current activity is skipped then
8        flag message as skipped
9      else
10     if transition condition evaluation fails then
11       flag message as skipped
12     end if
13     end if
14     insert message into internal queue
15   end for
16 end function

```

Listing 6.2: Navigation - Determine Next Set of Activities

After the start activity has been processed, the set of next activities is determined in Line 3 using the processing shown in Listing 6.2. These activities are then processed in Line 4 using the code shown in Listing 6.3. After all activities in the transaction have been processed, the transaction is finished (Line 5).

Listing 6.2 conceptually implements the navigation logic, that means the traversing of the graph. Line 2 starts a loop over all outgoing links. Line 3 determines if the current activity is a receive activity in creation phase (or any other multi-phase activity) that causes the transaction to finish) and is so, returns control as there is nothing to do anymore. Line 6 creates the message that identifies the activity that is the target of the link.

If the current activity is skipped (for example as the result of dead-path-processing) (Line 7), an appropriate flag in the message is set to indicate this to the target activity (Line 8). If the current activity was processed correctly, the transition condition is evaluated in Line 10. If the transition condition evaluates to false (Line 10), the message is also flagged as skipped, causing DPE to start (Line 11). Finally, Line 14 inserts the message into the internal queue.

Listing 6.3 conceptually implements the result of the navigation that is carried in Listing 6.2. It reads messages from the queue until the queue is empty; in this case all activities within the transaction have been processed.

Line 3 gets the message from the queue and identifies the activity that is the target of the message, which is then processed in Line 4. Line 5 checks whether the activity has finished; if the activity is a join activity then the activity will be processed multiple times before finishing. If finished, the set of next activities is carried out by calling the processing in Listing 6.2).

```
1  function HandleInternalQueue
2      while queue is not empty do
3          get message from queue
4          call HandleActivity
5          if activity finished then
6              call DetermineNextSetOfActivities
7          end if
8      end while
9  end function
```

Listing 6.3: Navigation - Handle InternalQueue

Listing 6.4 shows the function that handles the processing of the current activity which is obtained from the internal queue.

```
1  function HandleActivity
2      obtain activity
3      if join activity then
4          if all incoming links processed then
5              if join condition fails then
6                  set activity failed
7                  set activity finished
8                  return
9              end if
10             else
11                 increase number incoming links
12                 set activity not-finished
13                 return
14             end if
15         end if
16         process activity
17         set activity finished
18     end function
```

Listing 6.4: Navigation - Handle Activity

Line 3 determines if the activity is a join activity and if so the processing between Line 4 and Line 15 is being carried out. Line 4 checks if all incoming

links have been processed and if so, Line 5 checks the join condition associated with the activity. If the test fails, the activity is set to failed and processed and processing finishes.

Line 10 is carried out when not all incoming links have entered the activity. In this case, the number of incoming links is increased and control returns to the caller.

Line 16 starts processing of the activity. When complete, the state of the activity is set to finished.





CHAPTER  
7

# FLOW CONFIGURATION

This chapter discusses a set of configuration options that can be set by a process modeler via the Process Deployment Descriptor associated with a process model and, if set properly, provide large performance improvements. They fall into two major categories: (1) those that are implemented using different internal processing and (2) those that allow the selection of a database schema that handles more efficiently instances of a particular process model. Appropriate test runs demonstrate the achieved performance improvements, which are summarized in Figure 7.10. Note that all test cases are run using the ultimate transaction flow type with the Web Services fully optimized.

## 7.1. Internal Processing

This section presents a set of configuration properties that tell the SWoM to change its internal processing. In particular, the following configuration techniques are discussed:

- The usage of intra engine binding, which bypasses SOAP/HTTP when calling a Web Service, that is implemented as a WS-BPEL process man-

aged by the same SWoM. The service invoker in this situation calls the navigator directly via the appropriate navigator façade bean's local/remote interfaces with the appropriate request information.

- The invocation of a Web Service by having the navigator call the service invocator via the service invoker's Java local/remote interface instead of inserting an appropriate request message into the service invoker's input queue; this approach has been appropriately labeled *direct invocation*.
- The notion of *micro flows* as a particular way of executing a process model in a single transaction.
- The usage of a correlation cache for maintaining correlation set information between the invocation of a Web Service and its response to possibly eliminate the need to retrieve the correlation information from the runtime database.

### 7.1.1. Intra Engine Binding

As pointed out earlier, all benchmark Web Services are implemented as WS-BPEL processes. When the service invoker calls those Web Services, it uses standard SOAP/HTTP to do this. Since the called Web Services are deployed into and managed by the SWoM, it is possible to bypass this processing and simply hand over the information to the navigator façade bean using the provided local or remote interface.

Listing 7.1 illustrates the definitions that the process modeler must supply in the SPDD to exploit intra engine bindings for a particular invoke activity.

All activity related information in the SPDD is provided via the `activityOptions` element with the activity specific information specified via the `activity` element. The appropriate activity is identified via the `name` attribute.

```
1 <executionOptions>
2   <activityOptions>
3     <activity name="C">
4       <swomProcess>LOCAL</swomProcess>
5       <swomProcessModelName>ProcessB
6     </swomProcessModelName>
```

```

7         <swomProcessActivityName>receive
8         </swomProcessActivityName>
9     </activity>
10 </activityOptions>
11 </executionOptions>

```

Listing 7.1: Intra Engine Binding

Three elements are needed so that the service invoker can construct the message that the façade bean builds when calling the navigator. `swomProcess` (Line 4) indicates that the invoked Web Service represents a SWoM process and that it can be called via the local interface of the navigator façade bean (the call in this case, via the local interface, is just an ordinary Java method call). Other options are `REMOTE` if the process is carried out on another application server. The `swomProcessModelName` and `swomProcessActivityName` elements tell the service invoker, which process model is to be used and the activity that implements the operation which is called. This information is needed to obtain the PMID identifying the process model and AID identifying the activity that is the target of the operation; this information is normally generated into the façade bean during process deployment.

Figure 7.10 shows that the usage of intra engine bindings improves performance by almost a factor of 2. No analysis has been carried out to determine how much each of the involved components, such as the façade bean or IBM WebSphere with its JAX-WS and DOM processing, originally contributes to the CPU cycles used.

It should be noted that the invoked Web Services are rather simple processes; the achievable performance improvements are expected to be lower for more complex processes.

Note, that the intra engine binding has the additional property that the execution of a Web Service is carried out within the service invoker transaction. The transaction established by the service invoker is carried forward to the navigator façade bean who then joins the transaction (the corresponding transaction attribute is set to `TransactionAttributeType.REQUIRED`).

### 7.1.2. Inline Invocation

The navigator requests, as illustrated earlier, the invocation of a Web Service by sending a message to the service invoker, which then reads the message and performs the appropriate SOAP/HTTP call. Each invocation of a Web Service is carried out in an appropriate transaction. This transaction can be eliminated by having the navigator call the service invoker. This execution mode is called *inline invocation*.

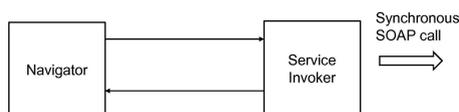


Figure 7.1.: Inline Invocation for Synchronous Invocation

For this approach, the service invoker is extended via an appropriate local/remote EJB interface, which is called by the navigator instead of inserting a message into the service invoker's input queue. The transaction participation property of the service invoker is set to `SUPPORTS`, so that the service invoker joins the navigator transaction when called by the navigator.

Figure 7.1 illustrates this processing for the synchronous invocation type. As can be seen, the navigator invokes the service invoker and waits until the service invoker returns. This processing saves two transactions: the transaction that the service invoker needs for calling the Web Service and the transaction that the navigator needs for processing the response from the service invoker.

```
1 <executionOptions>
2 <activityOptions>
3 <activity name="H">
4 <invocationMode>SYNCHRONOUS_INLINE</invocationMode>
5 <swomProcess>LOCAL</swomProcess>
6 </activity>
7 </activityOptions>
8 </executionOptions>
```

Listing 7.2: Inline Invocation of Synchronous Web Service

Listing 7.2 shows how one defines in the process deployment descriptor that inline invocation should be used for invoking a synchronous Web Service.

Figure 7.2 shows the invocation of an asynchronous Web Service in inline invocation mode. In this case only the service invoker transaction is saved.

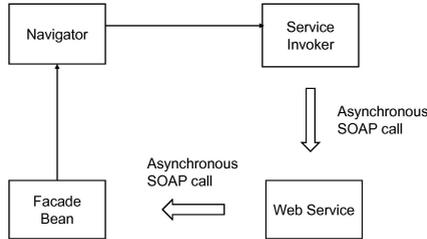


Figure 7.2.: Inline Invocation for Asynchronous Invocation

Listing 7.3 shows how one defines in the process deployment descriptor that inline invocation should be used for invoking a synchronous Web Service.

```
1 <executionOptions>
2 <activityOptions>
3 <activity name="H">
4 <invocationMode>ASYNCHRONOUS_INLINE
5 </invocationMode>
6 <swomProcess>LOCAL
7 </swomProcess>
8 </activity>
9 </activityOptions>
10 </executionOptions>
```

Listing 7.3: Inline Invocation of Asynchronous Web Service

The inline invocation approach not only reduces the number of transactions but also the number of SQL calls, as shown in Figure 7.3, for the ultimate transaction flow type. The appropriate performance improvement amounts to 2% as illustrated in Figure 7.10.

The approach has two disadvantages. The first one is the increased length of the navigator transaction, in particular for the synchronous case, where the navigator needs to wait until the invoked Web Service completes. This can result in an increased number of active navigator instances that are needed,

	Nav TX	SI TX	SQL calls	Batch calls	Calls in Batch
Ultimate	4	4	14	3	13
Direct Invocation	3	2	12	2	10

Figure 7.3.: Transaction and SQL Calls Savings for Inline Invocation

and if no more are available in the session pool, in the delay of processing requests that come from the outside (the navigator requests delivered via the queue are no problem, since they just stay in the queue until they are processed). Note that the system optimizer introduced later in Section 9.1 helps finding the proper IBM WebSphere setting so that this situation should occur less frequently.

The second one is a very subtle problem that may occur. If the service invoker invokes a Web Service, which calls back rather quickly (as is the case of the benchmark), the navigator instance that submitted the request is still active and IBM WebSphere hands over the response to this navigator instance. Appropriate tests have shown this behavior for activities C and D. The original navigator instance starts processing the received request while it is still in transaction termination processing. IBM WebSphere handles the situation by raising an appropriate error. Why IBM WebSphere actually starts a stateless session bean while the session bean is still in post-processing is unclear. It is the responsibility of the process modeler to determine whether this situation can happen, and if it can, direct invocation must not be specified. In fact, direct invocation should only be specified for synchronous and fire-and-forget invocation of a Web Service. This is the reason that the main process model still needs two service invoker transactions.

### 7.1.3. Microflow

Section 3.5 shows the normal processing that the SWoM carries out when the process model is exposed as a single synchronous Web Service.

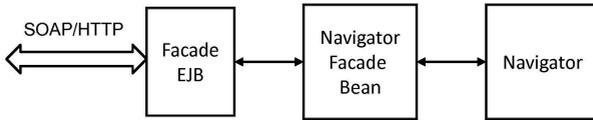


Figure 7.4.: Microflow Processing

This processing can be significantly simplified with better performance if a process model can be processed as a single transaction. Either the process really consists of a single transaction only, or a single transaction can be achieved by using inline invocation. In this case, there is no need for the navigator to respond to the façade bean via the reply queue. The façade bean calls the navigator using a special interface that returns the result directly, as shown in Figure 7.4.

The information that a process model should be executed as a microflow is specified in the SPDD as shown in Listing 7.4.

```
1 <processDeploymentDescriptor processModel="ProcessB"
2 <executionOptions>
3 <baseOptions>
4 <microflow>YES</microflow>
5 </baseOptions>
6 </executionOptions>
7 </processDeploymentDescriptor>
```

Listing 7.4: Microflow Definition

Valid entries for the `microflow` element are `NO` indicating that the process should be carried out as a normal process, and `YES` that the process is a microflow.

The algorithm for processing a micro flow is the same as described for the ultimate transaction flow type shown in Section 6.3.3.

Figure 7.5 shows the performance improvements that can be achieved for the Web Service that activity H invokes.

	Processes/sec	Improvement (%) wrt Ultimate	Improvement (%) relative
Short	107	- 27 %	
Ultimate	142	-	24.6 %
Microflow	153	7.7 %	7.7 %
Optimized	154	8.5 %	0.6 %
Compiled	155	9.1 %	0.6 %

Figure 7.5.: Microflow Performance Improvements

The performance test is run using soapUI in the same setup as the one used for the benchmark and the calibration test shown in the introduction. The soapUI client is simulating thirty requests with no wait time, driving the CPU to to saturation at 97 % utilization. The full optimized version is run for comparison only. As one can see, there is not much that can be gained through additional optimization, an experience that has already been discussed in the introduction.

#### 7.1.4. Correlation Caching

Section 3.9.4.2 illustrated conceptually how correlation processing works. This processing has been modified with the implementation of the transaction cache to take advantage of the functions delivered by the transaction cache (as partly illustrated by Figure 7.6).

When the navigator processes an invoke activity and the activity is associated with a correlation set, it calls the correlation manager to create the correlation set in the transaction cache. When the transaction finishes, the batch update facility of the transaction cache writes the correlation set information into the correlation table.

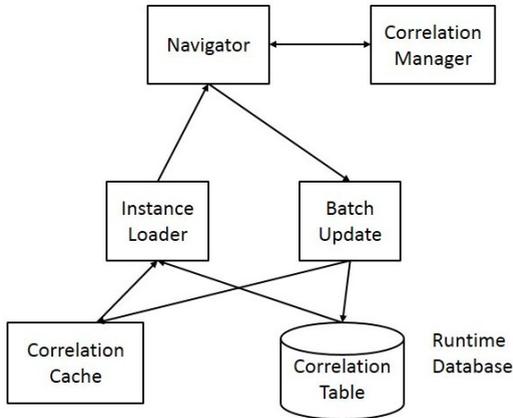


Figure 7.6.: Correlation Cache

When the navigator processes a message for a receive activity that is associated with correlation information, it calls the correlation manager to obtain the correlation information from the message and store it in the transaction cache, and then calls the instance loader of the transaction cache to locate the process instance by accessing the correlation table with the correlation value determined by the correlation manager.

Processing is now augmented by a correlation cache, which keeps correlation information and associated process instance identifiers in memory. The correlation cache follows the same architecture as all the other caches. It is implemented as a singleton; using the Java language synchronized method identifier ensures that the cache is not corrupted during update, delete, and insert operations.

When the batch update facility writes the correlation information to the database, it stores it also in the correlation cache; the instance loader now looks first in the correlation cache and if found, no longer needs to query for the correlation information in the database. When a process instance terminates, the appropriate correlation information is removed from the correlation cache.

```

1     <systemDeploymentDescriptor>
2         <correlationCache>
3             <technique>LRU</technique>
4             <entries>10000</entries>
5         </correlationCache>
6     </systemDeploymentDescriptor>

```

Listing 7.5: Correlation Cache Properties

The characteristics of the correlation cache are defined similar to other caches as shown in Listing 7.5.

Obviously, caching correlation sets only makes sense for those process models and their correlation sets where the response time of the invoked Web Service is shorter than the life time of entries in the cache. Thus the process modeler must tell the SWoM via the Process Deployment Descriptor which correlation sets should be cached and which not.

Listing 7.6 illustrates how one activates correlation set caching for the benchmark process model, where caching definitely makes sense, since the called Web Services respond immediately.

```

1     <executionOptions>
2         <cachingOptions>
3             <correlationCacheSupport>YES</correlationCacheSupport>
4         </cachingOptions>
5         <correlationSetOptions>
6             <correlationSet name="correlation1">
7                 <caching>YES</caching>
8             </correlationSet>
9             <correlationSet name="correlation2">
10                <caching>YES</caching>
11            </correlationSet>
12        </correlationSetOptions>
13    </executionOptions>

```

Listing 7.6: Correlation Caching Specifications

The `cachingOptions` are used to define the caching options that should be applied to the associated process model. In this case, correlation cache processing is activated by specifying YES for the `correlationCacheSupport` element. Whether a particular correlation set is cached is specified individually for each correlation set, as shown in the listing.

Caching of correlation information does not reduce the number of SQL calls. The normal SQL call that retrieves the process instance root via a query that joins the process instance root with the correlation information, is replaced by a simple call to retrieve (and possibly lock) the process instance root.

The performance improvements associated with correlation caching, as shown in Figure 7.10.

## 7.2. Databases

The SWoM uses IBM DB2 extensively, so it is worthwhile to provide appropriate configuration options that help the process modeler to tune the usage of IBM DB2 in the following areas:

- The usage of *referential integrity* and *primary keys* in the runtime database.
- The usage of *BLOBs* for storing variables.
- The efficient *handling of correlations*.
- Strategies for reducing the impact of *process instance deletions*

## 7.3. Referential Integrity

The runtime database, as shown in Section 3.8.3, specifies referential integrity rules that govern the relationship between the different tuples that make up a process instance. The appropriate delete rules allow the SWoM to remove a process instance by simply deleting the appropriate entry in the process instance table.

Without referential integrity, one needs to carry out appropriate SQL delete calls for each of the different parts of a process instance. This could be done using the SQL batch update facility, so that only one call to the DBMS is needed. Still, deleting a process instance using referential integrity is most likely more efficient than deleting the object types, such as variables, separately.

Another advantage of using referential integrity is the easier maintenance of the SWoM code and the simpler addition of new functions. Furthermore,

referential integrity potentially provides greater robustness to the SWoM as programming errors caused by invalid references are discovered earlier.

The disadvantage of using referential integrity is the need for IBM DB2 to verify the specified constraint whenever an object of a process instance, such as an activity instance, is inserted or updated.

Which approach is more efficient depends on various factors, such as the efficiency of the referential integrity implementation in the DBMS, the deletion strategy of the SWoM, and the size and complexity of the process instance.

The SWoM addresses this by allocating two different sets of tables, one with referential integrity and one without. The process modeler can configure in the SPDD whether referential integrity should be used or not.

Running the benchmark with referential integrity turned off does not provide, as shown in Section 7.9, any measurable performance improvements. One can only assume that the additional processing that is needed during insert and update operations is compensated by the more efficient delete processing when the process instance is removed.

## 7.4. Primary Keys

All objects in the runtime database, such as activity, process, or variable instances, have as primary key a unique identifier that is generated by the Object Identifier generator described in Section 3.12. However from a performance objective it may be advantageous to have tables with no primary key, if the key is never used in SQL queries. An appropriate entry in the SPDD tells the navigator to use tables without primary keys; the only table that needs a primary key is the process instance table. The appropriate performance tests shown in Section 7.9 indicate that performance does not improve. More tests are needed to determine if the results are any different if the database contains many process instances.

## 7.5. Large Objects Usage

IBM DB2 provides support for Large Objects (LOB) as a special data type in two versions: CLOB for storing character data and BLOB for storing binary information. Until lately, the contents of those objects were stored in a separate file with a reference kept in the associated tuple. The LOBs were not cached nor was it possible to put an index on it. IBM DB2 V9 provides the capability to store smaller LOBs in the tuple itself; however, there are many associated limitations and drawbacks when used in the SWoM. Furthermore it still does not provide the capability of putting an index on a LOB.

The SWoM uses the data type CLOB extensively to store process context information. For example, each variable instance is stored in a CLOB, as the size is not known until the variable is used in a particular process instance.

Listing 7.7 shows a modification of the variable instance table shown in Listing 3.15. that helps improve processing of CLOBs. Instead of storing the value of a variable instance in one single CLOB, it is stored in two fields.

```
1 CREATE TABLE VARIABLE_INSTANCE_STRING_256
2 (
3     VIID          CHAR (12)          NOT NULL ,
4     PRIMARY KEY ,
5     VID          CHAR (12)          NOT NULL ,
6     PIID        CHAR (12)          NOT NULL
7     REFERENCES PROCESS_INSTANCE
8     ON DELETE CASCADE ,
9     VALUE       VARCHAR (256) ,
10    VALUE_REST  CLOB (2000k)
11 ) ;
```

Listing 7.7: Modified Variable Instance Table

The first 256 bytes of the variable value are stored in the `VALUE` field (Line 9). The rest, if the value is longer than 256 bytes, is stored in the `VALUE_REST` field (Line 10). Both fields are accessed when the variable is retrieved from the databases. If the `VALUE_REST` contains a value, the two fields are concatenated to construct the actual value of the variable. This approach is extremely efficient if most of the variables have less than 256 bytes.

The SWoM offers, to cope with a range of variable instance sizes, several sizes for the VALUE field: 256, 512, 1024, and 2048 bytes. Which one the SWoM should use for a particular activity is defined via an appropriate entry in the SPDD.

```
1 <processDeploymentDescriptor>
2   <executionOptions>
3     <variables>
4       <variable name="InRequest">
5         <length>100</length>
6       </variable>
7       <variable name="OutRequest1">
8         <length>100</length>
9       </variable>
10    </variables>
11  </executionOptions>
12 </processDeploymentDescriptor>
```

Listing 7.8: Variable Length Definitions

The process modeler specifies for each variable the most common length, and the SWoM automatically selects the most appropriate table. The benchmark has been run with setting the length of all variables to 100, since that's the length that is used when the client generates the requests. The performance improves, as shown in Section 7.9, by 1 %. Note that no further improvement should be expected from this optimization technique, since LOBs are never cached, so data must always be written to the database and brought in from there (at least in the IBM DB2 version used).

## 7.6. Correlation Handling

The correlation instance table shown in Listing 3.23 can not be implemented in IBM DB2, since IBM DB2 V9 does not support the indexing of LOB fields. One option would have been the introduction of an implementation restriction, for example limiting correlation sets to 512 bytes. We have chosen to address these problems via the following approaches:

The first approach is to move the first 256 bytes of the correlation value into a separate newly added field CORR\_BASE, (see Line 9) as shown in Listing 7.9, the same approach that has been taken for variables in the previous section.

```

1 CREATE TABLE CORRELATION_SET_INSTANCE_STRING_SHORT
2 (
3     CSIID          CHAR (12)          NOT NULL
4     PRIMARY KEY ,
5     CSID           CHAR (12)          NOT NULL ,
6     PIID           CHAR (12)          NOT NULL
7     REFERENCES PROCESS_INSTANCE
8     ON DELETE CASCADE,
9     CORR_BASE      VARCHAR (256)      NOT NULL ,
10    CORR_REST       CLOB (2000k)
11 ) ;

12 CREATE INDEX CORRELATION_SET_INSTANCE_STRING_SHORT_INDEX
13 ON CORRELATION_SET_INSTANCE_STRING_SHORT
14 (CORR_BASE, CSID, PIID) ;

```

Listing 7.9: Improved Correlation Set Instance Table

Listing 7.10 shows the code that inserts a correlation set into the correlation set table. No unique index is on the CORR\_BASE field; extra code is needed to ensure that only one correlation set instance exists with a particular correlation set value.

```

1 SELECT CORR_BASE FROM
2     CORRELATION_SET_INSTANCE_STRING_SHORT
3     WHERE (CORR_BASE = CorrelationValueFirstPart AND
4           CSID = CorrelationSetIdentifier)
5
6 if notFound then
7     INSERT INTO CORRELATION_SET_INSTANCE_STRING_SHORT
8     return
9 end if
10 if new correlation set value is shorter than 256 then
11     INSERT INTO CORRELATION_SET_INSTANCE_STRING_SHORT
12     return
13 end if
14 SELECT CORR_REST FROM
15     CORRELATION_SET_INSTANCE_STRING_SHORT
16     WHERE (CORR_BASE = CorrelationValueFirstPart AND
17           CSID = CorrelationSetIdentifier)
18
19 if the stored correlation set value is the same
20     as the new one then
21     signal error
22 end if
23 INSERT INTO CORRELATION_SET_INSTANCE_STRING_SHORT

```

Listing 7.10: Improved Correlation Insert Processing

Line 1 checks whether the database holds a correlation set instance with the given CORR\_BASE value. If not found (Line 2), no correlation set instance exists with the given value, and the new correlation set instance can be inserted. Line 6 checks whether the new correlation set value is shorter than 256 bytes (the length of CORR\_BASE). If so, no correlation set instance with the new value exists in the database and the correlation set instance can be inserted. Line 10 retrieves the CORR\_REST for the specified correlation set. This call is needed for the construction of the correlation set value in the database. Note that the CORR\_REST field could have been retrieved in the first call; however has been deferred until really needed, as fetching the CLOB causes an extra I/O. If the stored value is the same as the new value, an error is raised and processing returns to the caller for error handling (Line 11); otherwise the correlation set instance is inserted into the database.

It is obvious that the instance loader must take care of the situation that the value is no longer unique; that means multiple correlation set instances have the same value. The instance loader addresses this problem via the modified correlation processing shown in Listing 7.11.

```
1  SELECT PIID FROM
    CORRELATION_INSTANCE
    WHERE (CORR_BASE = CorrelationValueFirstPart AND
           CSID = CorrelationSetIdentifier)

2  if not found then
3      handle not found situation
4      exit
5  end if

6  if correlation value shorter 256 then
7      return PIID
8  end if

9  SELECT PIID FROM
    CORRELATION_INSTANCE
    WHERE (CORR_BASE = CorrelationValueFirstPart AND
           CORR_REST = CorrelationValueSecondPart AND
           CSID = CorrelationSetInstanceIdentifier)

10 if not found then
11     handle not found situation
```

```

12     exit
13   end if
14   return found PIID for further processing

```

### Listing 7.11: Improved Correlation Processing

Line 1 selects from the correlation set instance table those process instances where the first 256 bytes of the correlation value (correlation value first part) in the message are found in the CORR\_BASE field; the correlation set itself is identified via the CSID field.

Line 2 handles the situation that no entry is found in the correlation set table; that means no process instance can be located that is the target of the external request.

Line 6 handles the situation when the correlation set value is shorter than 256 bytes. In this case, no further actions are needed and the PIID can be returned for further processing.

Line 9 repeats the call in Line 1 with the complete correlation value as the selection criteria. Note, that one could have used the CSIID as a selection criteria; however, since the tuple is already in the database buffers, the shown approach is better from a performance point of view.

Line 10 handles the situation that no entry is found in the correlation set table; that means no process instance can be located that is the target of the external request. It is handled as described previously.

Line 14 returns the found PIID.

The second approach follows a similar strategy; however, instead of using the first part of the correlation value and storing it as a string, the correlation value is hashed using the MD5 message-digest algorithm [Ron92]. This leads to the table structure in Listing 7.12. Processing is similar to the previous approach, since the MD5 hash may not be unique.

```

1   CREATE TABLE CORRELATION_SET_INSTANCE_MD5
2   (
3     CSIID          CHAR (12)          NOT NULL
4     PRIMARY KEY ,
5     CSID           CHAR (12)          NOT NULL ,
6     PIID           CHAR (12)          NOT NULL
7     REFERENCES PROCESS_INSTANCE

```

```

8             ON DELETE CASCADE,
9     VALUE_MD5    CHAR (16) FOR BIT DATA NOT NULL ,
10    VALUE        CLOB (2000k)
11    )
12    ;

13    CREATE INDEX CORRELATION_SET_INSTANCE_MD5_INDEX
14        ON CORRELATION_SET_INSTANCE_MD5
15        (VALUE_MD5, CSID, PIID) ;

```

Listing 7.12: Improved Correlation Set Instance Table using MD5-Hashing

The advantage of the approach is the small size of the field used for the initial selection of potentially qualifying correlation sets; the disadvantage is the increase in CPU cycles needed for computing the MD5 hash.

The third approach uses a much stronger message-digesting algorithm that is collision free, such as SHA-256 [NIS02]. Listing 7.13 shows how the correlation instance table looks.

```

1     CREATE TABLE CORRELATION_SET_INSTANCE_SHA256
2     (
3         CSIID          CHAR (12)          NOT NULL
4         PRIMARY KEY ,
5         CSID          CHAR (12)          NOT NULL ,
6         PIID          CHAR (12)          NOT NULL
7         REFERENCES PROCESS_INSTANCE
8         ON DELETE CASCADE,
9         VALUE_SHA256  CHAR (32) FOR BIT DATA NOT NULL
10    ) ;

11    CREATE UNIQUE INDEX CORRELATION_SET_INSTANCE_SHA256_INDEX
12        ON CORRELATION_SET_INSTANCE_SHA256
13        (VALUE_SHA256, CSID) ;

```

Listing 7.13: Improved Correlation Set Instance Table using SHA-256-Hashing

The advantage of the approach is the simplified processing of correlation values, since only one SQL call is necessary when creating a correlation set instance and locating the process instance for a given correlation value. The disadvantage is the further increase in CPU cycles compared to MD5 hashing; whether this is compensated through the cycle savings depends on several factors, including the efficiency of the hashing code.

The hash algorithm to be used is specified via an appropriate entry in the SPDD, as shown in Listing 7.14.

```
1 <executionOptions>
2 <correlationSets>
3   <correlationSet name="correlation1">
4     <hashAlgorithm>STRING_SHORT</hashAlgorithm>
5   </correlationSet>
6 </correlationSets>
7 </executionOptions>
```

Listing 7.14: Correlation Set Definitions

All correlation set information is provided via the `correlationSetOptions` element, with each correlation set identified via the `name` attribute in the `correlationSet` element. The `hashAlgorithm` specifies the algorithm to be used; valid entries are SHA-256, MD5, `STRING_SHORT` (256 bytes), `STRING_MEDIUM` (512 bytes), and `STRING_LONG` (1024 bytes).

The problem with missing uniqueness of the `CORR_BASE` and thus the additional SQL calls that are required can be removed through the introduction of an additional specification in the SPDD, as shown in Listing 7.15.

```
1 <processDeploymentDescriptor>
2 <executionOptions>
3   <correlationSets>
4     <correlationSet name="correlation1">
5       <hashAlgorithm>STRING_SHORT</hashAlgorithm>
6       <hashUnique>YES</hashUnique>
7     </correlationSet>
8   </correlationSets>
9 </executionOptions>
10 </processDeploymentDescriptor>
```

Listing 7.15: Extended Correlation Set Definitions

The additional `hashUnique` element tells the SWoM for the `STRING` hash algorithms that the hash field uniquely identifies the correlation set.

```
1 CREATE TABLE CORRELATION_SET_INSTANCE_STRING_SHORT_UNIQUE
2 (
3   CSIID          CHAR (12)          NOT NULL
4   PRIMARY KEY ,
5   CSID          CHAR (12)          NOT NULL ,
6   PIID          CHAR (12)          NOT NULL
```

```

7          REFERENCES PROCESS_INSTANCE
8          ON DELETE CASCADE,
9          STRING_SHORT          VARCHAR (256)          NOT NULL
10     ) ;

11     CREATE UNIQUE INDEX CORRELATION_SET_INSTANCE_STRING_SHORT_UNIQUE_INDEX
12         ON CORRELATION_SET_INSTANCE_STRING_SHORT_UNIQUE
13         (STRING_SHORT, CSID) ;

```

Listing 7.16: Extended Correlation Set Definitions Table

This allows the SWoM to use the table layout shown in Listing 7.16. The unique index on the correlation set, together with the correlation set identifier simplifies, the handling the same way as for the SHA-256 encoding.

The standard setting of the SWoM is VARCHAR (256). Initial tests [Che08] show that for a single correlation set value the 256 char string solution with limited length is the fastest for values shorter than 256 and SHA-256 for correlation set values larger than 256 bytes. In any case, MD5 was the slowest in the cases tested.

The benchmark results presented in Section 7.9 show an improvement of around 1 % when running with string and limited length. It can be expected that the improvements are larger if working against a database with many process instances; IBM DB2 in this case most likely must go to the database and can not service the request from its buffers.

## 7.7. Instance Deletion Strategy

A business process is normally removed from the runtime database at completion unless there are other reasons for keeping the process instance for some time, for example, to support queries by call center people. Deleting a process instance is a performance intensive task and therefore has impact on concurrent operations against the runtime database. IBM MQSeries Workflow, for example, addresses this problem by just flagging the process instances as finished and have a dedicated component, the delete server, removing the finished process instances at user-specified times [IBM98]. The SWoM implements three strategies for deleting process instances: ongoing, completion, and

deferred.

It should be noted that the strategies have been implemented; however, no performance tests have been carried out to determine which strategy is the best in which situation. It is expected that the benchmark is not responsive to the different deletion strategies. The average number of active process instances is less than 50 at any one time with each process instance holding less than 1000 bytes. Required memory size for holding all process instances in memory is around 50 KB, which easily fits into the IBM DB2 buffers, so that IBM DB2 can carry all database processing without going to the database.

So, to simplify the test case setup, all benchmark tests are run with completion deletion mode.

### 7.7.1. Completion Deletion

The simplest deletion strategy method, from a conceptual as well as implementation viewpoint, is the deletion of a process instance at completion time. It is performed as part of the final transaction. This approach is most likely the cheapest with respect to the overall resource consumption for deleting process instances for the following two reasons. First IBM DB2 already accesses the appropriate process instances. Second, since all occurrences of the different objects are physically grouped together via a clustered index, removal of the different objects are handled by emptying the appropriate pages.

The disadvantage of this approach is the impact on other running process instances, in particular if the process instance contains many or large objects. If the overall system load is rather high or the SWoM is very busy, deleting a process instance right away may impact the processing of other process instances.

### 7.7.2. Deferred Deletion

The impact on running process instances can be reduced by deferring the deletion of process instances to a later time, when the impact can be absorbed without, or at least with tolerable, impact. The SWoM implements two distinct

approaches that control when process instances are deleted: load controlled or time controlled.

```
1 <systemDeploymentDescriptor>
2   <deleteProcessing type="loadDependent">
3     <CPULevel>80</CPULevel>
4     <levelCheckingPeriod>PT1M</levelCheckingPeriod>
5   </deleteProcessing>
6 </systemDeploymentDescriptor>
```

Listing 7.17: Load Dependent Deletion Of Process Instances

The SWoM, in the load control mode, deletes process instances only when the CPU load is below a certain level. Listing 7.17 shows the appropriate definition in the system deployment descriptor.

The `deleteProcessing` element defines the deletion strategy with the `type` attribute indicating which strategy to used; in this case the `loadDependent` deletion strategy is defined.

Two parameters control the processing: `CPULevel` and `levelCheckingPeriod`. For example, the above specification indicates that process instance deletion should be suspended when the CPU level is above 80%; the CPU level is checked every minute.

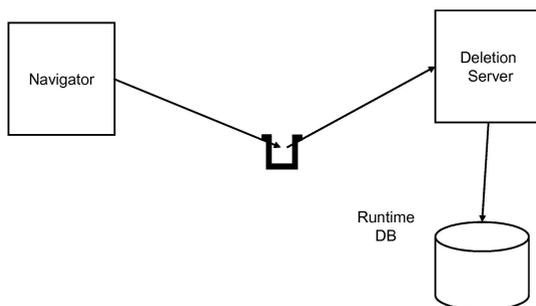


Figure 7.7.: Process Instance Deletion Server

This type of processing necessitates an addition to the architecture shown in Figure 3.1. When the navigator has completed processing of a process instance, it inserts a message with the appropriate process instance identifier into the

queue of the process instance deletion server.

The process instance deletion server reads, within a transaction, a request message and deletes the specified process instance. The server stops deleting process instances when the specified CPU load is exceeded and resumes deleting process instances when the CPU level falls below the specified one.

The second approach, for example implemented by IBM MQSeries Workflow [IBM98], deletes process instances during a certain time frame. Listing 7.18 illustrates how the definition of this approach could look.

```
1   <systemDescriptor>
2     <deleteProcessing type="timeDependent">
3       <startTime>20000</startTime>
4       <endTime>2400</endTime>
5       <deleteInterval>60</deleteInterval>
6       <deleteIdleInterval>60</deleteIdleInterval>
7     </deleteProcessing>
8   </systemDescriptor>
```

Listing 7.18: Time Dependent Deletion Of Process Instances

The elements `startTime` and `endTime` specify the time in which the process instance delete server is active. During that time, the server is active deleting process instances for the time specified in `deleteInterval` and is stopping for the time specified in `deleteIdleInterval`. The purpose of stopping deletion of process instances during the time the server is active is to cool down the database so that the impact on concurrently executing process instances is limited.

This approach does not require that the process instances to be deleted are stored in a queue. When activated, the server queries the database for completed process instances and uses the identifier of the found process instances to delete one after the other.

The off-loading of process deletion is not necessary free; in fact, additional processing is required. First, the navigator must update the process instances in the runtime database with the finished state. Second, the delete server requires additional processing. In the load-controlled mode, it can directly delete a process instance by issuing a qualified SQL DELETE call. In the time-controlled mode, it must retrieve the records first before deleting them; firing off an SQL

DELETE call for the deletion of all finished process instances may result in the locking of the table, if many records qualify.

### 7.7.3. Ongoing Deletion

In this mode, information that is no longer needed is immediately removed from the runtime database, for example, variables that are no longer used or activities and links that are no longer needed.

An appropriate definition the SPDD for a process model could look like the one in Listing 7.19. It specifies that all activity as well as variable information is removed as soon as it is no longer needed.

```
1 <processModelDescriptor>
2   <deleteProcessing type="ongoing">
3     <objects>
4       <object>activities</object>
5       <object>variables</object>
6     </objects>
7   </deleteProcessing>
8 </processModelDescriptor>
```

Listing 7.19: Ongoing deletion of process instance information

The `deleteProcessing` element defines the deletion strategy with the `type` attribute indicating which strategy to used; in this case the `ongoing` deletion strategy is defined.

The `objects` element contains the definition of the individual objects that should be deleted when no longer in use; the individual objects are identified via the `objects` element.

This strategy is characterized by two advantages over both other alternatives. First, most of the deletion processing is spread over many transactions rather than being handled in one transaction. Second, and this is more important, the runtime database contains only as much information as is required, possibly reducing significantly the size of the runtime database. Unfortunately, the efficiency of ongoing deletion is rather low, since the navigator normally does not have enough information to do a decent job. This is completely different with the introduction of the flow optimizer in the following chapter; the flow

optimizer can precisely determine when an object is no longer needed, and therefore can be removed.

## 7.8. Flow Compilation for Microflows

The process execution components, navigator and service invoker, carry out microflows by interpreting the process model (and the flow execution plan as discussed later). The performance can be improved by having the user provide Java code that performs the same functions as would the microflow.

It should be noted that the usage of user-supplied code for processes that are carried out in multiple navigator transactions is not supported. It is virtually impossible to write the appropriate Java code by hand, since one would need to call the internal cache to manage the variables, correlation sets, and activities. This code can only be generated by the flow optimizer to be discussed in the following chapter. [LR00a, Pau09, HK04] show various approaches, how process models can be transformed into executable code.

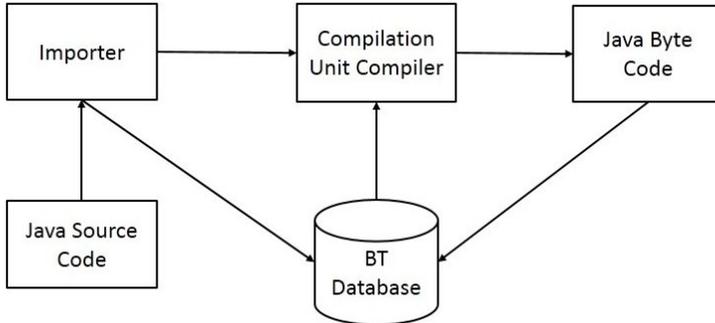


Figure 7.8.: Compilation Unit Preparation

Figure 7.8 shows the steps that are carried out to get the Java code into the buildtime database and have it compiled and stored as byte code in the buildtime database. The name *compilation unit* has been chosen to simply refer to a piece of Java code that either represents a microflow or a single navigator

transaction within a process.

Listing 7.20 shows the compilation unit for the process that is invoked by activity H in the benchmark process. The process does not carry out any function; it just returns the passed variable.

```
1 package iaas.swom.processExecution.processes;

2 import iaas.swom.processExecution.shared.CompilationUnitExecutionContext;
3 import iaas.swom.shared.exception.InternalErrorException;

4 import java.util.logging.Logger;
5 import java.util.logging.Level;

6 public class ProcessD {

7     static String className = ProcessB.class.getName();
8     public static Logger logger = Logger.getLogger(className);

9     private CompilationUnitExecutionContext compilationUnitExecutionContext;
10    private boolean traceOn;

11    public TTProcdSP(
12        CompilationUnitExecutionContext compilationUnitExecutionContext)
13        throws InternalErrorException {

14        final String methodName = "ProcessD";

15        this.compilationUnitExecutionContext = compilationUnitExecutionContext;
16        this.traceOn = compilationUnitExecutionContext.getTraceIndicator();

17        if (traceOn) {
18            logger.entering(className, methodName);
19        }
20    }

21    public void run() throws InternalErrorException {
22        compilationUnitExecutionContext.setReplyValue(
23            compilationUnitExecutionContext.getRequestValue());
24    }
25 }
```

Listing 7.20: Compilation Unit

Navigator and compilation units need to exchange information, such as variables or trace information. The `CompilationUnitExecutionContext` provides this mechanism; it is passed to the compilation unit in the appropriate con-

structor (Line 11).

The constructor of the compilation unit must save the compilation unit execution context for further usage in subsequent invocations (Line 15). It should be noted that a compilation unit is loaded when used the first time and is kept loaded in the compilation unit cache until the navigator instance finishes, causing the compilation unit cache to terminate.

The actual function that the compilation unit implements must be provided in the run method (Line 21). Processing within the process is rather simple, it just returns the value that has been passed to it. Line 23 obtains the passed value from the `CompilationUnitExecutionContext`; Line 22 then stores this value as return value, which is then returned to the navigator for returning it to the original invoker.

If the compilation unit detects an error, it must throw a SWoM specific exception, for example `InternalErrorException`, signaling an internal error; the navigator catches this error and starts the normal error handling.

Figure 7.9 shows how the navigator manages compilation units. When the navigator determines that a compilation unit is available for a micro flow, it calls the compilation unit cache manager to execute the run method of the appropriate compilation unit. Note that the compilation unit cache manager is implemented as any other cache manager.

If the compilation unit has not been loaded so far, the compilation unit cache gets the byte code from the model manager, prepares it for invocation, and stores it in the cache.

The performance improvements, as shown in Figure 7.5, are rather small. Further instrumentation of the SWoM code is needed to understand exactly why the performance improvement is so small. For now, one can only speculate why. One of the reasons could be that dynamic loading and running general classes in IBM WebSphere is less efficient than loading and running static classes. Another could be that the cycles used by the SWoM navigator are small compared to all the other cycles that are required for processing a Web Service request; evidence may the performance achievements that are achieved by exploiting intra engine binding.

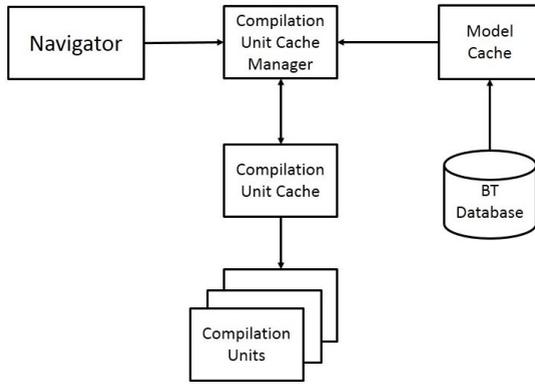


Figure 7.9.: Compilation Unit Execution

## 7.9. Performance Improvements

The following figure summarizes the performance improvements that can be achieved with some of the described flow configuration approaches.

ID	Configuration Option	Processes/ min	Total Processes/ min	Improve- ment (%)
1	Ultimate with opt WS	1143	5715	0
2	Intra engine binding	3120	15600	173
3	Inline invocation	1166	5830	2
4	Correlation caching	1150	5750	0.5
5	No primary key	1140	5700	0
6	No foreign key	1142	5710	0
7	Correlation - LOB	1154	5570	1
8	Variables - LOB	1153	5765	1
9	All (Except intra engine binding)	1195	5975	4.5

Figure 7.10.: Performance Improvements

All test cases are run using the benchmark process with ultimate transaction flow type. The invoked Web Service have been optimized via the flow optimizer introduced in the next chapter. This was done to reduce their impact on the overall execution of the process and thus provide a more accurate picture of the improvements that the various optimization techniques deliver.

The final figure in the table shows the performance improvement that can be achieved when all configuration options (except intra engine binding) have been set properly. The *Processes/min* column shows the number of main processes (ProcessA) to be carried out; the column *Total Processes/min* the total number of processes that are executed (the main process and the four called Web Services). The *Improvement* column shows the improvement with respect to the figures shown in *ID 1*. The performance improvements are as expected fairly moderate; further performance improvements can only be achieved through the introduction of a flow optimizer; see the next chapter for details.

## 7.10. Statistics Manager

The performance improvements shown so far can only be realized if the process modeler's assumptions are correct. In fact, if the assumptions of the process modeler are way off, it is quite possible that performance actually degrades.

The statistics manager, shown in Figure 7.11, helps the process modeler in the configuration task by providing accurate information about the properties that are input to the flow configuration properties, such as the length of variables. The statistics manager is implemented, as all other cache managers, as a singleton. Problems arising from concurrent requests by navigator or service invoker instances are avoided by serializing the requests through appropriate Java mechanism.

The collection of statistical information by the navigator and the service invoker, the transmission of the collected information to the statistics manager, and the storing of statistical information in the statistics database is controlled by appropriate settings in the SSDDA, as shown in Listing 7.21. Note that the statistical information to be collected is specified via the `statisticsOptions` in

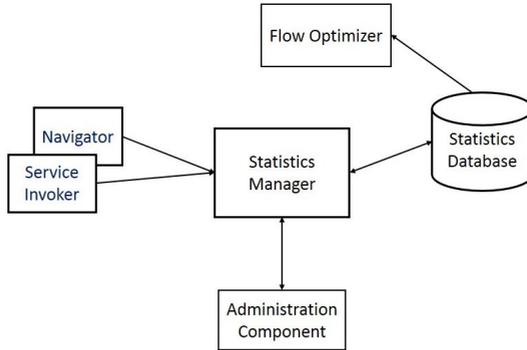


Figure 7.11.: Statistics Manager

the system deployment descriptor, as shown in in Section 3.7.1.

```

1  <systemDeploymentDescriptor>
2    <statisticsManagementOptions>
3      <persistence>YES
4      </persistence>
5      <collectionIntervalCount>1000
6      </collectionIntervalCount>
7      <collectionIntervalTime>PT2H
8      </collectionIntervalTime>
9      <recordAfterTransactions>100
10     </recordAfterTransactions>
11     </statisticsManagementOptions>
12  </systemDeploymentDescriptor>
  
```

Listing 7.21: Statistics Manager Options

The persistence element defines whether or not the collected information is written to the statistics database described later. If not, the information is just collected in memory and disposed when either IBM WebSphere or the statistics manager terminates. This mode is fairly sufficient to have a process modeler get some feeling about the settings of some of the configuration parameters. The SWoM makes the information available via its administration interface.

The recordAfterTransaction defines the number of transactions a navigator or service invoker instance must have processed before the collected statistical

information is handed over to the statistics manager. Retaining the statistical information for some time and not delivering it to the statistics manager significantly reduces the CPU consumption by eliminating the cycles needed for communication between the requesting components and the statistics manager. All statistics manager requests need to be serialized (which causes the serialization of the requesting transactions), so sending batches significantly reduces the impact of serialization. The navigator and service invoker instances, in any case, send the information to the statistics manager when they are terminated by IBM WebSphere.

The `collectionIntervalCount` and `collectionIntervalTime` elements tell the statistics manager when the collected information should be written to the statistics database. The information is written when either the number of requests exceeds the number specified in the `collectionIntervalCount` or when the duration between the last push to the database exceeds the duration specified in `collectionIntervalTime`.

Listing 7.22 shows the basic structure of the statistics database. The root table `PROCESS_MODEL` holds one tuple for each of the process models for which statistical information is collected. The process model is identified via the process model identifier stored in the `PMID` field. The `NAME` holds the name of the process model; this information is needed if the process model has been removed from the buildtime database and therefore the contents of the `PMID` field are no longer valid.

```

1 CREATE TABLE SWOM.PROCESS_MODEL (
2     PMID          CHAR(12)          NOT NULL
3     PRIMARY KEY ,
4     NAME          VARCHAR (255)     NOT NULL
5 ) IN STATISTICSTS ;

6 ALTER TABLE SWOM.PROCESSMODEL VOLATILE ;

7 CREATE TABLE SWOM.VARIABLES (
8     VID          CHAR(12)          NOT NULL ,
9     NAME         VARCHAR(256)     NOT NULL ,
10    PMID         CHAR(12)          NOT NULL
11    REFERENCES SWOM.PROCESS_MODEL
12    ON DELETE CASCADE,
13    START_TIME   TIMESTAMP        NOT NULL,

```

```

11         END_TIME          TIMESTAMP          NOT NULL,
12         MIN                INTEGER             NOT NULL,
13         MAX                INTEGER             NOT NULL,
14         AVG                INTEGER             NOT NULL,
15         COUNT              INTEGER             NOT NULL

16     ) IN STATISTICSTS;

17 ALTER TABLE SWOM.VARIABLES VOLATILE ;

18 CREATE INDEX SWOM.CLPMDONVARS
19     ON SWOM.VARIABLES (PMID ASC, AID ASC)
20     PCTFREE 10
21     CLUSTER MINPCTUSED 10;

```

Listing 7.22: Statistics Database (excerpt)

A separate table is maintained for each object, such as variables, for which statistical information is collected. When the statistics manager writes information for an object, it inserts a tuple into the proper tables when the collection intervals have been exceeded or the system administrator has requested the flushing of the statistics to the statistics database.

The table `SWOM.VARIABLES` is the table which keeps the information about a particular variable. The `VID` field identifies the variable; the `NAME` field is used to identify the process model, in case the `PMID` is no longer valid, as the process model has been removed. The `START_TIME` and `END_TIME` fields define the time interval that the tuple is associated with. The `COUNT` contains the number of variables that this tuple represents.

The actual values, in this case the lengths of the variables, are provided in the rest of the fields: the `AVG` contains the average length, the `MIN` and `MAX` fields the average of ten percent of the lowest and highest values (used to get a feeling about the distribution of the values).

# FLOW OPTIMIZATION

This chapter presents a flow optimizer that not only automatically generates configuration options but also introduces a set of new optimization techniques that improve the performance of the SWoM. Those new processing capabilities are:

- The construction of transaction flows for optimized navigation processing.
- The computation of the execution sequence of the activities within a transaction for improved transaction internal navigation.
- The automatic and correct generation of flow configuration properties, such as the length of variables or the usage of inline invocation.
- The usage of an intra transaction cache that allows to keep a transaction cache across transaction boundaries eliminating the need to reconstruct the transaction cache from information stored in the runtime database.
- The replacement of simple XPath expression processing in transition conditions, assign activities, and correlation sets through more efficient String based algorithms.

- The optimized database access of variables/correlation sets by analyzing the life cycle of variables/correlation sets through appropriate data flow analysis.
- The support of cache persistence as an alternate persistence option to the current one; instead of storing the individual pieces in the different runtime database tables, the complete transaction cache is persisted as a single object.
- The restructuring of process models with parallel paths, if possible, into a linear structure that can be executed more efficiently.

## 8.1. Flow Optimizer

The performance improvements that can be achieved through selection of the proper transaction flow type and flow configuration options are realized by having the process modeler make the correct selection of transaction flow type and flow configuration parameters. The process modeler can verify the correctness of some of the flow configuration properties by analyzing the values determined by the statistics manager.

The flow optimizer is a new SWoM component, that provides performance improvements. It runs automatically as part of the SWoM import processing to generate an initial flow execution plan (FEP). This flow execution plan contains all information that the navigator needs to optimally carry out a process instance.

The execution of the flow optimizer can also be requested through a special administrative request, usually initiated by a system administrator after a particular process model has been running for some time and statistical information is available for making more intelligent decisions.

## 8.2. Flow Execution Plan

Figure 8.1 shows the basic structure of the flow execution plan. It consists of two major parts: (1) the transaction flow with its transactions and (2) the

execution properties of the process model.

### Flow Execution Plan

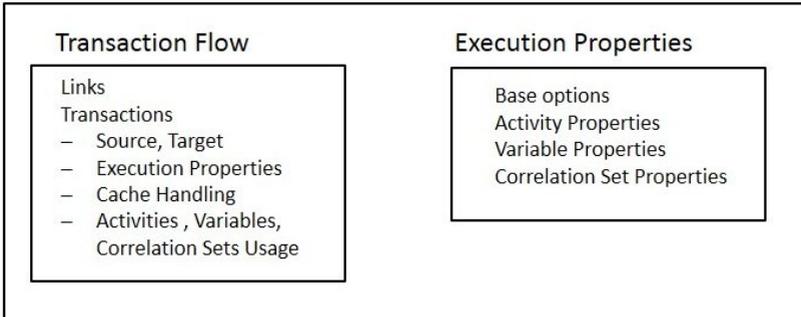


Figure 8.1.: Flow Execution Plan

The transaction flow is described as a set of transactions and appropriate links that connect the transactions. The mechanism for connecting transactions is basically the same as prescribed in the WS-BPEL specifications; the links are referenced as source or target in the transactions. Links are also associated with transition conditions, that have been factored out from appropriate transition conditions in the process model.

The execution properties are made up of two distinct sets: the base options for the process model, such as whether correlation caching should be exploited or not, or whether the process model represents a microflow, and the properties of the individual objects making up the process model, such as the properties of the variables.

The number of FEP parameters is rather large, so they are presented when the appropriate concept/technology/performance option is presented (rather than presenting the complete list here).

## 8.3. Transaction Flow Construction

The construction of the transaction flow is the first step in building the flow execution plan. As can be seen in Listing 8.1, constructing the transaction flow is carried out in seven major steps. The first step is the construction of the transactions that make up the process model. Next, the transactions are connected for those activities that are carried out in two phases, such as receive, wait, pick, or synchronous invoke activities. The third step creates join links; that means links that are used by transactions that share join activities. The fourth step is the creation of standard links between transactions; links that are reflecting the links between two activities that are now part of two transactions. In the fifth step, the necessary join conditions are attached to the individual transactions. Step six determines the transactions whose processing can be joined. Finally, step seven determines whether the execution of the process may cause deadlocks between different transactions and thus require the serialized execution of the individual transaction to avoid this.

```
1   construct transactions
2   create multi-phase activity links
3   create join links
4   create standard links
5   create join conditions
6   create transaction joining
7   determine deadlock freeness
```

Listing 8.1: Transaction Flow Construction

### 8.3.1. Transactions Construction

Listing 8.2 is the top process for determining the transactions of the process model. Line 1 empties the transaction queue that holds the start activities of the transactions to be constructed. Whenever it is determined that a new transaction should start, the start activity is inserted into the transaction queue. Line 2 obtains the start activity of the process model, which is either a receive or a pick activity. Line 3 stores this activity as start activity into the transaction queue. The loop that Line 4 performs is carried out until all transactions have been created, in other words until all start activities of the individual

transactions are processed. Line 5 creates the next transaction with the start activity in the queue by calling the function `TransactionCreation` shown in Listing 8.3.

```
1     empty transaction queue
2     get process start activity
3     put process start activity into transaction queue
4     while transaction queue is not empty do
5         call CreateTransaction
6     end while
```

Listing 8.2: Transactions Construction

Function `TransactionCreation` shown in Listing 8.3 constructs the transaction including the activities that are part of the transaction. The net result is a transaction together with the activities in the sequence they are carried out. First the start activity of the transaction is obtained from the transaction queue. Line 3 creates a transaction and adds it to the list of transactions; Line 4 creates the list of activities, that is filled with the activities within the transaction. Line 5 handles the activity by calling the function `HandleActivity` shown in Listing 8.4. Line 6 determines the next set of activities using the outgoing links of the activity and stores them in the activity queue, which is then processed in Line 7. After all activities have been processed, all multi-transaction join activities are added to the list of activities by calling Listing 8.7. Multi-join activities are those join activities that can not be processed within a transaction, since the source activities of the incoming links are part of different transactions.

```
1     function CreateTransaction
2         get the start activity from the transaction queue
3         create a transaction and add it to the list of transactions
4         create list of activities
5         call HandleActivity
6         call DetermineNextSetOfActivities
7         call HandleActivityQueue
8         call HandleMultiTransactionJoinActivities
9     end function
```

Listing 8.3: Transaction Creation

Function `ActivityHandling` in Listing 8.4 handles the processing required for an activity. Line 2 checks if the activity is a join activity. If so, Line 3 tests if all incoming links have been processed within the transaction. If not, the number of incoming links is increased and the activity is added to the list of multi-transaction activities. They need to be processed separately after all other activities have been handled. Processing of the activity completes by returning to the caller. If all incoming links have entered the activity, the activity is removed from the list of multi-transaction join activities, causing the activity to be completely processed within the transaction.

```
1  function HandleActivity
2      if join activity then
3          if not all incoming links processed then
4              increase number of incoming links
5              add to list of multi-transaction join activities
6              return
7          else
8              remove from list of multi-transaction join activities
9          end if
10         if activity is receive activity and in phase creation then
11             insert activity as start activity in transaction queue
12             set activity to phase execution
13             set activity not finished
14         else
15             set activity finished
16         end if
17         add activity to list of activities in transaction
18 end function
```

Listing 8.4: Activity Handling

Line 10 starts the actual processing of the activity. First it is checked whether the activity is a receive activity. Receive activities (and all other activities that are multi-phase activities, such as wait, synchronous invoke, and pick) need special treatment, since they can not be processed within a single transaction; in fact they constitute a transaction boundary as pointed out in Section 6.1. In the first phase of a receive activity, the creation phase, the activity is created (and stored in the runtime database); in the second phase, the execution phase, the receive activity is actually processed, which is done in a second transaction whose execution is triggered by message arriving at the defined end point; IBM WebSphere then dispatches the appropriate façade bean which in turn

calls the navigator for processing. Note that the start activity of the process, either a receive or a pick activity, is set to phase execution when it is handled in Listing 8.2. The phase information is maintained in the activity during execution of the algorithm. If the activity is in the creation phase, the activity is set to phase execution and inserted as start activity into the transaction queue. The activity will later become the start activity of a new transaction. Line 13 sets the state of the activity to not finished, so that the outgoing links of the activity are not processed.

Line 15 sets the activity to finished so that the set of next activities to be processed is determined. Line 17 finishes off with adding the activity to the list of activities within the transaction. Note that the list of activities represents the execution sequence of the activities within the transaction and is used by the transaction navigator discussed in Section 8.4.3.

Function `HandleActivityQueue` in Listing 8.5 processes the activities that have been stored in the activity queue by function `DetermineNextSetOfActivities` shown in Listing 8.6 which determines the next set of activities. A loop is being carried out until all activities have been processed. Line 3 reads the message from the queue; Line 4 calls function `HandleActivity` in Listing 8.4 to handle the activity. If the activity has been processed, the set of activities originating from the activity are determined in Line 6 by calling function `DetermineNextSetOfActivities` shown in Listing 8.6.

```
1  function HandleActivityQueue
2      while activity queue is not empty do
3          get message from queue
4          call HandleActivity
5          if activity finished then
6              call DetermineNextSetOfActivities
7          end if
8      end while
9  end function
```

Listing 8.5: Activity Queue Handling

Function `DetermineNextSetOfActivities` in Listing 8.6 determines the next set of activities when navigating forward from the current activity.

```
1  function DetermineNextSetOfActivities
2      for all outgoing links do
3          add target activity to activity queue
4      end for
5  end function
```

Listing 8.6: Next Set of Activities Determination

Note that for simplicity only link processing is shown and not the processing of sequence activities. The code is very simple: the target activities of all outgoing links are selected and inserted into the activity queue for later processing.

Function `HandleMultiTransactionJoinActivities` in Listing 8.7 adds the multi-transaction join activities to the end of the transaction.

```
1  function HandleMultiTransactionJoinActivities
2      if one multi-transaction join activity then
3          add activity to list of activities in transaction
4          return
5      end if
6      determine if any of the multi-transaction join activities is reachable
          from any other of the multi-transaction join activities
7      if not then
8          for all multi-transaction join activities do
9              add activity to list of activities in transaction
10             end for
11             return
12         end if
13         discard the transaction
14         create the transaction with making just the first
            multi-transaction join activity part of the transaction and
            making all other multi-transaction join activities
            start activities of a new transaction
15 end function
```

Listing 8.7: Multi-Transaction Join Activities Handling

If only one multi-transaction join activity is in the multi-transaction join activity list, then it is added as the last activity to the list of activities within the transaction. It constitutes the single end activity within the transaction. If the list contains more than one multi-transaction join activity, then Line 6

determines if any of the join activities is reachable from any of the other join activities; for example, whether there is a link between a join activity and another. In this case, the link can not be processed within the transaction, since first all links need to have entered the source activity before the second activity can be processed. If no such dependencies are available, then all multi-transaction join activities are added to the list of activities in the transaction. Otherwise, the current transaction is discarded and reprocessed using the list of multi-transaction join activities. Only the first multi-transaction join activity is then made part of the transaction; all other multi-transaction join activities become start activities of new transactions.

Running the algorithm for the benchmark process results in the creation of four transactions. The construction of the first transaction starts with locating activity A as the root activity. The activity is stored as the start activity of a transaction into the transaction queue. This queue is now processed until empty by creating the different transactions from the activities in the queue.

Function `TransactionCreation` in Listing 8.3 now gets activity A from the start activity queue and creates a transaction. Next the activity is processed in function `HandleActivity`. The activity is not a join activity nor a receive activity in phase creation, so it is processed, that means added to the list of activities. Next the set of next activities is calculated in function `DetermineSetofNextActivities`. As a result, activity B is put into the activity queue. Then processing of the activity queue starts until empty; that means until all activities within the transaction have been processed. After B has been processed, activity C and D are added to the activity queue. Finally activities E and F are processed. They are receive activities in the phase creation, so they are inserted a start activities into the transaction queue. In addition, they are added to the list of activities within the transaction. Listing 8.8 shows the information that has been collected. Note that the flow optimizer labels the transactions by appending the running number of the transaction to the letter T; so the ID of the first transaction becomes T1.

```
1   <transaction ID="T1">
2     <activitiesInTransaction>
3       <activityInTransaction name="A"/>
4       <activityInTransaction name="B"/>
```

```

5         <activityInTransaction name="C"/>
6         <activityInTransaction name="D"/>
7         <activityInTransaction name="E"/>
8         <activityInTransaction name="F"/>
9     </activitiesInTransaction>
10 </transaction>

```

Listing 8.8: First Transaction

The transaction queue now contains the two activities E and F. Listing 8.2 starts creation of the next transaction. Function `TransactionCreation` reads activity E from the transaction queue; activity E was the first activity inserted into the transaction queue. Function `HandleActivity` processes the activity just normally adding it to the list of activities in the transaction. Function `DetermineNextSetOfActivities` yields activity G to be processed by function `ActivityHandling`. The activity is a join activity so the processing of Line 2 takes place. Since not all incoming links have been processed (actually this is the first time the activity is processed), the number of incoming links is increased by one, and the activity is added to the list of multi-transaction join activities. Since the activity can not be completed, processing completion is set to false, so that no outgoing links are processed. All activities have now been processed, so it is checked if there are multi-transaction activities. Activity G is the only one in the list, so it is added to the list of activities. Listing 8.9 shows the information that has been collected for the transaction.

```

1     <transaction ID="T2">
2         <activitiesInTransaction>
3             <activityInTransaction name="E"/>
4             <activityInTransaction name="G"/>
5         </activitiesInTransaction>
6     </transaction>

```

Listing 8.9: Second Transaction

Processing of third transaction with start activity F is similar. The only difference is the processing of the join activity G. In this case, all incoming links have entered and the activity is treated as finished. This causes the next activities within the transaction queue to be selected. Since the activity is flagged as a transaction boundary, the target activity H is added to the

transaction queue. Listing 8.10 shows the information that has been collected for the transaction.

```
1 <transaction ID="T3">
2   <activitiesInTransaction>
3     <activityInTransaction name="F"/>
4     <activityInTransaction name="G"/>
5   </activitiesInTransaction>
6 </transaction>
```

Listing 8.10: Third Transaction

The fourth transaction is constructed with activity H as start activity. It is assumed that the activity, a synchronous invoke activity, is invoked inline Section 7.1.2; so it is executed in a single phase. Finally activity I is processed, which constitutes the end of the process. Listing 8.11 shows the contents of the fourth transaction.

```
1 <transaction ID="T4">
2   <activitiesInTransaction>
3     <activityInTransaction name="H"/>
4     <activityInTransaction name="I"/>
5   </activitiesInTransaction>
6 </transaction>
```

Listing 8.11: Fourth Transaction

Since the transaction queue contains no more activities, all transactions have been generated.

### 8.3.2. Multi-Phase Activity Link Creation

This step connects the different appearances of the multi-phase activities retrieve, wait, synchronous invoke, and pick. As pointed out in Section 3.10.1, those activities are carried out in two transactions; the first activity phase, the creation phase, is carried out in the first transaction with the activity constituting a transaction boundary.

```
1   for all for all transactions do
2     for all receive activities do
3       if activity is in the creation phase then
4         for all transactions
5           if start activity is same activity in the execution phase then
```

```

6             create link
7             create target element with link in transaction
8             create source element with link in source transaction
9         end if
10    end for
11 end if
12 end for
13 end for

```

Listing 8.12: Multi-Phase Activity Links Creation

The second activity phase, the execution phase, is carried out in a second transaction with the activity being the start activity. The purpose of the code shown in Listing 8.12 is to construct the link between the two transactions the activity is part of. Note, that the illustration is for receive activities only; the code is identical for the other multi-phase activities.

Line 1 starts a loop over all transactions. Line 2 locates the receive activities within the transaction and if one is found that is in the creation phase (Line 3), Line 4 runs through all transactions. If the start activity of a transactions is the same activity in the execution phase, Line 6 through Line 8 constructs a multi-phase activity link, the source element in the transaction holding the receive activity in the creation phase, and the target element in the transaction holding the receive activity in the execution phase.

When the algorithm is applied to the benchmark process, the following actions are carried out. The loop over the four transactions discovers the two receive activities E and F in transaction T1, that are in the creation phase.

```

1 <links>
2 <link name="MPT1T2" type="MultiPhase"/>
3 <link name="MPT1T3" type="MultiPhase"/>
4 </links>
5 <transactions>
6 <transaction ID="T1">
7 <sources>
8 <source linkName="MPT1T2"/>
9 <source linkName="MPT1T3"/>
10 </sources>
11 </transaction>
12 <transaction ID="T2">
13 <targets>
14 <target linkName="MPT1T2"/>
15 </targets>

```

```

16     </transaction>
17     <transaction ID="T3">
18         <targets>
19             <target linkName="MPT1T3"/>
20         </targets>
21     </transaction>
22 </transactions>

```

Listing 8.13: Multi-Phase Activity Links

For each activity, the start activity of all transactions are analyzed whether they represent the same activity in execution phase: activity E is found again in transaction T2, activity F in transaction T3. So two links are constructed: between transaction T1 and transaction T2 and T3, respectively. Listing 8.13 shows the result.

### 8.3.3. Join Links Creation

The links leaving multi-transaction join activities must be reflected as links between the transactions holding the different occurrences of the join activity and the transactions that hold activities which are the targets of the links leaving the join activity.

```

1   for all transactions do
2       locate multi-transaction join activities
3   end for
4   for all multi-transaction join activities do
5       create join link
6       for all outgoing links of the activity do
7           determine target activity of link
8           determine transaction that contains target activity
9           create target element for link
10          for all transactions do
11              if current activity found then
12                  create source entry for link
13                  attach transition condition to source entry
14              end if
15          end for
16      end for
17  end for

```

Listing 8.14: Join Links Creation

Line 1 starts a loop over all transactions and determines the multi-transaction join activities. Line 4 runs over all multi-transaction join activities. First, a join link is created and added to the transaction flow. Next all outgoing links of the activity are processed. For each outgoing link, the target activity and the associated transaction is determined. An appropriate target element that references the link is added to the transaction. Next, all transactions are scanned to determine those transactions where the activity is specified is specified. If found, an appropriate source element is created in the transaction. If the activity has an associated transition condition, it is copied to the transactions source element.

When the algorithm is applied to the benchmark process, the following actions are carried out. The loop over the four transactions discovers the multi-transaction activity G. An appropriate join link JLG is created.

```
1   <links>
2     <link name="JLG" type="join" activity="G"/>
3   </links>
4   <transactions>
5     <transaction ID="T2">
6       <sources>
7         <source linkName="JLG"/>
8       </sources>
9     </transaction>
10    <transaction ID="T3">
11      <sources>
12        <source linkName="JLG"/>
13      </sources>
14    </transaction>
15    <transaction ID="T4">
16      <targets>
17        <target linkName="JLG"/>
18      </targets>
19    </transaction>
20  </transactions>
```

Listing 8.15: Join Links

The only outgoing link points to activity H in transaction T4. An appropriate target element referencing the link is created in this transaction. Finally, all transactions are scanned for the multi-transaction join activity G, which yields transactions T2 and T3. A source element referencing the link is added to both

transactions. No transition condition is associated with the join activity, so no further actions are needed. Listing 8.15 shows the result.

Note that a join link has the property that the link will be followed by the transaction flow navigator when the last transaction associated with the join link completes. In the benchmark process the join link is followed when transaction T2 completes and transaction T3 has already completed or when transaction T3 completes and transaction T3 has already completed.

#### 8.3.4. Standard Link Creation

If two activities that are connected via a link become part of two transactions, then a link must be drawn from the transaction holding the first activity to the transaction holding the second activity. The appropriate processing is shown in Listing 8.16.

```
1   for all transactions do
2     locate start activity as target activity
3     if start activity has incoming links do
4       for all incoming links do
5         locate source activity
6         create link
7         create source entry with link for source transaction
8         create target entry with link for target transaction
9         attach transition condition to source entry
10      end for
11    end if
12  end for
```

Listing 8.16: Standard Links Creation

Line 1 starts a loop over all transactions; Line 2 locates the start activity. Note, that a transaction has only one start activity. If the start activity has incoming links, then Line 4 starts a loop over all incoming links. For each link, the source activity associated with the link is determined. A link is constructed between the two transactions. If the source activity has an attached transition condition, then Line 9 copies the transaction condition to the newly created source entry of the transaction.

### 8.3.5. Join Condition Creation

This step adds the join condition to the transactions. This is fairly trivial as shown in Listing 8.17.

```
1  for all transactions
2      locate start activity as target activity
3      if start activity has incoming links then
4          copy the join condition of the activity to the transaction
5      end if
6  end for
```

Listing 8.17: Join Conditions Creation

The join conditions of the start activity are made the join conditions of the transaction. This is facilitated through the usage of join links, as they appear as only one link on the target side.

### 8.3.6. Transaction Chaining

The transaction flow type `ULTIMATE` is designed to reduce the number of transactions by having a subsequent transaction join the current transaction. This is possible for transactions that have a multi-transaction join activity. The situation is reflected in Section 6.1.5.

Listing 8.18 illustrates how those transactions can be found for whom the subsequent transaction can be joined.

```
1  for all transactions do
2      if transaction has one link AND the link is a join link then
3          for all transactions
4              if transaction is source of link then
5                  set transaction as joinable
6              end if
7          end for
8      end if
9  end for
```

Listing 8.18: Transaction Joining Creation

Line 1 starts a loop over all transactions. Line 2 determines if the transaction has one incoming link and this link is a join link.

```
1  <transactions>
2      <transaction ID="T2">
```

```

3         <executionProperties>
4             <joinTransaction>YES</joinTransaction>
5         </executionProperties>
6     </transaction>
7     <transaction ID="T3">
8         <executionProperties>
9             <joinTransaction>YES</joinTransaction>
10        </executionProperties>
11    </transaction>
12 </transactions>

```

Listing 8.19: Transaction Joins

In this case, the source transaction of the transaction can join the transaction, which is indicated by the setting shown in Line 5.

Listing 8.19 shows the result of carrying out the algorithm. Only one transaction, the fourth transaction T4 has one incoming join link. The link originated in the second and third transaction, so both transactions are flagged as joinable.

### 8.3.7. Deadlock Freeness Analysis

The SWoM serializes the execution of parallel transactions as shown in Section 6.3.1 to avoid deadlocks, that may occur if parallel transactions process the same set of activities or variables.

Determining which transaction can be processed in parallel is handled in a two-phase process. First the transaction flow is converted to a sequence of transaction scopes. A transaction scope is a set of transactions that are either carried in parallel or sequential. In a second phase, the transaction scopes are analyzed with respect to the usage of variables and activities.

Listing 8.20 starts the main processing. First, the transaction queue, used for processing the transactions is created; next the transaction scope list, that holds the sequence of transaction scopes is created. As the process starts with a single transaction, a sequential transaction scope is created and added to the transaction scope list.

The number of parallel transactions is set to one and the first transaction, the start transaction, added to the transaction scope. Line 8 starts a loop over all transactions that are part of the transaction flow. For each transaction the

function `HandleTransaction` shown in Listing 8.21 is called.

```
1   create transaction queue
2   create transactionScopes list
3   create sequential transactionScope
4   add transactionScope to transactionScopes list
5   set numberParallelTransactions to one
6   add first transaction to transactionScope
7   call DetermineNextSetOfTransactions
8   while transaction queue is not empty do
9       call HandleTransaction
10  end while
```

Listing 8.20: Transaction Scope List Generation

The actual handling of the transactions, that means the allocation to a particular transaction scope is handled in Listing 8.21. The number of parallel transactions is decremented to indicate that a path has completed.

If the activity has only one incoming link, then the transaction is added to the current transaction scope, the set of outgoing links is determined by calling the function `DetermineNextSetOfActivities`, and processing completes by returning.

```
1   function HandleTransaction
2       decrement numberParallelTransactions
3       if one incoming link then
4           add transaction to transactionScope
5           call DetermineSetOfNextTransactions
6           return
7       end if
8       if not all incoming links processed then
9           increase number incoming links
10          return
11      end if
12      add transaction to transactionScope
13      if numberParallelTransactions = 0 and transactionScope is parallel then
14          set endpointReached to true
15      end if
16      call DetermineNextSetOfTransactions
17  end function
```

Listing 8.21: Transaction Handling

Line 8 determines for a join transaction if all incoming links have entered the transaction already and if not, the number of incoming links is increased, and

processing continues with the next transaction. Note, that the transaction will be delivered via the transaction queue several time until all links have entered.

Line 12 is carried out if the transaction is processable; it adds the transaction to the transaction scope. If the number of parallel transactions is zero and the current transaction scope is parallel, the end of parallel paths has been reached , and the flag `endpointReached` is set. Finally the function `DetermineNextSetOfTransactions` is called to determine the next set of transactions by following the outgoing links.

```
1  function DetermineNextSetOfTransactions
2      if no outgoing links then
3          return
4      end if
5      for all outgoing links do
6          if join link and not last link then
7              continue with next link
8          end if
9          increment numberParallelTransactions
10         add target transaction to transaction queue
11     end for
12     if endpointReached then
13         if number outgoing links = 1 then
14             create sequential transaction scope
15             set endpointReached false
16             return
17         else
18             create parallel transaction scope
19             set endpointReached false
20             return
21         end if
22     else
23         if transaction scope is sequential and number outgoing links > 1 then
24             create parallel transactionScope
25             add transactionScope to transactionScope list
26         end if
27     end if
28 end function
```

Listing 8.22: Next Set of Transactions Determination

Line 5 runs over all outgoing links. If the outgoing link is a join link and it has not been processed for all originating transactions, processing continues with the next link. Otherwise the number of parallel paths is incremented and the target transaction is added to the transaction queue for processing.

Line 12 handles the situation that an endpoint of parallel processing has been reached. If the transaction has only one outgoing link, then sequential processing starts and thus a new sequential transaction scope is created, otherwise a new parallel transactions cope is created.

Line 23 checks whether a switch from a sequential to a parallel transaction scope is needed. This will be the case if the number of outgoing links is greater than 1. Then a new parallel transaction scope is created and added to the transactionScopes list.

Whether a transaction can be executed deadlock free is indicated to the transaction flow navigator via an appropriate setting in the execution properties of the transaction as shown in Listing 8.23.

```
1 <transactions>
2   <transaction ID="T1">
3     <executionProperties>
4       <deadLockFree>YES</deadLockFree>
5     </executionProperties>
6   </transaction>
7   <transaction ID="T2">
8     <executionProperties>
9       <deadLockFree>NO</deadLockFree>
10    </executionProperties>
11  </transaction>
12  <transaction ID="T3">
13    <executionProperties>
14      <deadLockFree>NO</deadLockFree>
15    </executionProperties>
16  </transaction>
17  <transaction ID="T4">
18    <executionProperties>
19      <deadLockFree>YES</deadLockFree>
20    </executionProperties>
21  </transaction>
22 </transactions>
```

Listing 8.23: Deadlock Settings

The setting is determined in the second phase of the algorithm as shown in Listing 8.24. Line 1 loops over all transaction scopes. If the transaction scope is a sequential one, then all transactions are flagged as deadlock free.

```
1   for all transaction scopes do
2     if transaction scope is sequential then
```

```

3      for all transactions in scope do
4          set deadLockFree to yes
5      end for
6  else
7      for all transactions in transaction scope do
8          if transaction has join link attached then
9              set deadLockFree to no
10         else
11             for all variables do
12                 determine other transactions in transactions scope
13                     that use the variable
14                 analyze usage pattern
15             end if
16             if usage conflicts then
17                 set deadLockFree to no
18             else
19                 set deadLockFree to yes
20             end if
21         end if
22     end for
23 end if
end for

```

Listing 8.24: Deadlock Settings Determination

If the transaction scope is parallel, then Line 7 loops over all transactions in the transaction scope. If a transaction has a join link attached as source, then the transaction cannot run deadlock free since it uses the same activity as another transaction. Line 11 runs over all variables within the transaction. Line 12 determines the other transactions in the transaction scope that use the variable. Line 13 analyzes the usage pattern of the variable in the various transactions and determines if a conflict exists that could cause a deadlock.

Listing 8.23 shows the result when the algorithm is applied to the benchmark. Transactions T2 and T3 can not run deadlock free, since they share the join activity G; transaction T4 can, since there is no parallel transaction.

The navigator uses the deadlock information to control the access to the process instance. If `deadLockFree` is set to `NO`, the appropriate SQL call locks the processing instance by using `WITH RS USE AND KEEP UPDATE LOCKS`. The execution of parallel transactions accessing the same process instance are blocked until the transaction holding the lock completes. If `deadLockFree` is set to `YES` no lock is obtained.

## 8.4. Navigation

The pre-computation of the transaction flow and its transactions mandates changes to the navigator implementation to take advantage of the acquired information.

### 8.4.1. Enhanced Navigation Architecture

The original single-level navigator is replaced by a two-level navigator. When a request is received, either via an internal request or via a request received from a Web Service, control is passed to the *Transaction Flow Navigator*. Its first action is to call the *Request Analyzer* to inspect the request and set up internal control structures for the efficient communication between the different navigator components and other components such as the correlation manager or the transaction cache.

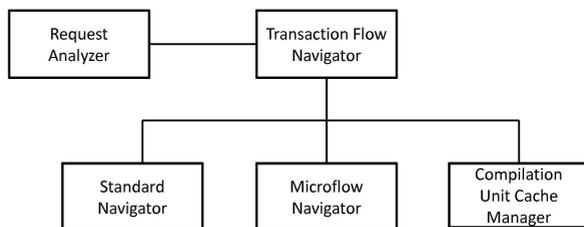


Figure 8.2.: Navigator Structure

If the request is valid, the transaction flow navigator calls the proper transaction processing component, either the *Standard Navigator*, the *Microflow Navigator*, or the *Compilation Unit Cache Manager*. The standard navigator is the original navigator that has been improved for a more efficient processing, for example by removing any processing needed for handling microflows. The microflow navigator is a very efficient version of the standard navigator; it is stripped-down by eliminating all the processing that is needed to persist

the process instance and to send off messages for service invocation or inter process navigation.

#### 8.4.2. Transaction Flow Navigator

The transaction flow navigator performs its processing in three phases: the preprocessing phase in which the request is analyzed, the transaction to be processed determined, the process instance located, and the attached join condition evaluated; the processing phase, in which the transaction is processed by the appropriate transaction navigator; and the post-processing phase, in which the set of subsequent transactions is determined.

Listing 8.25 shows the activities carried out in the preprocessing phase of the transaction flow navigator. Line 1 analyzes the request that is either coming from a façade bean or from the navigator queue. Based on this information, Line 2 locates the transaction to processed.

Line 3 locates the process instance. Depending on the setting of the `deadLockFree` element, appropriate locking is requested in the SQL call used.

Line 4 increases the number of incoming links; Line 5 increments the number of true incoming links based on the truth value of the link.

Line 6 determines if all incoming links have entered the transaction, and if so, the join condition of the transaction is checked.

```
1   analyze request
2   locate transaction
3   locate process instance
4   increase number of incoming links
5   increase true number of incoming links base on truth value of link
6   if all incoming links available then
7       check join condition
8       if join condition fails then
9           set transaction skipped
10      end if
11  else
12      return
13  end if
```

Listing 8.25: Transaction Preprocessing

If the join condition fails in Line 8, the transaction state is set to skipped. The post-processing phase will take care of this situation by performing dead-path-elimination for the follow-on transactions. If not all links have entered the transaction, control is returned, since the transaction can not yet be processed.

Listing 8.26 shows the actual processing of the transaction. Note it is only carried out when the transaction is not in state skipped. It is carried by one of the three transaction navigators. If the transaction represents a microflow or a compilation unit (that means a microflow that has been compiled), then the transaction represents the complete process and thus control is returned, as there is nothing more to be done.

```
1   call appropriate transaction navigator
2   if microflow OR compilation unit then
3       return
4   end if
```

Listing 8.26: Transaction Processing

After the transaction has completed, post processing takes place as illustrated in Listing 8.27.

Line 1 through Line 7 handle the situation that the transaction is skipped. For each outgoing link, an appropriate message for the target transaction is created, the false indicator is set, and inserted into the navigator queue.

Line 8 starts a loop over all outgoing links, that means links whose source is the current transaction. The actual processing depends on the type of link that is followed. If the outgoing link is a join link and has been set not processable by the transaction navigator, processing continues with the next link. The link is set to not processable if not all links have entered the attached multi-transaction join activity so processing of the join activity has not completed.

```
1   if transaction skipped then
2       for all outgoing links do
3           construct message for destination transaction
4           set false indicator
5           insert message into navigator queue}
6       end for
7   end if
8   for all outgoing links do
9       if multi-phase activity link then
10          continue with next link
```

```

11     end if
12     if join link not processable then
13         continue with next link
14     end if
15     construct message for destination transaction
16     if standard/join link is flagged fail then
17         set skipped indicator in message
18     else
19         if transaction condition attached to link then
20             evaluate transition condition
21             if transition condition fails then
22                 set skipped indicator in message
23             end if
24         end if
25     end if
26     insert message into navigator queue
27 end for

```

Listing 8.27: Transaction Post Processing

Line 15 constructs a message with the appropriate information about the target transaction to be inserted into the navigator queue. Next, Line 16 checks whether the link is flagged as failed. This occurs if the activity from which the link originated was skipped. If so, the skipped indicator in the message is set.

Line 18 starts the analysis of processable links by checking the associated transition condition. If it fails, the false indicator in the message is set. Finally, the message is inserted into the navigator queue to have the transaction flow navigator process the request.

### 8.4.3. Transaction Internal Processing

Both, the standard navigator and the microflow navigator use the sequence of activities, which has been constructed in Section 8.3.1 while determining the transactions that make up the transaction flow. The processing distinguishes between two different modes of operation: the unconditional execution mode and the conditional execution mode.

### 8.4.3.1. Unconditional Execution Mode

When the flow optimizer creates the transactions with the enclosed activities, it also checks whether there are any transition conditions attached to the links. If there are none, it sets the `executionMode` element to `UNCONDITIONAL` as shown in Listing 8.28.

```
1 <transactions>
2   <transaction ID="T1">
3     <executionProperties>
4       <executionMode>UNCONDITIONAL
5     </executionMode>
6   </executionProperties>
7 </transaction>
8 </transactions>
```

Listing 8.28: Unconditional Execution Mode

This allows the navigator to execute the code shown in Listing 8.29. The navigator runs over all activities defined in the transaction.

Processing of multi-transaction join activities, identified by being associated with a join link, requires the special processing shown in Line 2 through Line 21. Processing starts with Line 3 trying to get the activity from the database.

```
1   for all activities in transaction do
2     if outgoing join link references activity then
3       get activity instance from runtime database
4       if not found then
5         create activity instance
6         set number of incoming links to one
7         continue with next activity
8     else
9       increment number of incoming links
10    end if
11    if actual incoming links equal defined incoming links then
12      if join condition fails then
13        set activity instance skipped
14        call TransactionLinksProcessing
15      else
16        execute activity instance
17        set activity instance finished
18        call TransactionLinksProcessing
19      end if
20      continue with next activity
21    end if
22  else
```

```

23         create activity instance
24         execute activity instance
25         set activity instance finished
26         call TransactionLinksProcessing
27     end if
28 end for

```

Listing 8.29: Unconditional Execution Mode Navigation

If not found, that means not yet processed by any other transaction that share the activity, the activity is created, the number of incoming links set to one, and processing continues with the next activity in the activity list. Otherwise the number of incoming links of the activity is incremented by one.

Line 11 determines if all incoming links have entered the activity and if so, the activity is executed and the outgoing link is set to processable; otherwise processing continues with the next activity.

If the activity is no multi-transaction join activity, Line 23 creates the activity, Line 24 executes it, and Line 25 sets the activity to finished. After the activity has been processed, the function `TransactionLinksProcessing` shown in Listing 8.30 is invoked to handle any transaction links that reference the activity.

```

1  function TransactionLinksProcessing
2      if activity is attached to join/standard link then
3          for all join/standard links attached to activity do
4              set link processable
5              if activity skipped then
6                  set link fail
7              else
8                  set link success
9              end if
10         end if
11     end if
12 end function

```

Listing 8.30: Transaction Links Processing

The function determines if any join or standard link references the activity. Line 3 inspects all links and sets the links to processable. If the activity was skipped, the link is marked fail, otherwise it is marked success. These settings are inspected by the transaction flow navigator when carrying transaction post processing shown in Listing 8.27.

### 8.4.3.2. Conditional Execution Mode

When the flow optimizer determines that transition conditions are attached to any one of the links within the activities of the transaction, the execution mode is set to `CONDITIONAL` as shown in Listing 8.31.

```
1 <transactions>
2   <transaction ID="T1">
3     <executionProperties>
4       <executionMode>CONDITIONAL
5     </executionMode>
6   </executionProperties>
7 </transaction>
8 </transactions>
```

Listing 8.31: Conditional Execution Mode

The processing that the navigator carries out is basically the same one that is described in Section 6.3.3. Since the execution sequence of the various activities is known, processing can be simplified to the one shown in Listing 8.32.

```
1  for all activities in transaction
2    if outgoing join links reference activity then
3      get activity instance from runtime database
4      if not found then
5        create activity instance
6        set number of incoming links to one
7        continue with next activity
8      else
9        increment number of incoming links
10     end if
11     if actual incoming links less than defined incoming links then
12       continue with next activity
13     end if
14   end if
15   if activity instance does not exist then
16     create activity instance
17   end if
18   check join condition
19   if fail then
20     set activity instance skipped
21     call TransactionLinksProcessing
22     call TransactionInternalLinksProcessing
23     continue with next activity
24   end if
25   execute activity instance
26   set activity instance finished
```

```

27     call TransactionLinksProcessing
28     call TransactionInternalLinksProcessing
29 end for

```

### Listing 8.32: Conditional Execution Mode Navigation

Line 1 starts a loop over all activities in the transaction. Processing of multi transaction join activities requires the special processing shown in Line 1 through Line 14. Line 2 determines if an outgoing link references the activity and if so Line 3 tries to get the activity from the database. If not found, that means not processed so far by any other transaction that share the activity, the activity is created, the number of incoming links set to one, and processing continues with the next activity in the activity list. Otherwise the number of incoming links of the activity is incremented by one.

Line 15 checks if the activity instance has been created previously and if not, creates an instance. Next, the join condition associated with the activity is checked. If it fails, Line 20 sets the activity skipped, calls the function `TransactionLinksProcessing` to process any outgoing join/standard links and the function `TransactionInternalLinksProcessing` to handle all outgoing internal links, that means links whose target is an activity within the transaction. Processing continues with the next activity.

The activity instance is now carried out and when complete, the state is set to finished, and the functions `TransactionLinksProcessing` and `TransactionInternalLinksProcessing` are called.

Listing 8.33 shows the processing that determines the state of any outgoing link and sets the state of the target activity appropriately.

```

1  function TransactionInternalLinkProcessing
2      for all outgoing links do
3          if transition condition does not fail then
4              if target activity does not exist then
5                  create activity instance
6              end if
7              increase number true incoming links in target activity
8          end if
9      end for
10 end function

```

### Listing 8.33: Transaction Internal Links Processing

Line 2 starts a loop over all outgoing links to set the appropriate true incoming links counter in the target activities; the counter is later used to check the join condition. Line 3 checks the transition condition associated with the link; if the check condition does not fail, the number of true incoming links in the target activity is incremented.

## 8.5. Variable Usage Optimization

The instance cache described in Section 5.2 typically uses more SQL calls than actually required since the instance cache has no knowledge whether a particular variable instance exists in the runtime database. If, for example, the navigator requests a variable instance and the variable instance is not in the cache, the instance cache needs to go to the runtime database to determine if the variable instance exists.

The flow optimizer provides additional information that helps the instance cache to perform significantly better. Listing 8.34 shows the information that the flow optimizer generates for the variables in transaction T1 of the benchmark process.

```
1      <transactions>
2        <transaction ID="T1">
3          <variablesInTransaction>
4            <variableInTransaction name="inRequest">
5              <startState>NEW</startState>
6              <endState>DELETE</endState>
7            </variableInTransaction>
8            <variableInTransaction name="outRequest1">
9              <startState>NEW</startState>
10             <endState>DELETE</endState>
11           </variableInTransaction>
12           <variableInTransaction name="outRequest2">
13             <startState>NEW</startState>
14             <endState>DELETE</endState>
15           </variableInTransaction>
16         </variablesInTransaction>
17       </transaction>
18     </transactions>
```

Listing 8.34: Variable Usage Definitions

The `startState` specifies the initial state of the variable within a transaction. If the state is set to `NEW`, then the instance cache must not, when the navigator requests the variable, access the database to determine whether the variable exists.

The `endState` specifies the final state of the variable instance within the transaction. The state `DELETE` indicates that the variable instance is no longer needed in any of the subsequent transactions and can therefore be deleted from the database or, if it has been created in the transaction, no actions at all must be taken (the variable is kind of a temporary variable, something which has been proposed by IBM in an IP disclosure [IBM04]).

Applying this technique to the benchmark process reduces the number of SQL calls as shown in Figure 8.3. The first column shows the number of SQL calls. The second column shows the number of SQL batch update calls; the third column the number of SQL calls that are issued in the batch update calls. The fourth column shows the number of SQL calls that are requested by the SWoM; that is the sum of the SQL calls and the batch update calls. The last column shows the sum of SQL requests, the one issued regularly and the ones that are part of the batch update calls. The test results, listed in Section 8.12, were obtained for running the benchmark using transaction flow type `ULTIMATE` and configuration `optimized`. As can be seen, performance improves by 3.8 % with the standard database persistence. Note that the improvements for cache persistence are not related to variable optimization, as the variables are not written separately to the runtime database.

	SQL Calls	Batch Update Calls	Calls in Batch Update Calls	Total SQL Calls	Total SQL Requests
Ultimate Configured	14	2	10	16	24
Variable Usage Optimized	10	2	7	12	17

Figure 8.3.: Variable Usage Optimization Results

The flow optimizer carries out the generation of the variable usage information in several phases. The first phase shown in Listing 8.35 determines for each variable, the transactions that the variable is used in.

```
1      allocate variable usage list for defined variables
2      for all transactions do
3          for all variables in the transaction do
4              determine usage INPUT, OUTPUT, or both
5              add transaction and usage to variable
6          end for
7      end for
```

Listing 8.35: Variable Usage Optimization Phase 1

Line 1 allocates a list for all variables defined; each variable is associated with the transaction it is used. Line 2 runs through the transaction flow using the links that connect the different transactions and locates each transaction. Line 3 initiates the processing of all activities within the transaction. Line 4 determines for each variable how it is used in the transaction; Line 5 adds the transaction together with the usage to the variable.

After completion of phase 1, each variable has associated with it the sequence of transactions it is referenced.

The set of temporary variables, that means variables that are only used within a transaction, can now be easily determined using the code shown in Listing 8.36.

```
1      for all variables in the variable usage list do
2          if used in only one transaction then
3              create variableInTransaction definition
4              set start state NEW
5              set end state DELETE
6              remove variable from variable usage list
7          end if
8      end for
```

Listing 8.36: Variable Usage Optimization Phase 2

Line 1 runs over all variables in the variables usage list. If the variable is only used in one transaction, then Line 3 creates an appropriate `VariableInTransaction` definition and sets the start and end state appropriately. Line 6 removes the variable from the list, as no further processing is needed. The definitions shown in Listing 8.34 are the result of this first phase.

The flow optimizer in a third phase determines all single entry/single exit variables, that means variables that are created in one transaction and consumed in another transaction. Listing 8.37 illustrates the appropriate algorithm.

```
1   for all variables in the variable usage list do
2     if variable is define with OUTPUT in one transaction only then
3       create variableInTransaction definition for transaction
4       set start state NEW
5     endif
6     if variable is define with INPUT in one transaction only then
7       create variableInTransaction definition for transaction
8       set end state DELETE
9     endif
10  end for
```

Listing 8.37: Variable Usage Optimization Phase 3

Line 1 runs over all variables in the variables usage list. If the variable is defined with OUTPUT in one transaction only, then the variable is created in the transaction. The `variableInTransaction` element is appropriately set to the start state NEW. If the variable is defined with INPUT in one transaction only, then the variable is referenced in only one transaction and can then be deleted after subject transaction has been processed.

The next phase tries to determine the transaction in which a variable is created. Since the variable is used as output in several transactions, it may be constructed in parallel transactions. Listing 8.38 determines the sequence in which the variables are modified/created.

```
1   for all all variables in the variables usage list do
2     Construct transaction flow from the transactions that
3     reference the variable as OUTPUT
4     if transaction flow has one start transaction then
5       create VariableInTransaction definition for first transaction
6       set start state NEW
7       continue with next variable
8     endif
9     if start transactions have a common ancestor transaction
10      if the paths to this transaction are exclusive then
11        create VariableInTransaction definitions for all transactions
12        set start state NEW
13      endif
14    end for
```

Listing 8.38: Variable Usage Optimization Phase 4

Line 2 constructs a transaction flow for all transactions in which the variable is created/modified. If the transaction flow has one start transaction, then the start transaction is the one in which the variable is created, so an appropriate `variableInTransaction` element is constructed for the transaction and the start state is set to `NEW`. If the transaction flow has multiple start transactions, Line 8 determines if the transaction have a common ancestor and if so Line 9 determines if the appropriate paths to the ancestor are exclusive, that means navigation hits only one of the transactions. If so, a `variableInTransaction` element with state `NEW` is created for all transactions.

Finally phase 5 determines the transaction in which a variable can be deleted. Since the variable is referenced in several transactions, Listing 8.39 determines whether a transaction exists at which all transactions join.

```
1   for all variables in the variables list do
2     for all transactions of the variable do
3       construct reachable transactions graph
4     end for
5     build graph intersection
6     if intersection graph is not empty then
7       if intersect graph has one end transaction then
8         create variableInTransaction definition in transaction
9         set end state to DELETE
10      end if
11    end if
12  end for
```

Listing 8.39: Variable Usage Optimization Phase 5

Line 2 runs over all transactions that the variable references and constructs for each transaction a reachability graph, that means a graph that contains all following transactions. Line 5 builds the intersection of the various transaction graphs that have been built. If the intersection graph is not empty and if it has a single end transaction, then a `variableInTransaction` element is constructed for the transaction and the end state is set to `DELETE`.

It should be noted that the presented algorithm in general assumes that no transition conditions are attached to the links (as is the case in the benchmark). This approach has been used to simplify the presentation of the algorithm. Additional processing is needed to cope with transition conditions.

## 8.6. Variable Load Optimization

As shown, each variable is loaded via a separate SQL call. If multiple variables need to be loaded within a transaction, then an appropriate SQL statement can be constructed that loads multiple variables in a single SQL call. The flow optimizer generates, if the load of multiple variables is possible, appropriate information, as shown in Listing 8.40.

```
1   <variablesInTransaction>
2     <loadVariablesTogether>
3       <loadTogether>
4         <variable>inResponse1</variable>
5         <variable>inResponse2</variable>
6       </loadTogether>
7     </loadVariablesTogether>
8   </variablesInTransaction>
```

Listing 8.40: Multiple Variables Load

The `loadVariablesTogether` starts a set of definitions which variables should be loaded together. Each set is identified via the `loadTogether` element with its enclosed `variable` elements.

This optimization technique applies to the benchmark process only, if the flow optimizer is running with the transaction flow type set to `SHORT`, which results in the transactions shown in Section 6.1.2. In the transaction that contains the assign activity G, the two variables shown in Listing 8.40 are processed by the activity. Appropriate performance tests showed no measurable improvements; an explanation for the result is the fact that only one out of 33 SQLs calls is saved. Research is needed to understand the impact in fully optimized process models; also research is needed to understand whether it is possible to load different objects together, such as activities, variables, and correlation sets.

## 8.7. Correlation Set Usage Optimization

The life cycle of correlation sets is simple when comparing it to variables. They are created once and can not be modified anymore. So the optimizations that can be applied are rather limited; there is no optimization possible for the creation of correlation sets. If the correlation set is defined with `init=yes`, then the correlation set is created. If it exists, it is an error and an appropriate WS-BPEL fault needs to be thrown.

The only optimization that can be carried out is with regard to the time when a correlation set is no longer needed and can be removed. This helps in the following two cases. First, the deletion processing is set to ongoing deletion as described in Section 7.7.3. As pointed out there, precise information about when an object is needed, to make the processing efficient. Second, correlation caching is active. When the correlation set is no longer needed, it can be removed from the correlation cache making room for other entries.

Listing 8.41 shows the definitions that the flow optimizer generates for the correlation set `correlation1` within the second transaction. The correlation set has been created in the first transaction and is used by the receive activity C to find the process instance the message is targeted at.

```
1 <transactions>
2   <transaction ID="T2">
3     <correlationSetsInTransaction>
4       <correlationSetInTransaction name="correlation1">
5         <endState>DELETE</endState>
6       </correlationSetInTransaction>
7     </correlationSetsInTransaction>
8   </transaction>
9 </transactions>
```

Listing 8.41: Correlation Set Usage Definition

The name attribute of the `correlationSetInTransaction` element identifies the correlation set. The `endState` element indicates what to do with the correlation set after the transaction has been processed; `DELETE` indicates that the correlation is not needed by any of the subsequent transactions.

```

1   allocate correlation set usage list with defined correlation sets
2   for all transactions do
3       for all activities do
4           if correlation set used then
5               add transaction to correlation set
6           end if
7       end for
8   end for

```

Listing 8.42: Correlation Set Usage Optimization Phase 1

The definitions are created in a multi-phase processing. Listing 8.42 starts the process by determining the usage of the correlation set within the transactions. It only looks for those occurrences when the correlation set is used, not created.

At the end, the correlation set usage list contains for each correlation set the number of transactions where it is used. If a correlation set is used in only one transaction, then the correlation set can be deleted in the transaction; that means the end state of the correlation set is set to DELETE. It is more complex for those cases, where the correlation set is used in several transactions. Listing 8.43 shows the appropriate code.

```

1   for all correlation sets in more than one transaction do
2       for all transactions of the correlation set do
3           construct reachable transactions graph
4       end for
5       build graph intersection
6       if intersection graph is not empty then
7           if intersect graph has one end transaction then
8               create correlationSetInTransaction definition in transaction
9               set end state to DELETE
10          end if
11      end if
12  end for

```

Listing 8.43: Correlation Set Usage Optimization Phase 2

Line 1 starts the enclosed processing for all correlation sets that are referenced in more than one transaction. Line 2 performs the construction of a reachability graph for the transaction. The reachability graph contains all transactions that can be reached when continuing navigation. After the graphs have been generated for all transactions, the intersection of these graphs is built in Line 6. This graph represents the transactions that are carried out when

leaving any of the involved transactions. If the intersection graph is not empty and if it has a single end transaction, then a `correlationSetInTransaction` definition is constructed for the transaction and the end state is set to `DELETE`.

## 8.8. XPath Processing Improvement

XPath is used extensively in the WS-BPEL specifications: (1) in assign activities, for example, to specify the source as well as the target of a copy operation, at least in many cases, (2) in transition conditions associated with links, and (3) as join conditions.

Assign activities and transition conditions are using XPath queries to reference pieces of variables. Note that variables come into existence either through a message entering the process, for example, via a receive activity, or are created within an assign activity.

The evaluation of an XPath expression requires that the variable is brought (either by the requester or hidden by an appropriate implementation) into a suitable internal form, so that the XPath location expressions, including backward navigation, can be carried out. The most prominent internal structure for implementing XPath functions is Document Object Model (DOM) [W3C05]. The major problem with DOM is its performance, since it needs to load the complete document into memory before any operation on the DOM tree can be carried out [DP02].

Other approaches have been developed for carrying out XPath expressions on XML documents; however, it is unclear whether these approaches provide increased performance in all situations. The SWoM implementation uses the standard DOM-based XPath processing provided by the Java Development Kit (JDK) [ORA12].

Figure 8.4 shows the three strategies one could select for the efficient support of the XPath expressions in assign activities and transition conditions.

Strategy A maintains the variable value as an XML String, both for the internal and external representation. It should be noted that variables entering from the outside, for example via a receive activity, or exiting to the outside, for example via an invoke activity, are represented via this format. A variable,

<b>Strategy</b>	<b>Memory</b>	<b>Persistence</b>
A	Literal XML	Literal XML
B	DOM	Literal XML
C	DOM	Serialized DOM

Figure 8.4.: XPath Processing Strategies

to which an XPath expression is applied, needs to be transferred to a DOM tree before the XPath expression can be applied, and if the variable is generated, the DOM tree must be transformed into the XML String format.

Strategy B maintains the variable value internally as a DOM tree and stores the variable in the XML String format. This is best implemented by having the instance cache transform the variable to a DOM tree when the variable is loaded or created and transforming all variables that need to be persisted into the XML String format before storing them in the database.

Strategy A and B can be combined into a strategy AB, by DOMifying a variable instance only if an XPath expression needs to be applied. It can be assumed that the additional effort of maintaining two representations of the variable instance value is easily compensated by the savings in transformations.

Strategy C maintains the variable instance value internally as a DOM tree and stores the variable instance value in the database by serializing the DOM tree when written to the database and deserializing the stored Document Object Model tree when the variable instance is loaded from the database.

All three strategies have their pros and cons with respect to the efficiency of processing the XPath expressions in assign activities and transition conditions. Which one is the most efficient depends on the complexity of the XML document, the complexity of the XPath expression, the number of times a variable is touched and the number of exchanges with the outside. Strategy A, for example, is the most efficient one if only complete variables are copied in assign activities.

The flow optimizer can determine the best strategy by trying to minimize the number of DOM tree constructions and DOM tree to XML String transformations. The efficiency of strategies A, B, and the combined strategy AB can be calculated for each transaction separately, since the selection of a particular strategy has no impact on other transactions for the simple reason that all strategies start and end with the same persistence state. The efficiency of strategy C can not be calculated for each transaction independently: if a transaction finishes with persistence DOM-tree-serialized, the subsequent transaction must obtain the variable as a DOM tree. Further work is needed to determine if switching of strategies within the execution of a process instance is beneficial or not.

Even if the most efficient strategy has been selected, processing of XPath expressions is still quite expensive; most of the DOM tree transformations most likely do not go away. A major performance improvement can only be expected if the XPath processing is replaced through other mechanisms that do not require any transformations.

The flow optimizer approaches this problem by analyzing each XPath expression together with the variable schema and trying to replace XPath expressions with an appropriate simpler String-based function. The method uses a simple brute-force approach using a list of hard-coded scenarios with appropriate function replacements. If a particular expression has no replacement specified, the standard XPath processing is carried out. Listing 8.44 illustrates the approach that is being taken, constituting a framework into which new XPath replacement functions can be plugged in.

```
1   for all transactions do
2       for all assign activities do
3           for all parts of the assign activity do
4               get from specification
5               get to specification
6               if from:VariablePart and to:VariablePart then
7                   get XPath query of from specification
8                   get XPath query of to specification
9                   if both are single fields then
10                      use string function CSFSF
11                      construct definitions
12                   if variable created and not yet touched then
13                      disable initialization
```

```

14         end if
15     end if
16     end if
17 end for
18 end for
19 end for

```

Listing 8.44: XPath Expressions Replacement

Line 1 runs over the transaction flow, visiting each transaction. Line 2 processes the assign activities within the transaction with Line 3 looping over all parts of the assign activity. The next statements analyze the `from` and `to` elements to conceptually construct a matrix with all possible combinations and then determine under which condition a particular SWoM replacement functions can be used. For example, Line 10 replaces the XPath expression with the SWoM function `CSFSF`, which copies a single field to a single field, when both the `FROM` and `TO` elements have an XML Path Language query with a single field attached. Line 13 disables initialization if the variable is created in the activity, but has not been touched so far.

This approach can be used for the assign activity B in the benchmark process (Listing A.13). As can be seen, two fields are extracted from the incoming message and two variables are created with one of the fields for each variable. Several entries are generated that help the assign activity processing handler to carry out its work.

Listing 8.45 controls the initialization of variables; the element `initialize` controls whether the associated variable is initialized as specified in the appropriate variable definition in the process (see Line 2 in Listing A.9).

```

1 <variable name="outRequest1">
2   <initialize>N0</initialize>
3 </variable>
4 <variable name="outRequest2">
5   <initialize>N0</initialize>
6 </variable>

```

Listing 8.45: Variable Initialization Suppression

Initialization is normally carried out by the instance cache when the navigator requests the creation of a variable instance. Initialization is needed in those situations where the target of a copy operation is defined via an XPath

expression and the target variable does not exist. The instance cache, in this situation, constructs the variable instance by creating a DOM tree from the initialization definition and then, if needed, transforms the DOM tree into literal XML. As pointed out, Line 13 disables initialization of the two variables, since they are created in the transaction and have not been touched previously.

The actual processing of an assign activity is controlled by an appropriate entry in the FEP, as shown in Listing 8.46.

```
1      <activity name="B">
2          <copy>
3              <type>CSFSF</type>
4              <from>
5                  <variable>inRequest</variable>
6                  <part>longMessage</part>
7                  <field>Field1</field>
8              </from>
9              <to>
10                 <variable>outRequest1</variable>
11                 <part>shortMessage</part>
12                 <field>Field</field>
13                 <target>OutRequest1</target>
14                 <namespace>http://iaas.perfTest.org/datatypes</namespace>
15                 <create>YES</create>
16             </to>
17         </copy>
18         <copy>
19             <type>CSFSF</type>
20             <from>
21                 <variable>inRequest</variable>
22                 <part>longMessage</part>
23                 <field>Field2</field>
24             </from>
25             <to>
26                 <variable>outRequest2</variable>
27                 <part>shortMessage</part>
28                 <field>Field</field>
29                 <target>outRequest1</target>
30                 <namespace>http://iaas.perfTest.org/datatypes</namespace>
31                 <create>YES</create>
32             </to>
33         </copy>
34     </activity>
```

Listing 8.46: Copying Fields

Each copy in an assign activity is represented, as in the WS-BPEL definition, via a copy element. The purpose of the copy element is defined via type element; it is interpreted by the navigator and handled by the function that is associated with the type. For example, the type CSFSF indicates that the contents of a field should be copied into a field within a variable. Additional information supplied in the from and to elements helps the appropriate function to locate the source field and construct the target field and the associated variable if needed.

```
1   determine start of field value in source using
2     the name of the field provided in the field element of the from element
3   determine end of field value in source using
4     the name of the field provided in the field element of the from element
5   extract field value
6   create target variable by copying the field value to the
7     target defined in the to element
```

#### Listing 8.47: Copying Fields Function

Listing 8.47 shows the code that the navigator executes in this situation. The shown navigator function performs exactly what is specified in the assign activity (see Line 10) without using XPath.

The current approach is more like a divide-and-conquer approach, where over time more and more assign expressions are implemented as base navigator functions. If no navigator function is available for a particular situation, standard XPath processing is used. It should be noted that the flow optimizer can determine rather precisely which function is needed, since it has enough time to use the underlying schema information for analyzing the query and determine possible results. It is suggested that additional work is performed to determine how many copy operations can be mapped to navigator internal functions.

Other areas where XPath processing can be replaced through internal processing are the management of transition conditions and correlation processing. In those cases, the determination of the functions is slightly easier, since the XPath functions are only used in selecting the proper pieces in variables.

```

1     <correlationSet name="correlation1">
2         <properties>
3             <property>
4                 <type>SF</type>
5                 <variable>OutRequest1</variable>
6                 <part>message</part>
7                 <field>Field</field>
8             </property>
9         </properties>
10    </correlationSet>

```

Listing 8.48: Correlation Set Definition

Listing 8.48 shows the appropriate definition for the correlation set `correlation1`. All properties that make up the correlation set are defined via property elements within the properties element.

The processing of each property is defined via the type element; for example, the value `SF` defines that the correlation set is identified via a single field. In this case, the field is identified via the `field` element. Other elements identify the variable and the part that contains the correlation set.

The performance improvements shown in Section 8.12 are the result of the reduced CPU cycles. As can be seen, the numbers are quite impressive, so further work in this area most likely will help to improve performance for more complex expressions.

## 8.9. Intra Transaction Cache

The instance cache, managed by the Instance Cache Manager (ICM), is constructed at the beginning of each transaction, over time filled with appropriate information from the runtime database, changed information persisted at the end of the transaction in the runtime database, and finally discarded. If the time between two subsequent transactions of a process instance is very short, it is advantageous to move the instance cache to another cache, the *intra transaction cache* (ITC), managed by the Inter Transaction Cache Manager (ITCM) at the end of the transaction, and reconstruct the instance cache from the intra transaction cache when the following transaction starts.

Figure 8.5 illustrates the overall processing. When the navigator finishes a transaction, it calls the ICM to persist any changes in the runtime database.

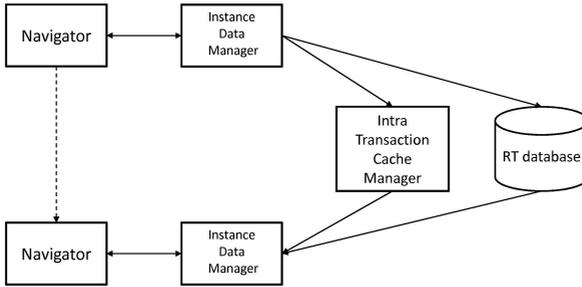


Figure 8.5.: Intra Transaction Cache Architecture

The ICM carries out the processing, shown in Listing 8.49, to shrink the instance cache before handing it over to the ITC and persisting those process instance objects that must be persisted, such as correlation set instances.

Line 1 removes any activity, that is no longer needed. The only activities that are kept are those activities, that are carried out in two phases, such as receive, wait, pick, or synchronous invoke activities, and multi-transaction join activities. Line 2 removes any variable, that is no longer needed, that means whose end state in the appropriate variable definition in the FEP is set to DELETE; Line 3 performs the same for correlation sets, using the appropriate correlation set information in the FEP.

Next, for all activity, variable, and correlation set instances that need to be persisted in the runtime database, appropriate SQL calls are generated and added to the SQL batch. This batch is then carried out in Line 7.

```

1   eliminate all activities that are no longer needed
2   eliminate all variables that are no longer needed
3   eliminate all correlation sets that are no longer needed
4   determine activities that need to be persisted and
    put them in batch update SQL
5   determine variables that need to be persisted and
    put them in SQL batch
6   determine correlation sets that need to be persisted and
  
```

```

        put them in SQL batch
7      carry out batch update call
8      nullify all references in the cache object to the
        model information
9      hand over cache root reference to the intra transaction
        cache manager
10     nullify the cache root reference in the instance
        cache manager to destroy cache contents

```

#### Listing 8.49: Copying Instance Cache to Intra Transaction Cache

Line 8 removes all references in the different objects to the associated meta information by nullifying them, for example the reference to the associated variable information in a variable instance is eliminated. This breaks the linkage to the process model in the model cache instance associated with the current navigator instance, a linkage that may be invalid when the next step of the process instance is carried out by a different navigator instance.

Line 9 hands over the cache root object, which is the process instance root, to the intra transaction cache. The intra transaction cache inserts the passed reference into the tables that manage the information; that means the process instance is not copied. This approach is only possible since all SWoM components are running in the same IBM WebSphere; note that this approach does not work in a clustered environment- in this topology the intra transaction cache actually needs to copy the instance cache contents to its own memory (this scenario is discussed in Section 12.3).

Line 10 finally removes the original cache reference in the transaction cache by nullifying it; that means the content of the cache is destroyed.

The persistence and caching processing is driven by information in the FEP. Listing 8.50 shows the persistence and caching information for a particular activity. The caching property tells the instance cache whether the activity should be kept in the cache or not; the persist property tells the instance cache whether the activity should be written to the runtime database. Similar information is provided for variables and correlation sets.

```

1      <activitiesInTransaction>
2          <activityInTransaction name="A">
3              <caching>NO
4                  </caching>
5                  <persist>NO

```

```

6         </persist>
7     </activityInTransaction>
8 </activitiesInTransaction>

```

### Listing 8.50: Persistence and Caching Information

Listing 8.51 shows the processing that the instance cache manager carries out when the navigator requests the loading of a process instance, either via the process instance identifier supplied by the navigator or by using correlation information provided in the message that the navigator received.

```

1     access process instance in database
2     if process instance has been terminated text
3         tell intra transaction cache to remove
           the appropriate cache entries
4         return
5     end if
6     get appropriate instance cache entry from
           intra transaction cache
7     if not found then
8         store retrieved process instance in instance cache
9         return
10    end if
11    re-link all objects in the cache to the model information

```

### Listing 8.51: Copying Intra Transaction Cache To Instance Cache

The instance loader, as a first action, retrieves the process instance from the runtime database (Line 1) to determine if the process instance has been terminated (Line 2) and to set an appropriate lock on the process instance if required. If the process instance has terminated, the instance loader tells (see Line 3) the intra transaction cache to eliminate all cache entries that are associated with the process instance. Line 6 locates the process instance in the intra transaction cache; if not found (see Line 7), the process instance retrieved in Line 1 is inserted into the instance cache (Line 8). The instance loader obtains the process instance by retrieving it from the runtime database. Line 11 finishes the processing by restoring all meta data references in the process instance, for example the reference to the associated variable in a variable instance.

The intra transaction cache processing is controlled via appropriate information in the FEP. In particular, the ICM must tell the ITCM exactly what it

wants it to do. Listing 8.52 shows the information that the ICM needs to know to make the proper request to the ITCM. The `storeInIntraTransactionCache` tells the ICM whether it should store the instance cache in the ITC.

```
1 <transaction>
2   <executionProperties>
3     <getFromIntraTransactionCache>COPY
4   </getFromIntraTransactionCache>
5   <storeInIntraTransactionCache>YES
6   </storeInIntraTransactionCache>
7 </executionProperties>
8 </transaction>
```

Listing 8.52: Intra Transaction Cache Requests

The `getFromIntraTransactionCache` element controls the type of request that the instance cache manager should use when getting the cache contents. Three values are supported:

**YES** which just tells the ITCM to copy the reference to the ICM, but still keeps a copy. Note that all the changes that the ICM makes to the cache are automatically reflected in the ITC.

**YESANDDELETE** tells the ITCM to copy the reference to the ICM, and then remove the entry in the intra transaction cache.

**COPY** tells the ITCM to make a copy and return the reference to the ICM.

### 8.9.1. Multiple Cache Execution

The value `COPY` is needed since the same instance cache copy may be used by parallel transactions. In this case, the first transaction that is carried out would overwrite data in the process instance, causing wrong data to be supplied to the second transaction. The obvious solution to the problems arising from multiple parallel transactions is the usage of multiple (parallel) caches for a single process instance. Figure 8.6 illustrates this situation for the benchmark process executing with transaction flow type `MEDIUM`. Note that the sequence in which the transactions T4 and T5 are carried out is non-deterministic. The figure assumes that transaction T5 is carried out before transaction T4.

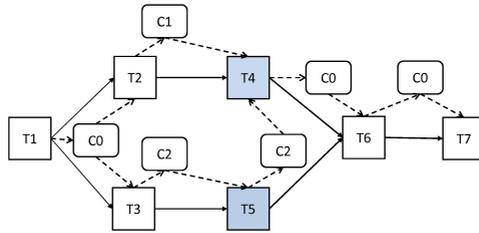


Figure 8.6.: Parallel Cache Usage

When the first transaction T1 finishes, it saves the transaction cache in ITC slot C0, and inserts two messages into the navigator queue that trigger the execution of the two transactions T2 and T3. Each of the transactions obtains the instance cache created by the first transaction; this mandates that the ITC makes a copy of the cache, so that each of the transactions operates on its own cache. When the transactions complete, they save their instance caches into the two ITCs slots C1 and C2. Listing 8.53 shows the definitions for T2.

```

1   <transaction ID="T2">
2     <executionProperties>
3       <getFromIntraTransactionCache>COPY
4     </getFromIntraTransactionCache>
5     <storeInIntraTransactionCache>YES
6   </storeInIntraTransactionCache>
7     <targetCache>C1</targetCache>
8     <sourceCache>C0</sourceCache>
9   </executionProperties>
10  </transaction>

```

Listing 8.53: Cache Slot Definitions

The sourceCache element identifies the cache slot from which the cache should be obtained. In this case it is the one in which the first transaction T1 has stored the cache. The targetCache is the cache slot into which the instance cache is stored.

The cache handling needs a final modification for those transactions that are participating in a join activity. Listing 8.54 illustrates the additional definitions that are added.

```

1     <transaction ID="T4">
2         <executionProperties>
3             <sourceCache>C1</sourceCache>
4             <targetCache>C1</targetCache>
5             <joinCaches>
6                 <joinCache>C2</joinCache>
7             </joinCaches>
8             <completionCache>C0</completionCache>
9         </executionProperties>
10    </transaction>
11    <transaction ID="T5">
12        <executionProperties>
13            <sourceCache>C2</sourceCache>
14            <targetCache>C2</targetCache>
15            <joinCaches>
16                <joinCache>C1</joinCache>
17            </joinCaches>
18            <completionCache>C0</completionCache>
19        </executionProperties>
20    </transaction>

```

Listing 8.54: Cache Joining

The `joinCaches` element identifies the set of caches that eventually are joined and that are used to process the join activity. The navigator handles the situation of join activities by retrieving the join activity from the database and determine the number of links that have entered the activity. The appropriate statements, for example Line 3 through Line 10 in Listing 8.29, are replaced by the statements shown in Listing 8.55.

```

1     for all defined join in caches do
2         get cache from intra transaction cache manager
3     end for
4     if not all caches found then
5         create activity instance
6         continue with next activity
7     end if
8     combine all caches into one cache

```

Listing 8.55: Join Cache Processing

Line 1 through Line 3 obtain all caches that are defined as `joinCache`. These caches contain the other executions of the multi transaction join activity. If not all are found, then the execution of the join activity is not the last one. So, an activity instance is created and processing continues with the next activity in

the transaction. Otherwise all caches are combined into one single cache and processing continues as normal.

Figure 8.7 shows the improvement in SQL calls for the benchmark process. Note that comparison is for the transaction flow type ultimate.

	Nav Tx	SI Tx	SQL Calls	SQL Batch Update Calls	SQL Calls in Batch Update
Normal	3	2	12	2	10
ITC	3	2	3	2	7

Figure 8.7.: Intra Transaction Cache Statistics

The usage of several caches has the advantage that parallel parts of a process model can be carried out in parallel, if the different paths do not modify the same set of variables or activities. In other words they do execute deadlock-free. The approach has several disadvantages. First, the same object, in particular variables, may be stored multiple times in several transaction caches. If the same variable is used in parallel paths, then each cache contains the variable. Second, transactions can have only one multi-transaction join activity. The appropriate code in Section 8.3.1 constructs transactions that may contain several multi-transaction join activities as end activities. It is impossible to handle this situation with caching, as the cache combination can only be carried out if all caches for all join activities are available. So, the flow optimizer can only generate transactions with a single multi transaction activity, increasing the number of transactions for a particular process model. Third, the handling of multiple caches, in particular the joining of multiple caches into a single one, is complex and CPU resource intensive.

### 8.9.2. Single Cache Execution

The disadvantages of the multiple cache approach can be overcome by using a single cache approach. The advantage is a simpler cache, and thus provides for a simpler execution processing. The disadvantage is that the transactions that

make up the process instance execution need to be carried out sequentially which only impacts the response time (see the discussion in Section 4.2).

```
1 <executionOptions>
2   <baseOptions>
3     <cacheMode>SINGLE</cacheMode>
4   </baseOptions>
5 </executionOptions>
```

Listing 8.56: Cache Mode Definition

The flow optimizer tells the navigator and the instance cache to use single cache mode via the `cacheMode` entry in the FEP shown in Listing 8.56.

The performance impact in terms of SQL calls is the same as for the multi-cache mode. Further work is needed to determine if there are situations where the single cache mode is significantly less efficient than the multiple cache mode. Unless those situations have been identified, the SWoM uses single cache mode as default.

## 8.10. Cache Persistence

The usual way of storing process instance information is to maintain it in de-composed form in the runtime database. This approach has been presented as the only one and all database-related optimizations have been applied to the underlying database schema. The flow optimizer indicates this storage mechanism through an appropriate setting in the FEP, as shown in Listing 8.57.

```
1 <executionOptions>
2   <baseOptions>
3     <persistenceMode>DATABASE</persistenceMode>
4   </baseOptions>
5 </executionOptions>
```

Listing 8.57: Database Persistence Definition

This approach of decomposing the process instance has several advantages:

- Only the data that is needed is actually retrieved from the database, and only changed information is written back to the database.

- If one, for whatever reason, would like to query on information not maintained in the process instance, one could set up indices in the database so that appropriate queries can be carried out very efficiently.

The disadvantages are the ones which have been the target of several performance improvement techniques :

- The number and efficiency of the SQL calls
- The CPU cycles that are needed to handle the impedance mismatch between the database and the memory representation.

Another approach for maintaining process instance information is by storing the cache images in the runtime database. This approach has remote similarity with the storage approach used in IBM FlowMark [LR94], where the persistence mechanism was ObjectStore[LLOW91], an object database; the process instance memory structures were directly mapped into the database [Kim90].

```

1   <executionOptions>
2     <baseOptions>
3       <persistenceMode>CACHE</persistenceMode>
4     </baseOptions>
5   </executionOptions>

```

Listing 8.58: Cache Persistence Definition

The basic approach is to store the caches as they are maintained in the ITC as cache images into the runtime database. The storage is determined by the cache ID that is associated with the cache in the transaction. The flow optimizer sets the appropriate persistence mode, as shown in Listing 8.58.

Listing 8.59 shows the structure that is used for storing the cache with slot identifier 0 in the process instance table. This slot is used exclusively if in single-cache mode, and for the cache allocated to slot 0 in multi-cache mode. From a performance perspective it is desirable to use cache slot 0 as often as possible, since the process instance is accessed in every transaction to determine if the process instance has terminated in the mean time.

```

1 CREATE TABLE SWOM.PROCESS_INSTANCE (
2     PIIID                CHAR(12)                NOT NULL
                          PRIMARY KEY ,
3     PMID                 CHAR(12)                NOT NULL,
4     SHORT_CACHE          VARCHAR (32000)  FOR BIT DATA ,
5     CACHE                 CLOB (2000K)
6 )

```

Listing 8.59: Cache Slot Zero Storage

When the cache needs to be persisted, the final cache, after the instance cache manager has done its work of cleaning the cache and determining the objects that need to be stored in any case, is serialized.

Listing 8.60 shows the table that is maintained for caches with slots greater than 0.

```

1 CREATE TABLE SWOM.CACHE (
2     SLOT                SMALLINT                NOT NULL
                          WITH DEFAULT 1 ,
3     PIIID                CHAR(12)                NOT NULL
                          REFERENCES SWOM.PROCESS_INSTANCE
                          ON DELETE CASCADE ,
4     SHORT_CACHE          VARCHAR (32000) ,
5     CACHE                 CLOB(2000K) ,
6     PRIMARY KEY (PIIID, SLOT)
7 )

```

Listing 8.60: Cache Slot Non Zero Storage

These slots are used for storing those caches in multi-cache mode, that have a slot identifier of greater 0. The `SLOT` element contains the cache slot identifier. It is combined with the process instance identifier to make up the key of the table.

Figure 8.8 shows the improvements in SQL calls when running the benchmark process in multi-cache cache persistence mode. These measurements have been carried out to see how multi-caching performs in this situation. When running in multi-cache mode, the size of the individual caches is smaller than the size of the cache that is used in single-cache mode, since the cache in single-cache mode contains all the data. So, the size of the data that has to be exchanged between the SWoM and the database is smaller in multi-cache mode.

The disadvantage of the multi-cache mode are the extra SQL calls needed for cache joining.

	Nav Tx	SI Tx	SQL Calls	SQL Batch Update Calls	Calls in Batch Update Calls
Normal	3	2	12	2	10
ITC - Multi	3	2	5	2	4

Figure 8.8.: Multi Cache Mode Persistence Statistics

Figure 8.9 shows the number of SQL calls that are carried out in single-cache mode. The two calls that are saved are the ones that are used when processing the multi-transaction join activity G. In this case the other cache must be retrieved to evaluate whether the join activity can be processed (see Listing 8.55). Incidentally, this is the minimal number of SQL calls that can be obtained. The process is carried out in three transactions: in the first and second transaction, the data needs to be persisted, in the third transaction, the process instance information is removed from the database.

	Nav Tx	SI Tx	SQL Calls	SQL Batch Update Calls	Calls in Batch Update Calls
Normal	3	2	12	2	10
ITC - Single	3	2	3	2	4

Figure 8.9.: Single Cache Mode Persistence Statistics

It is obvious that the major advantage of the cache persistence approach over the standard database approach is the reduction of SQL calls and the CPU cycles that are needed for preparing the in-memory structures for persistence and vice versa.

The only big disadvantage of cache persistence is the reading and writing of non-used information; the cache contains all information that is used in the current transaction as well as all following transactions. If, for example, a variable of 10 MB is used in the first transaction  $T_1$  and again in the third transaction  $T_3$ , then the variable is read and written in the second transaction  $T_2$  without even being touched (in addition to the cycles required for serialization/deserialization). It can be assumed that this problem is of more significance in single-cache mode, since always all data is kept in the cache.

The advantages and disadvantages of the two persistence options cache persistence and standard database storage can be leveraged by having the flow optimizer determine the optimal process context distribution strategy. This is tremendously complex and needs significant more research.

Another option is to just store the process instance state in the intra transaction cache, resulting in a memory-only execution. If the system terminates for any reason, or if the process instance state must be flushed from the intra transaction cache, the process instance state is lost. This is certainly not tolerable for most types of process models, however is a valid execution option for low-value process instances. This approach is discussed in Section 10.4.

## 8.11. Execution Linearization

As pointed out earlier, the individual transactions of a process instance are in general processed sequentially (unless the access in parallel paths does not cause any deadlocks). The SWoM issues appropriate SQL calls that cause the process instance to be locked, resulting in the sequential execution of the transactions that make up the transaction flow. Note, that carrying out the individual transactions in sequence does not mean that the complete process is carried out in sequence. It just means, that the execution of the transactions for the individual parallel paths are carried out in sequence.

Conceptually, each process model can be rewritten for linear execution. Such a linearized process model can then be carried out more performantly for two reasons: (1) No synchronization is required for join activities and (2) no locking of the process instance must be performed since no more than one

request is active for a process instance. It should be noted that process model linearization is a prerequisite for memory only execution of process models as shown in Section 10.4.

Unfortunately, the execution time may go up significantly, since all wait times, either as the result of wait activities or time elapsed between invoke and associated receive activities, are added to come up with the total execution time. Whether this is a real problem depends on the execution requirements expressed by the user and the structure and wait time characteristics of the process model. In the benchmark, for example, the execution time of an execution linearized process model is not any significantly different from the non-linearized version since there are virtually no wait times. Listing 8.61 shows the appropriate linearized process model structure for the benchmark process.

```
1      <sequence>
2      <receive createInstance="yes"
3          name="A" .. />
4      <assign name="B" .. />
5      <invoke name="C" .. />
6      <receive name="E" .. />
7      <invoke name="D" .. />
8      <receive name="F" .. />
9      <assign name="G" .. >
10     <invoke name="H" .. >
11     <invoke name="I" .. />
12     </sequence>
```

Listing 8.61: Linearized Process Execution

Rewriting a process model into linear execution is using a depth-first algorithm to keep the different paths separated. Note, that the algorithm shown in Section 8.3.1 for creating the transaction flow is a breadth-first algorithm.

Listing 8.62 shows the main processing that is carried out. Line 1 allocates the activity queue that contains the activities to be processed; Line 2 allocates the fork activity stack; it used to keep track of the fork activities where a path is being followed. Line 3 allocates the activities list, which at the end contains the linearized process model.

```

1   allocate activity queue
2   allocate fork activity stack
3   allocate activities list
4   get process start activity
5   put process start activity into activity queue
6   while activity queue is not empty do
7       call HandleActivity
8   end while

```

Listing 8.62: Process Linearization

Line 4 gets the process start activity and puts into the activity queue for processing. Line 6 reads the activities to be processed from the activity queue until all activities have been processed. For each activity that is read from the queue, the function `HandleActivity` is called.

Function `HandleActivity` shown in Listing 8.63 performs the processing that is needed for each activity. Line 2 checks whether the activity is a join activity and if so, checks whether all incoming links have been processed. If not, then the number of incoming links is increased, the activity is marked as a stop activity, the next activity is determined, and then control is returned.

```

1   function HandleActivity
2       if join activity then
3           if not all incoming links processed then
4               increase number of incoming links
5               mark activity as stop activity
6               call DetermineNextActivity
7               return
8           end if
9       add activity to activities list
10      call DetermineNextActivity
11  end function

```

Listing 8.63: Activity Handling

The activity is processed after all incoming links have been processed; the function `DetermineNextActivity` in Listing 8.64 delivers join activities multiple times for processing. Line 9 adds the activity to the list of activities and then have the next activity being determined.

Listing 8.64 determines the next activity to be processed. It first checks whether the currently processed activity is a join activity. If not all outgoing

links have been processed, then Line 5 through Line 9 perform the appropriate processing. First, the activity is put into the fork activity stack; it is used later again, after all depending activities have been processed, to backtrack to the last fork activity to continue processing. Second, the next outgoing link is determined, the target activity of the link identified, and the activity inserted into the activity queue. If all links have been processed, the fork activity is marked as a stop activity, so that backtracking to a previous fork activity is being carried out.

```
1  function DetermineNextActivity
2  if activity is fork activity then
3      if not all outgoing links processed then
4          insert fork activity into fork activity stack
5          determine next outgoing link
6          increase number of outgoing links for activity
7          determine target activity
8          insert activity into the activity queue
9          return
10     else
11         mark activity is stop activity
12     end if
13 end if
14 if activity is stop activity or has no outgoing links then
15     pop activity from fork activity stack
16     if stack is not empty then
17         call DetermineNextActivity
18     end if
19     return
20 end if
21 determine outgoing link
22 add target activity to activity queue
```

Listing 8.64: Next Activity Determination

If the activity is marked as stop activity or the current activity has no outgoing links, backtracking to the last fork activity is started by obtaining the latest fork activity from the fork activity stack. If the stack is not empty, function `DetermineNextActivity` is called again with the found fork activity.

The following processing takes place when the benchmark process is linearized. First, activity A is put into the activity queue. After it has been processed, the next activity is determined, which yields activity B, that is processed as well. When called for finding the next activity, function `DetermineNextActivity`

determines that activity B is a fork activity and that not all links have been processed. So it puts activity B into the fork activity stack and selects the first link. The first link delivers activity C which is just added to the activities list. Following the outgoing link returns activity E; it is added to the activities list. The next activity to be processed is activity G. As it is a join activity, the number of incoming links is incremented and since not all links have entered, the activity is marked as a stop activity. This causes activity B to be retrieved from the fork activity stack and have the next outgoing link to be processed, which yields activity D, followed by activity F. Next activity G is selected again. This time all incoming links have been processed, so activity G is added to the activities list. Next activities H and I are processed. As activity I has not outgoing links, activity B is now fetched again from the split activity stack. Since all outgoing links have been processed, the next activity is retrieved from the fork activity stack. The stack is empty, so control is returned. Since the activity queue is empty, processing completes. Listing 8.61 shows the final result.

After the flow optimizer has rewritten the process model, it determines the appropriate transactions using the algorithm shown in Listing 8.65.

```
1   create transaction
2   for all activities in activities listdo
3       add activity to transaction
4       if activity is receive activity then
5           close transaction
6           create new transaction
7           add receive activity to transaction
8       end if
9   end for
10  close transaction
```

Listing 8.65: Transactions Determination

As can be seen each receive activity causes the current transaction to be finished and a new transaction to be created. The receive activity is then inserted into the newly created transaction as the start transaction. Listing 8.66 shows the result of applying the algorithm to the linearized process model shown in Listing 8.61.

```

1   <transactionFlow>
2     <transaction ID="T1">
3       <activitiesInTransaction>
4         <activityInTransaction name="A"/>
5         <activityInTransaction name="B"/>
6         <activityInTransaction name="C"/>
7         <activityInTransaction name="E"/>
8       </activitiesInTransaction>
9     </transaction>
10    <transaction ID="T2">
11      <activitiesInTransaction>
12        <activityInTransaction name="E"/>
13        <activityInTransaction name="D"/>
14        <activityInTransaction name="F"/>
15      </activitiesInTransaction>
16    </transaction>
17    <transaction ID="T3">
18      <activitiesInTransaction>
19        <activityInTransaction name="G"/>
20        <activityInTransaction name="H"/>
21        <activityInTransaction name="I"/>
22      </activitiesInTransaction>
23    </transaction>
24  </transactionFlow>

```

Listing 8.66: Linearized Process Transaction Flow

## 8.12. Performance Improvements

Figure 8.10 shows the performance improvements that are achieved for the various optimization techniques that the flow optimizer implements.

The first column identifies the individual benchmark run. ID 1 shows the throughput that one achieves with the transaction flow type `ULTIMATE` and all invoked Web Services fully optimized; this is the maximum throughput that one can achieve using the flow configuration options introduced in the previous chapter. This forms the base for relating any performance achievements obtained via flow configuration. Note, that all performance numbers are relative to the number given with ID 1.

ID 2 shows the results of the improvements in transaction internal processing, the variable data access optimization introduced in Section 8.5, and the

ID	Optimization Options	Processes/ min	Improvement (%)
1	Configured with opt WSSs	1195	-
2	Variables/correlation sets optimized		
	a Database persistence	1241	3.8
	b Multiple cache persistence	1250	4.6
	c Single cache persistence	1294	8.3
3	Intra transaction cache		
	a Database persistence	1260	5.4
	b Cache persistence	1294	8.3
4	XPath processing improvement	1294	8.3
5	Execution linearization	1202	0.6
6	Best	1400	17.2

Figure 8.10.: Optimization Performance Improvements

correlation set optimization introduced in Section 8.7. Three different numbers are given for the three different cache persistence options.

ID 3 shows the impact of intra-transaction caching for the database and the single cache image option. As can be seen, the intra transaction does not buy anything if cache persistence is active. This result should be expected. In single cache persistence mode, the number of SQL calls are the same with and without intra transaction caching; the only difference is that the cache must not be deserialized if inter transaction caching is active. The serialization mechanism used in cache persistence mode is very efficient, so the impact is minimal.

ID 4 shows the impact of the improvements in XPath processing, ID 5 the impact of process execution linearization, and ID 6 the throughput performance that can be achieved by combining single cache persistence, XPath optimization, and process execution linearization.

### 8.13. Flow Execution Plan Management

The flow execution plan can be constructed in two different ways: by the flow optimizer or by some external tool. The flow optimizer is invoked via the administration interface. It generates the FEP and stores it in the buildtime database.

The FEP can also be generated by some external tool. An appropriate import facility in the administration interface provides the capability to import the FEP. A set of consistency checks is carried out to make sure that the specified values are valid; however a complete consistency checking is not yet performed.

Other administrative functions provide the process administrator with the capability to export a flow execution plan, delete it from the buildtime database, and to show whether a flow execution plan is attached to a process model.

### 8.14. Road Map

The notion of transaction flows forms an excellent base for improving the performance of the SWoM: the configuration and optimization techniques and their usage by the flow optimizer are evidence. This section outlines a road map for adding more optimization functions and improving the flow optimizer. It should be noted that building a production-level WfMS is a tremendous effort, requiring hundreds of developers, let alone the building of a flow optimizer. The author knows from his professional experience that building the query optimizer in a RDBMS requires significant research and development effort. He further believes that building a flow optimizer for a WfMS is significantly more challenging and resource-intensive than building a database optimizer, since the underlying meta model, WS-BPEL, is more complex, particularly when people support is added to the WfMS.

### 8.14.1. Cost-Based Transaction Flow Type Selection

The flow optimizer uses the transaction flow type the process modeler specifies, or if not specified, the default of the SWoM, which is set to ULTIMATE. The next step is that the flow optimizer selects the optimal transaction flow type based on a specified optimization target. A typical optimization target is the cost associated with the execution of process instances. Cost, in this context, means CPU cycles spent by the infrastructure and the SWoM, memory usage, and network usage; all of these properties can be expressed as cost figures. In a first step, one could use some higher-level constructs, such as number of transactions, SQL calls used, and messages processed as cost criteria (assuming the cycles used by the SWoM are fairly constant for the internal processing). This rudimentary approach needs to be refined so that the actual costs are used for making the selection of the proper transaction flow. The amount of work that is associated with cost-based optimization is enormous; just the work needed to get, for example, the cost factors for the individual SQL calls is tremendous.

### 8.14.2. Statistics-Supported Flow Optimization

The quality of the flow execution plan increases with the amount of information that is available to the flow optimizer. Figure 8.11 shows the overall processing of the flow optimizer when the statistics manager has collected information.

The usage of statistical information requires that the SWoM implements the following functions:

- The statistics manager must be extended to not only collect the information but also determine if any of the collected values have changed over time. Generating FEPs is time-consuming, so it is important that a new FEP is only regenerated if really needed.
- It is most unlikely that no process instances are active when a new FEP is generated, so the SWoM must support versioning. The simplest way of doing this is by storing the identifier of the FEP in the process instance. Unfortunately, this trivial approach does not help existing

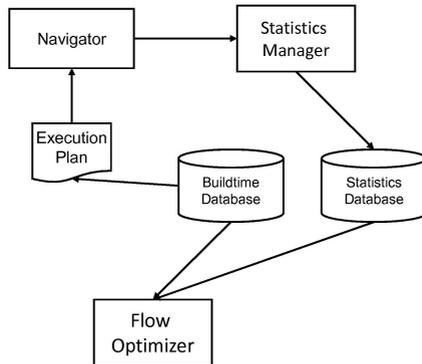


Figure 8.11.: Statistic-Supported Optimization Processing

process instances, which eventually could benefit from the new statistical information. Research is needed to understand how FEPs could be migrated.

### 8.14.3. Feature Completion

The flow optimizer supports only the basic activities receive, invoke, reply, and assign. Further work is needed to support other basic constructs such as pick, or wait. It can be assumed from the experience with the currently supported activities that implementation should be fairly straight forward.

This is most likely different for the more advanced constructs such as compensation spheres. One needs to specify which activity fails, so that one can calculate the processing that needs to be carried out during compensation processing. Proper optimization for compensation spheres definitely requires that one knows the probability of activity failure and link execution. In fact, more advanced optimization requires that the statistics manager collects the probabilities of the different execution paths, as discussed in Section 8.14.5.

#### 8.14.4. Dynamic Transaction Boundaries

The transactions and their boundaries are constructed by determining the proper transaction flow type, either supplied by the process modeler or determined by the flow optimizer by analyzing the costs of each of the transaction flow types.

A next step is the dynamic setting of transaction boundaries by the flow optimizer by evaluating all possible transaction flows, calculating the costs associated with each transaction flow, and using the one with the least cost. The creation of transaction flows can be controlled by users defining the transaction participation properties of activities, such as *commit after* causing the transaction boundary to be set right after the activity.

#### 8.14.5. Execution History Based Optimization

The statistics manager only collects fixed statistical information, such as the length of variables or the response time of a Web Service. This information is sufficient to perform basic optimization. More advanced optimization techniques require that the statistics manager collects information about the execution history of process instances, such as the probability with which a particular path is being taken or the probability with which an activity fails, causing compensation or dead path elimination to be carried out.

A typical optimization technique that benefits from this information is the processing of multiple outgoing links. If an activity has multiple outgoing links, the links are evaluated in the sequence in which they are defined, one by one. The flow optimizer can improve this processing in case the transition conditions are structured in such a way that not all links are followed, by rearranging the order in which the links are evaluated by the probability by which the individual links are followed.

#### 8.14.6. Application-Specific Optimizations

All optimization techniques discussed so far are more or less application agnostic; that means they are not geared towards special application requirements.

Obviously they are the most important ones for their large applicability. However, there are a number of application areas or characteristics that require special optimization techniques. The following list shows some of those application specific optimization techniques.

- WS-BPEL variables contain data that is typically generated and consumed by Web Services that the process invokes. The SWoM, and most likely all other WfMSs, maintains the content of variables in its own data store. This is normally the right architectural approach; however there are situations where it would be more efficient to leave the data in the original data stores and have it brought into a variable when needed and stored back into the data store when the data has been modified. [LR10] proposes a technique for materializing/dematerializing variables.
- The execution of a workflow is typically carried out under the assumption that the data used by the invoked Web Service is locally available when the Web Service is called. Otherwise the Web Service must either obtain the data through a remote call or first copy the data from the remote site. [LR04, LR00c] propose an optimization technique to cope with this situation by allowing the process modeler to control when the data to be used for a Web Service is brought to the location where the Web Service resides, so that access to the data is via local speed. This technique definitely reduces the execution time of the process instance; whether the overall resource consumption is reduced resulting in improved throughput depends most likely on the appropriate situation.

#### 8.14.7. Business Goals Optimizations

All optimization techniques discussed so far are basically geared towards reducing the resources that the SWoM needs to carry out process instances. All of them help improve the throughput, some of them in addition the latency/response time. There are other ones for which a particular process model could be optimized: business goals.

A typical business goal optimization is the minimization of the costs associated with the called Web Services. In the real world, Web Services are quite often associated with certain costs, for example the price that one pays for the goods ordered. In addition, canceling an order typically involves some cancellation fees. [AKL<sup>+</sup>08] proposes an optimization technique for minimizing the overall external costs in case cancellations are taking place as the result of compensation processing.

One can drive the cost optimization even further by optimizing the requests of several process instances together. For example, if the cost of a pencil is 5 cents if you order 500 pieces and four cents if your order 1000 pieces, then one could combine the requests of all process instances that order pencils in such a way that only one order for pencils is created and the result is distributed accordingly. Research is needed to come up with solutions for this type of optimizations.

# INFRASTRUCTURE SPECIFIC OPTIMIZATIONS

It can be expected that some if not all of the optimization techniques developed so far can be applied to other Workflow Management Systems running in different infrastructures. However, additional optimization/tuning techniques have been developed that are only applicable to the infrastructure provided by IBM WebSphere and IBM DB2. Whether the achieved results can be applied to other infrastructures, that means other application servers and relational database management systems, can not be judged.

The efficient execution of the SWoM depends to some extent on the proper configuration of the components of the underlying infrastructure, that means on IBM WebSphere and IBM DB2. This chapter discusses configuration settings for both infrastructure components, introduces a new optimizer component, the System Optimizer (SO), that optimizes the infrastructure settings, and the appropriate System Statistics Manager (SSM) that maintains the necessary information for system optimization.

The SO has been used to tune the settings used in the benchmark runs,

for example, it has been used to set the collection pool size used by IBM WebSphere.

### 9.1. System Optimizer

The system optimizer (SO) is responsible for tuning the settings of the SWoM and the infrastructure components IBM WebSphere and IBM DB2. Figure 9.1 shows the basic architecture of the SO.

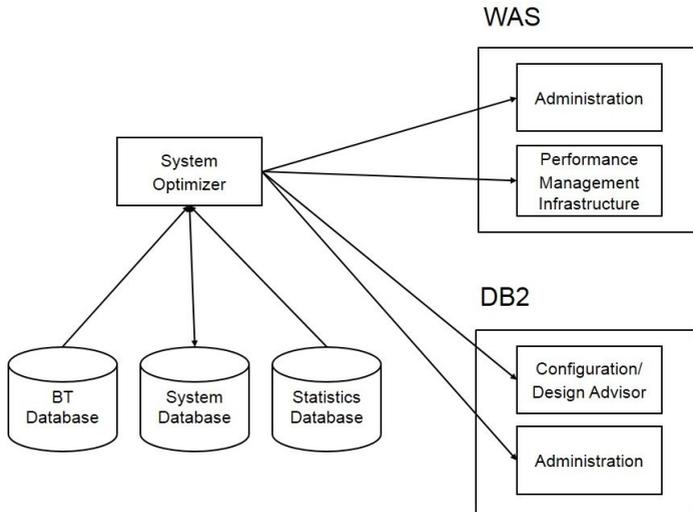


Figure 9.1.: System Optimizer

It uses information stored in the buildtime database as well as information collected by the SSM and stored in the system statistics database to help carry out the tuning. These settings are then propagated to IBM WebSphere and IBM DB2 through the respective administrative and related components.

The SO is made up of several internal components that are introduced when the appropriate configuration settings and their handling are introduced later. The SO is periodically activated by the administrative components to determine

if any actions have to be taken and if so, an appropriate administrative alert is generated so that the system administrator can initiate the proper actions. If possible and if set appropriately, the actions are automatically carried out by the SO, reducing the need for manual interventions.

## 9.2. System Statistics Manager

The System Statistics Manager (SSM), shown in Figure 9.2, records important IBM WebSphere and IBM DB2 related activities, such as the number of active EJBs for determining the connection pool size or the average number of SQL calls within a transaction as input to the DB2 Design Advisor. The architecture of the SSM follows the one of the Statistics Manager introduced in Section 7.10.

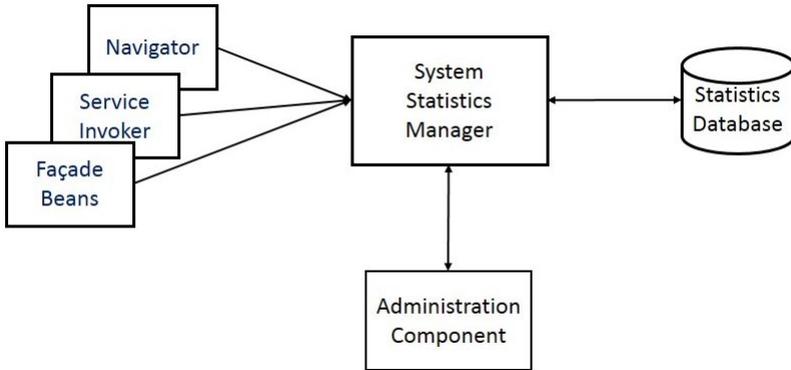


Figure 9.2.: System Statistics Manager

The SSM is called whenever the SWoM performs something that is related to IBM WebSphere or IBM DB2, such as the creation or destruction of navigator, service invoker, or façade beans instances as well as the allocation and deallocation of a database connection. The collected information is periodically written to the system database; the administration component via its administration interface provides appropriate display functions. The collected information can

be used by the system optimizer to modify the IBM WebSphere or IBM DB2 settings that better suit the processing characteristics of the SWoM.

### 9.3. IBM WebSphere

IBM WebSphere provides several functions that directly affect the performance of the SWoM: (1) it manages the execution of the different SWoM components, (2) it provides timer support for time-related WS-BPEL constructs, such as the wait activity, (3) it provides the communication mechanism for the SWoM via queuing, and (4) it handles all interactions between the SWoM and IBM DB2. The relevance of these functions is discussed next.

#### 9.3.1. Connection Handling

The database accesses of any application in IBM WebSphere are controlled by IBM WebSphere. Each database that needs to be accessed is defined as a data source that is associated with a set of properties defining the behavior of the data source. Only the proper setting of these properties provides the optimal execution of the SWoM's database operation. The most important properties, whose correct settings are critical from a performance perspective, are the size of the connection pool and the size of the prepared statement cache [HB11].

##### 9.3.1.1. Connection Pool Size

All SWoM database accesses are managed within a connection that the SWoM requests from IBM WebSphere when it starts accessing the database and releases when processing finishes. IBM WebSphere, for performance reasons, manages these connections in a pool (connection pooling) from which a connection is fetched when one is needed by a SWoM component and returned when the connection is no longer needed. The number of connections within the pools is defined via the *connection pool size* property that can be set either via the administration or scripting interfaces of IBM WebSphere.

If the number of connections is defined as too low, not enough SWoM component instances can run, as they will not get access to the database; if the

number is set too high, resources are wasted that could be used for productive work [Dug11].

SWoM maintains internal statistics counters for the EJBs that maintain connections. The SWoM administrative interface provides the capability to display this information together with the connection pool information from IBM WebSphere using the built-in performance management infrastructure (PMI) [QR04]. This helps an administrator to make appropriate changes to the pool size.

#### 9.3.1.2. Prepared Statement Cache Size

When the SWoM issues an SQL call, two high-level processes are being carried out: (1) the statement is *prepared*, and (2) the statement is *executed*. During the prepare phase, IBM DB2 parses the SQL statement and performs the steps necessary to put the query into some internal form suitable for efficient processing, a phase that consumes a noticeable amount of CPU cycles.

IBM WebSphere helps reduce the time it takes to run the prepare phase by caching the prepared statements. When the SWoM executes an SQL statement, IBM WebSphere determines if the SQL text is already in the cache, and if so, it uses that cached statement instead of preparing a new one. The best performance is achieved when the prepared statement cache is made large enough to hold all of the statements being prepared. IBM Tivoli Performance Viewer, an integral part of IBM WebSphere, can be used to see how many statements are being discarded from the prepared statement cache. Statements are discarded when the cache is full and room must be made for newly prepared statements. Further work is needed to determine whether the SWoM can calculate the size of the statements that it executes and so can propose the correct size of the prepared statement cache.

#### 9.3.2. Scheduler Table

The SWoM uses, as explained in Section 3.10.3, the timer service of IBM WebSphere to handle time-controlled activities, such as the wait activity. IBM

WebSphere stores all timer requests persistently and transacted in the scheduler table so that IBM WebSphere can recover if an error occurs.

IBM WebSphere periodically queries the scheduler table to see whether any of the stored timers has expired, and if so, carries out the designated actions [Joh04], such as the sending of a message to the SWoM. The frequency with which the queries are carried out is defined by the system administrator via appropriate administrative commands. Obviously, this frequency has some impact on the amount of resources needed for carrying out the queries and the precision with which time-related activities are processed. A small value causes only small delays in processing an appropriate action. If, for example, the value is set to 1 minute, then a wait activity completes definitely within a minute of its requested completion time. The disadvantage is the amount of resources required if the scheduler table is accessed so often. If a large value is chosen, then the precision with which an activity is processed is lower, but also the resource consumption is lower. The selected value in fact must be chosen as a trade-off between the business process preciseness requirements and the resource consumption needed to access the scheduler table.

The optimal checking frequency can be determined by attaching the process models or even the individual activities with a preciseness specification for the associated timer; Listing 9.1 shows how the precision for an activity is specified.

```
1 <processDeploymentDescriptor>
2   <activity name="wait">
3     <precision>PIH</precision>
4   </activity>
5 </processDeploymentDescriptor>
```

Listing 9.1: Precision Specification for Wait Activity

The precision element specifies how accurate the wait activity should be carried out. In the example, the wait activity should complete within an hour after starting. In this case, it would be sufficient if the scheduler performs an appropriate query every hour.

It is the responsibility of the system optimizer to determine the optimal access frequency of the scheduler by keeping information about the individual process

models. If a new access frequency is needed, the scheduler properties are modified through invocation of the appropriate IBM WebSphere administration functions.

The precision specification of the various wait activities helps the system optimizer to come up with the minimal time that the timer service should use. It is therefore possible that a process which is carried out only rarely determines the timer frequency, which possibly is not optimal.

```
1 <systemDeploymentDescriptor>
2 <schedulers>
3 <scheduler>
4 <to>P1H</to>
5 <name>WFSchedulerHour</name>
6 </scheduler>
7 <scheduler>
8 <to>P1D</to>
9 <name>WFSchedulerDay</name>
10 </scheduler>
11 <scheduler>
12 <to>P1M</to>
13 <name>WFSchedulerMonth</name>
14 </scheduler>
15 <scheduler>
16 <from>P1M</from>
17 <name>WFSchedulerGreaterMonth</name>
18 </scheduler>
19 </schedulers>
20 </systemDeploymentDescriptor>
```

Listing 9.2: Multiple Scheduler Definition

The performance of the timer service can be improved by using multiple schedulers, with each scheduler addressing a particular time range. Listing 9.2 shows how one can specify four schedulers. Each scheduler is identified via the scheduler element. The to and from elements specify the precision the scheduler is supporting.

The SWoM checks during import of the SSDD whether all schedulers have already been defined to IBM WebSphere, and if it detects one that does not exist, it defines the scheduler to IBM WebSphere using the appropriate IBM WebSphere interface [Joh04].

When the navigator encounters a wait activity, it determines which scheduler is most suited. Furthermore, it keeps track, via the SSM of the number of hits for each scheduler, including the distribution within each range. This provides the system optimizer with sufficient information to propose, when requested, a set of new schedulers that better match the current load.

### 9.3.3. JMS Message Engine

IBM WebSphere comes with a built-in message engine that delivers the necessary JMS functionality. IBM WebSphere offers three options for storing messages persistently: (1) by means of a flat file, (2) the exploitation of the built-in Apache Derby database [The11] and (3) the usage of an external DBMS, such as IBM DB2.

Which one of the first two options is better seems to depend on the type of disk drives that are used for storing the messages. [HB11] claims that the flat file approach outperforms the Apache Derby database approach. [MPGM08] states that an external DBMS is faster than the Apache Derby database; this is also supported by [HB11], which claims that a well tuned flat file approach using a RAID device may almost achieve the performance of the remote database.

### 9.3.4. JVM Setting

It is important that the Java Virtual Machine (JVM) that runs IBM WebSphere and the SWoM is optimized for the execution characteristics of the SWoM.

The first property that needs to be set is the size of the JVM heap in terms of minimum and maximum size. The default settings of IBM WebSphere are not well suited; so when the SWoM is installed, the minimum and maximum size of the JVM heap are set to 1024MB, the value recommended by [MPGM08]. Setting minimum and maximum heap size to the same value prevents the JVM from compacting, that means from dynamically changing the heap size. This operation is quite expensive and can make up as much as 50 % of garbage collection pause time ([HB11]).

Another important aspect of JVM tuning is the garbage collection policy. Since the SWoM is optimized for throughput and has quite a number of long-

lived objects, the *gencon* policy is used. This policy treats short-lived and long-lived objects differently to provide a combination of lower pause times and high application throughput.

## 9.4. IBM DB2

IBM DB2 applications only perform satisfactorily if the databases and tables they use are configured properly. Most, even commercial, WfMSs only offer limited support for tuning databases and tables; they just provide a tuning guide or a set of tuning papers. IBM, for example, offers a short manual that provides only a few elementary tips on how to tailor the databases [IBM10]. ORACLE has a chapter about the tuning of the WS-BPEL engine in the tuning guide of the application server [ORA08].

The SWoM approaches this area differently by assisting IBM DB2 in optimizing by providing input to the optimization tools. This section presents those parts of IBM DB2 that can be configured for optimal performance:

- The *basic configuration* of IBM DB2 so that the SWoM operates optionally in the given environment.
- The setup of the different *databases* that the SWoM uses for its operation.
- The usage of *table spaces* as a means for providing fine-grained storage allocation for databases and hereby reducing the contention of access to different tables that are part of the same database.
- The optimal way of allocating the *tables* that are used during process execution, in particular the tables in the runtime database.
- The usage of *indices* to efficiently support administrative as well as context-dependent process instance queries.
- The allocation of *buffer pools* to hold rapidly and frequently accessed information in memory, in particular the information stored in the runtime database.

The section presents for each of the different areas the SWoM support for IBM DB2 tools, such as the *configuration advisor* or the *database wizards*.

#### 9.4.1. Basic Configuration

The efficient operation of IBM DB2 is achieved through a number of settings and actions that minimize the resources that IBM DB2 consumes as well as reduce contentions to internal resources. In fact, without some basic tuning, as described in this section, application performance, and in particular for a WfMS, is poor. Basic configuration includes the following tasks:

- The *placement* of the different pieces of *system data* on different disk drives.
- The proper setting of the *configuration parameters* that control the execution of IBM DB2.
- The proper support for the *query optimizer* built into IBM DB2.

##### 9.4.1.1. Data Placement

An important task is the proper placement of the different files that hold the data that IBM DB2, IBM WebSphere, and the SWoM use.

Most important is the separation of the IBM DB2 log files from the IBM DB2 data files holding the user tables by putting the log file on a disk drive that is separate from the disks that hold the data files (if sufficient disk are available). This avoids contentions between the read/write operations of the instance data and the write operations of the log.

Furthermore, the IBM DB2 files (log and data) should be separated from disks that hold data for the operating system, IBM WebSphere, the queues, and the logs for the persistent messages. For example, IBM recommends for IBM Process Server in [IBM10] that each of these repositories should be placed on separate disks.

If multiple disks are needed for a database, most likely for the runtime database, or even for a table such as the variable instance table, [MPGM08]

recommends to use a striped RAID array instead of single disks, since the mapping to the different disks is significantly more efficiently handled in the RAID device through appropriate hardware compared to an appropriate software solution. A thorough discussion of placing the different tables of the SWoM onto appropriate disks is presented in detail in upcoming sections.

#### 9.4.1.2. Configuration Parameters

IBM DB2 offers a plethora of configuration parameters that help adapt IBM DB2 for the particular environment IBM DB2 needs to support. For example, the LOCKLIST value defines the maximum number of locks that IBM DB2 is allowed to hold. If the number of actual locks is lower than the specified value, precious memory is wasted that otherwise could be used efficiently for other purposes, such as buffer pools. If the number of actual locks exceeds the capacity of the lock list, lock escalation needs to occur with typically detrimental effects on performance [Bon05, EN94].

The sheer number of configuration parameters and their proper settings for a particular execution environment becomes a nightmare for database administrators, in particular since every new IBM DB2 version introduces new configuration parameters. The developers of IBM DB2 have therefore already begun, for quite a while, to provide appropriate tools that help set up and tune the IBM DB2 environment. For example, the DB2 Configuration Advisor tool helps to set the global parameters for IBM DB2. Input to the tool is basic processing information, such as the main usage of the database (query, transactions, or both), the average number of SQL calls per transaction, the management priority of the database (throughput or recovery), or the isolation level for the database [SS04, KLS<sup>+</sup>03]. Using this tool eliminates, even with the little input that the user must provide, most of the basic configuration and setup errors.

Figure 9.3 illustrates the steps that the system optimizer carries out when requested to run the DB2 Configuration Advisor.

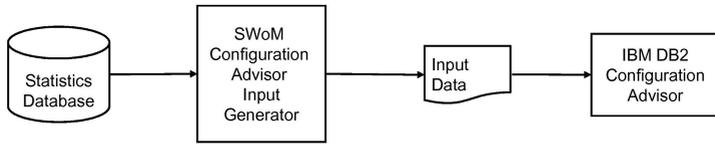


Figure 9.3.: Running the IBM DB2 Configuration Advisor

The *SWoM Configuration Adviser Input Generator*, a component of the system optimizer, generates the appropriate input for the DB2 Configuration Advisor using the information that the statistics manager has stored in the statistics database.

#### 9.4.1.3. Query Optimizer Support

The efficiency of the internal execution of SQL calls is determined through the query optimizer [Cha98]. The optimizer uses statistical information about the physical characteristics of a table and the associated indexes, such as number of records, number of pages, and average record length, stored in the IBM DB2 catalog to come up with an optimal query plan. If the contents of tables change frequently, it is necessary to frequently run the RUNSTATS utility [IBM06], that collects the statistical information, so that the query optimizer always uses the newest statistics: the more accurate the statistical information, the better is the query plan.

Unfortunately, the execution of RUNSTATS is a very time and resource consuming task. It is therefore desirable to run the utility as little as as possible, collecting the minimum amount of information [L. 04]. Autonomic features added in the latest versions of IBM DB2, as described in [PWYT07], help the database administrator in the efficient scheduling of appropriate RUNSTATS runs. It does so by periodically querying the UDI counter, which IBM DB2 maintains for each table. The counter is increased every time a row

is updated, deleted, or inserted. The counter is reset each time a RUNSTATS run against the table has completed. The automatic features check the ratio of *UDI Counter/Table Cardinality* which represents the percentage of changes that have been made against the table. If a system-defined threshold is exceeded, running of RUNSTATS is recommended.

Recent development of database optimizers that compare the information in the query plan with the information of the actual query and automatically adjust the appropriate query plans [MLR03] may eventually make this utility obsolete or allow at least to increase the time intervals between individual RUNSTATS runs.

The function offered through the autonomic feature is a rather crude one. The SWoM can do better in the determination of the optimal time interval and selection of tables. The *SWoM Runstats Analyzer* component, a part of the system optimizer, uses the information collected by the statistics manager to generate a report that indicates whether a RUNSTATS run is needed. This information that the SWoM Runstats Analyzer uses, includes among others, the number of process instances that are created and deleted, the amount of time each SQL call takes, the number of variables where the length of the variable is increased, and the frequency with which each SQL call is carried out.

#### 9.4.2. Databases

The usage of the various SWoM databases is quite different: the buildtime database is only used sporadically, since the SWoM caches process models; the same is true for the system database. Only the runtime database and, if audit trailing is active, the audit trail database are heavily used. So, from a performance perspective, it is important to put these two databases onto separate disks; this reduces significantly I/O contention between the two databases.

However, if the runtime database is maintained on a separate disk, there still will be significant I/O contention between the accesses to the different tables in the runtime database. Typical examples of heavily used tables that compete for the same I/O are the variable instance table and the activity instance table. So

it is desirable to separate the heavily used tables by putting them on different disks. IBM DB2 addresses this need through the notion of *table spaces* to achieve this objective.

### 9.4.3. Table Spaces

A table space is a separately allocatable (and extensible) part of an IBM DB2 database. Each table needs to be part of a table space; IBM DB2 maintains, for simplicity, a default table space, to which all tables are assigned, that are not explicitly assigned to a table space or if no table space has been defined at all.

There are conceptually many different allocations of the SWoM tables to table spaces. The following allocation of the SWoM tables to table spaces is based to some extent on the proposal of IBM Process Server in the appropriate tuning guide [IBM05c].

- AUDIT contains the audit trail table. It is used heavily if audit trailing is active; if audit trailing is disabled, there is no activity at all against the table space.
- INSTANCE contains all tables that are used for navigation except the variables tables.
- VARIABLES contains the tables that hold variables.
- BPEL contains the process models and the associated deployment descriptors. This table space constitutes the buildtime information. Activity on the table space is generally low, since as already pointed out, the SWoM performs process model caching.

It is obvious that the above specified allocation of tables to table space is just a particular setup. It basically divides the different types of objects into different table spaces within the various databases. There are many more options, including assigning each individual table to its own individual table space. This granularity provides the greatest freedom in assigning tables to physical locations, something typically required by large mainframe installations.

It is the responsibility of system and database administrators to allocate the different table spaces to different disks. Appropriate I/O tracking tools and buffer pool analyzers will help them to carry out the task.

#### 9.4.4. Bufferpools

IBM DB2 keeps data and index records in memory to avoid retrieving them from the physical files. IBM DB2 calls these caches *bufferpools*. Appropriate commands are provided to specify the size of the pages as well as the total number of pages within the bufferpool. Each table space can be assigned to a particular bufferpool provided the size of the pages is compatible with the appropriate page size of the tables in the buffer pool. If no bufferpool is defined for a table space, the table space is assigned to the default bufferpool.

It is desirable that the hit rate of the bufferpool is as good as possible; that means the savings in locating objects in the bufferpool are bigger than the costs of maintaining the bufferpool. Obviously the larger the bufferpool the more hits could be expected; however, searching a large bufferpool requires significantly more resources. IBM DB2 facilitates the finding of the right bufferpool size through appropriate monitoring. Unfortunately, tuning the bufferpool is an iterative process where measurements are taken, the bufferpool (size, number of page cleaners, prefetchers) is adjusted accordingly and new measurements are taken. This process is repeated until the results are satisfactory.

IBM DB2 facilitates the finding of the correct bufferpool sizes through the introduction of the self tuning memory manager (STMM) [M. 07]. The STMM is an autonomic computing feature that eases the task of memory configuration by automatically setting optimal values for most memory configuration parameters, including bufferpools, package cache, locking memory, sort heap, and total database shared memory. When STMM is enabled, the memory tuner dynamically distributes the available memory among the various memory consumers.

#### 9.4.5. Indices

IBM DB2 supports the notion of indices as a means for directly locating data in a table, significantly reducing the number of physical I/Os needed for fetching the data from the disk. In fact, without the use of indices, only very small tables can be processed efficiently. IBM DB2 implements different kinds of indices that help to provide additional benefits beside the simple lookup of a data item. A typical example is clustered indices in which related information is physically located together, improving the access and update performance significantly. The SWoM uses the type of index that is best suited. The following section discusses three aspects of the usage of indices, such as the basic indices that the SWoM needs for performing at least satisfactorily, the usage of clustered indices for achieving better performance, and the usage of the DB2 Design Advisor tool for adjusting the set of indices.

##### 9.4.5.1. Basic Indices

The SWoM comes with a set of indices on the different tables in the buildtime database and, more importantly, in the runtime database. In particular, the SWoM maintains indices for the following objects:

- All objects, such as process instances in the process instance table or variables in the variable instance table, are identified through a universal unique identifier that is generated by the SWoM. This identifier is the primary key of the table that holds the object, so an index is automatically create by IBM DB2. identifier.
- The different objects in the different tables are connected using the primary key of the referenced object. For example, the activity instance table contains the identifier of the associated process model. Typically these fields are defined as foreign keys, however, regardless whether they are part of a referential integrity relation or not they need to carry an index. Without such an index, removal of the dependent object of a process instance, such as activity instances, requires the scanning of the table to locate the objects.

- All information that is accessed through some value of a field always carries an index on this field. A typical example is the correlation table where the correlation set value is indexed so that the appropriate process instance can be found quickly using the correlation information received in a message.

#### 9.4.5.2. Clustering Indices

Clustering indices provide the capability to have the tuples of a table stored on the disk in the order specified in the index definition. Listing 9.3 illustrates how such a clustering index looks for the activity instance table.

```

1  CREATE TABLE ACTIVITY_INSTANCE
2  (
3      AIID          CHAR (12)          NOT NULL ,
4      AID          CHAR (12)          NOT NULL ,
5      PIID          CHAR (12)          NOT NULL
6                                     REFERENCES PROCESS_INSTANCE
7                                     ON DELETE CASCADE,
6      ACT_NR_IN_LINKS  INTEGER          NOT NULL ,
7      STATE        INTEGER          NOT NULL ,
8      PRIMARY KEY (AIID)
9  ) ;
10 CREATE UNIQUE INDEX ACTIVITY_INSTANCE_CLUSTER ON ACTIVITYINSTANCE
11      (PIID,ATID) CLUSTER PCTFREE 10 MINPCTUSED 10 ;

```

Listing 9.3: Clustered Activity Instance Table

The previously simple unique index of Listing 3.13 that is used to locate activity instances based on process instance identifier and the activity identifier is changed to a clustering index. This causes all activity instances of a particular process instance to be grouped together physically. This has tremendous advantages when, within a transaction, multiple activity instances are processed together. If, for example, all activity instances fit on one page, then IBM DB2 must only obtain one page when activities need to be read/written from/to the disk and only one page must be maintained in the appropriate bufferpool. In a nutshell, the clustered index not only reduces the number of I/Os but also improves the usage of the associated bufferpools.

The usage of clustered indices delivers appropriate performance benefits in the following scenarios:

- When the display of the status of a process instance is requested, all activity instances must be retrieved. If all activity instances are grouped together on a page, all information for activities can be retrieved with a single fetch of a page.
- When a process instance is deleted, all activity instances can be physically deleted by just emptying the parts of the page that holds the activity instances.
- When the navigation engine moves from a completed activity instance and creates one or more new activity instances, typically all activity instances are on the same page so that only one physical page write needs to occur.

Other candidates for using a clustering index (for having all objects of a particular process instance) are link instances, variable instances, and correlation instances.

#### 9.4.6. Optimizing the Indices

The DB2 Design Advisor tool assists in improving the performance of IBM DB2 by providing appropriate recommendations for tailoring and customizing IBM DB2 and its tables [ZRL<sup>+</sup>04]. In particular, it makes recommendations for new indices, new materialized query tables (MQT), repartitioning of tables, and the deletion of indices and MQTs. Input to the tool are a set of SQL statements and their relative frequency,

The SWoM can provide significantly better input to the DB2 Design Advisor than the normal way, such as using SQL snapshots or the query patroller. Figure 9.4 illustrates the processing that the system optimizer carries out.

The information stored in the statistics database is extracted by the *SWoM input generator utility*, a component of the system optimizer, and converted into the structure that is expected by the DB2 Design Advisor. The DB2 Design

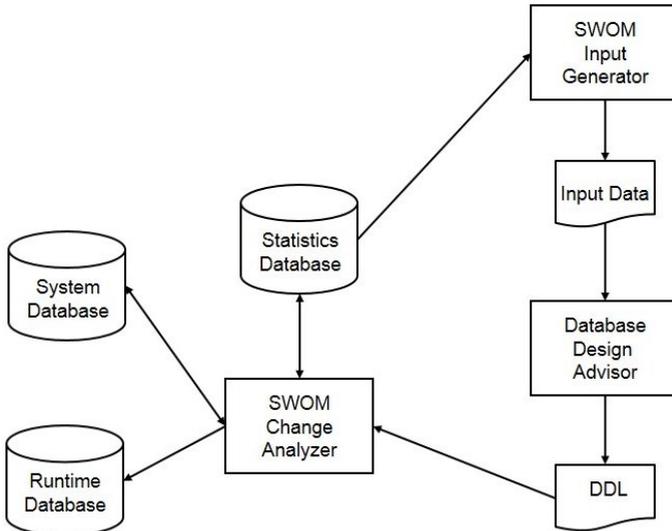


Figure 9.4.: Index Optimization

Advisor then generates DDL statements that contain the appropriate definitions for the indices that should be built or dropped.

These DDL statements are read by the *SWoM change analyzer* utility, another component of the system optimizer. It obtains the current index definitions from the buildtime database, runs the DDL statements that cause changes against the runtime database, stores the new index information in the buildtime database, and finally runs the appropriate IBM DB2 utilities so that IBM DB2 uses the newly provided information.



# QUALITY REDUCTION OPTIMIZATIONS

The performance optimization techniques presented so far do not change the basic processing of the SWoM as a production workflow management system; no compromises have been made with respect to reliability and availability. There are situations, however, where this robustness is not needed and compromises can be made, as the following examples illustrate: for a process that invokes a Web Service which only retrieves data but makes no changes, failure of a particular process instance is tolerable; the client recognizes the situation through an appropriate timeout and just resubmits the request. Or, a Web Service is called which returns information that does not need to be up-to-date; a typical example is an exchange rate used in a product offer (the actual exchange rate will be used when the customer is billed). In these situations, performance can be improved by reducing the execution quality of a process model.

This chapter introduces a set of optimization techniques that achieve the improvement by reducing the service quality of the SWoM; they are appropriately labeled *Quality Reduction Optimization Techniques*.

## 10.1. Persistence Options for Messages

The SWoM uses, as illustrated in Section 3.2.2, a set of queues for communication between façade bean, navigator, and service invoker instances. These messages are persistent messages that are managed in persistent queues and participate in the transaction established by the appropriate component. Both, persistence and transaction participation of the messages, are resource-intensive.

If a process model does not require the robustness of persistent messages, then non-persistent messages could be used. If such a non-persistent message is lost for whatever reason, the associated process instance stays in the running state forever. The SWoM provides appropriate administrative functions to cope with this problem; these administration functions are already available, for example, queries that help detect process instances that have been quiet for some time.

```
1 <processDeploymentDescriptor>
2   <baseOptions>
3     <messagesPersistence>NON_PERSISTENT
4   </messagesPersistence>
5 </baseOptions>
6 </processDeploymentDescriptor>
```

Listing 10.1: Messages Persistence Definition

Listing 10.1 illustrates how the persistence of messages is defined; valid values are `NON_PERSISTENT` indicating that the messages are not persisted, `PERSISTENT` indicating that the messages are persisted.

The option to separately define persistence options for individual message types has not been pursued, since it seems to be quite hard to understand the complication of having a mix of messages with different persistence characteristics. The only meaningful scenario of using different persistence options is for activities that invoke an asynchronous, fire-and-forget type Web Service, whose

execution is not that important. In this case the request can be transmitted to the service invoker via a non-persistent message; the appropriate definition is shown in Listing 10.2.

```
1 <processDeploymentDescriptor>
2   <activity name="A">
3     <invocationMode>NON_PERSISTENT
4   </invocationMode>
5 </activity>
6 </processDeploymentDescriptor>
```

Listing 10.2: Non Persistent Invoke

In SWoM, non-persistent messages are managed by using non-persistent queues, which are defined in IBM WebSphere as EXPRESS\_NON\_PERSISTENT.

The impact of using non-persistent messages depends on the number of messages that are processed for a particular process model. Initial tests using the benchmark process with transaction flow type SHORT and no further optimizations reveal no measurable performance improvements. It can only be speculated why this is so. One explanation could be that the message implementation in IBM WebSphere is optimized for persistent messages and thus processing a non-persistent message does not bring that much. Another explanation could be that the message implementation is so efficient that its impact on the overall execution of a process instance is very low. It should be noted that the benchmark process when optimized fully does not use messages anyhow. So, using non-persistent messages is an optimization technique that needs to be evaluated further.

## 10.2. Transaction-Less Execution

The base architecture of the SWoM assumes that all processing is carried out as transactions. This is achieved by setting the transaction attribute of the façade bean, as shown in Listing 10.3, and having the navigator and service invoker joining the transaction, and by running the navigator and service invoker MDBs transactional.

```
1 @TransactionAttribute(TransactionAttributeType.REQUIRED)
```

### Listing 10.3: Transaction Attribute Setting

This transactional processing is not required, for example, for microflows, that carry out query only processing.

Removing the transactional property reduces the work of IBM WebSphere for managing transactions. Listing 10.4 shows how the process modeler disables transaction processing.

```
1 <processDeploymentDescriptor>
2 <executionOptions>
3 <baseOptions>
4 <transactionProperty>NO_TRANSACTION
5 </transactionProperty>
6 </baseOptions>
7 </executionOptions>
8 </processDeploymentDescriptor>
```

### Listing 10.4: Transaction-Less Execution

The implementation of this performance optimization technique requires a number of changes to the process execution component. First, a different transaction attribute is generated into the façade bean. Second, a new navigator façade bean is added, that runs transaction-less and that is called by the non-transaction façade bean. Third, a new service invoker façade bean is added, that runs transaction-less; it is called from the transaction-less navigator when direct invocation is active. Finally, a set of new queues and navigator and service invoker MDBs are added, so that all parts of an interruptible process run transaction-less.

Initial tests also show no performance improvements. For microflows, the explanation is most likely trivial: since no SQL calls or messages are processed, IBM WebSphere recognizes that no transaction handling needs to be performed. For regular process models, it can be assumed, as in the case of persistent messages, that either processing is optimized for transactional or that the amount of work needed for transaction coordination is so small that it has no influence on the overall performance of the SWoM.

### 10.3. Service Request Result Caching

Some Web Services return data that is somewhat static, that means it does not change that often, so that a requesting client can live with a slightly incorrect value. Furthermore, it is also possible that the requesting client does not need the precision of having up-to-date information. A typical example is a business process that needs an exchange rate for generating a proposal. The response time of a business process as well as the throughput can be improved through caching the results of this type of Web Service in the service invoker.

[SHLP05] proposes such a cache for the enterprise service bus (ESB)[Cha04]; an appropriate cache mediation pattern for service invocation is presented in [RFT<sup>+</sup>05]. [Xue10] shows how the dynamic cache of IBM WebSphere can be exploited for service request caching in IBM Process Server. The Stuttgarter Workflow Maschine implements the caching directly into the workflow engine, which provides better performance, since the actual Web Service call is not carried out. Furthermore the implementation does not rely on any specific proprietary feature of the underlying infrastructure.

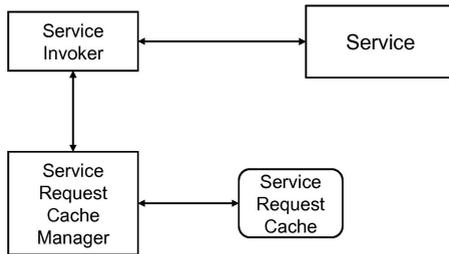


Figure 10.1.: Caching Synchronous Web Services Invocations

Figure 10.1 illustrates how the SWoM implements service cache request caching. Instead of building the cache right into the service invoker, it has been factored out into the Service Request Cache Manager (SRM) component. This design has been chosen since usually several service invoker instances are

active at one time, so with a separate Service Request Cache Manager only one instance of a particular request is cached (and not one in each service invoker). The implementation of the Service Request Cache Manager follows the architecture of all other SWoM caches.

After the service invoker has received an appropriate request from the navigator (either via the service invoker queue or via direct navigation) it checks the request message whether the service request is flagged as cachable. Whether a service request is cachable is defined in the process deployment descriptor, as shown in Listing 10.5.

```
1 <processDeploymentDescriptor>
2   <activity name="H">
3     <serviceRequestCaching>
4       <caching>YES</caching>
5       <invalidationTime>PT5M</invalidationTime>
6     </serviceRequestCaching>
7   </activity>
8 </processDeploymentDescriptor>
```

Listing 10.5: Service Request Caching Definition

If the service invoker receives a cachable service request, it calls the SCRM with the service request and the invalidation time. If the service request is found, the service request answer is returned to the service invoker and the service invoker returns the answer to the navigator.

If the service request is not found, the service invoker is informed about this situation; the same result is returned in case the invalidation time is exceeded, as the SCRM removes the entry from the cache. The service invoker in this case carries out the request and when finished, it provides the request and the result to the SCRM. This design has been chosen to avoid blocking the SCRM, since the requests to the SCRM must be synchronized for cache coherency.

The size of the cache is controlled, as for all other caches within the SWoM, through appropriate settings in the system deployment descriptor. It should be noted that the SWoM just implements the basic concept; the caching of service requests can be driven to all levels of sophistication to obtain the best performance.

Figure 10.2 shows the performance improvements that one achieves. This

	Processes/sec	Improvement (%)
Optimized	54	-
Service Request Caching	111	105

Figure 10.2.: Performance Improvements for Service Request Caching

time the calibration process model has been chosen, since it contains only synchronous Web Services. As can be seen, the impact is rather impressive: the throughput more than doubles. Note that the structure of the process model, where service caching is used for the five invoked Web Services, helps to achieve these significant performance improvements. The benchmark process contains only one only contains one synchronous Web Service, so the achievement performance improvement is only 9 %, as measured in [WRK<sup>+</sup>13].

#### 10.4. Memory-Only Execution

The combination of the different caches, such as intra transaction cache and correlation cache, provide the capability to even carry out an interruptible process model without any database access, having a memory-only execution. Note that microflows are always carried out as a memory-only execution, if the process instance is deleted right away.

An efficient implementation of memory-only execution requires that the process model is transformed into a linear execution, as presented in Section 8.11. Only then can access to the various caches be done without any significant implementation and execution effort for latching and locking objects in the caches. Incidentally, implementing the wait processing that would be needed is not a desired architectural approach in an application server environment. Furthermore, it only makes sense to execute processes in memory that are short running, where short-running is a relative term. If the cache is only used lightly, then even a longer-running business process can stay in memory.

Figure 10.3 shows the performance achievements realized when running the benchmark with memory only. The first line shows the best throughput that is achieved through the presented optimization techniques (see Section 8.12). The second line shows the throughput for the same setup with memory only execution.

	<b>Main Processes/ min</b>	<b>Total Processes/ min</b>	<b>Improvement (%)</b>
Optimized	1400	7000	-
Memory Only	1673	8365	19.5

Figure 10.3.: Performance Improvements for Memory Execution

The performance improvements are quite impressive. The results are the best that can be achieved for SWoM without resorting to intra engine bindings, presented in Section 7.1.1.

# FEATURE OPTIMIZATIONS

The SWoM provides, as many WfMSs do, functions/features besides the ones that are required by the WS-BPEL standard. The purpose of this chapter is not to provide a complete list of features and their optimization, but rather show that from a performance perspective, it is imperative to develop performance optimization techniques for those features if they are used more than occasionally. Three features that are typically provided by a WfMS, and that are implemented in the SWoM, illustrate the point made: audit trail recording, process query functions, and process instance monitoring.

## 11.1. Audit Trail

The purpose of the audit trail is to record the execution history of business processes. Each action that changes the state/context of a process instance is written to an external file that can be later used for various purposes, such as a documentation of the process execution, the improvement of processes by analyzing the execution history, or just for legal reasons [All01].

### 11.1.1.1. Audit Trail Table

The SWoM maintains the audit trail in the table shown in Listing 11.1, following the approach taken by other WfMSs, such as IBM Process Server [IBM]. The audit trail table is, for better administrative control, maintained in a separate database, the audit database; this allows, for example, a system administrator to allocate the audit and the runtime database onto different physical disks, minimizing the impact of the SQL calls carried out for audit and process instance information.

```
1      CREATE TABLE SWOM.AUDITTABLE (
2          ALID                CHAR(12)          NOT NULL
                                PRIMARY KEY,
3          EVENT_TIME          TIMESTAMP          NOT NULL,
4          EVENT_TYPE          SMALLINT          NOT NULL,
5          PIID                CHAR(12)          NOT NULL,
6          PMID                CHAR(12)          NOT NULL,
7          PROCESS_NAME        VARCHAR (256) ,
8          AIID                CHAR(12) ,
9          AID                 CHAR(12) ,
10         ACTIVITY_NAME        VARCHAR (256) ,
11         ACTIVITY_STATE      SMALLINT ,
12         VID                  CHAR(12) ,
13         VARIABLE_NAME        VARCHAR (256) ,
14         VARIABLE_DATA        CLOB(3900K)
15     ) IN AUDITTS;
```

Listing 11.1: Audit Trail Table

The process instance identifier (PIID) identifies the process instance for which the entry is written. The time the event is created is maintained in `EVENT_TIME` (Line 3), the type of event identified via the `EVENT_TYPE` (Line 4). The associated process, variable, or activity instances are identified via the appropriate unique identifiers. Since the audit trail may still need to be analyzed long after the process instance information in the runtime database or even the process model information in the buildtime database has been removed, the appropriate names for WS-BPEL constructs, such as activities, are kept as well (Line 7, Line 10, and Line 13).

### 11.1.2. Audit Trail Manager Architecture

At the heart of the audit trail architecture is, as shown in Figure 11.1, the audit trail manager. It comes in two flavors: one in which the navigator core loads the audit trail manager as a simple class, and one in which the audit trail manager is a separate component. The first one is commonly used; the second one is used for the caching execution mode discussed later in Section 11.1.8.

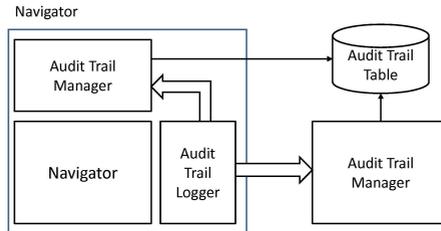


Figure 11.1.: Audit Manager Architecture

The navigator core talks to the audit trail manager via the audit trail logger. The audit trail logger prepares the audit trail record information and then calls the audit trail manager to manage the audit trail records. In fact, it is the audit trail logger that implements the various optimization techniques discussed later.

### 11.1.3. Basic Audit Trail Control

The amount of information that the SWoM writes is specified in the process deployment descriptor, as shown in Listing 11.2.

```
1 <processDeploymentDescriptor>
2   <executionOptions>
3     <auditOptions>
4       <amount>MINIMUM</amount>
5     </auditOptions>
6   </executionOptions>
7 </processDeploymentDescriptor>
```

Listing 11.2: Audit Trail Amount Control

The SWoM supports a rather crude way of specifying the amount of audit trail records that are written by using the single indicator amount element (Line 4). Valid entries are `NO`, indicating that no audit trail records are written, `MINIMUM`, causing only process instance related events to be recorded, `MEDIUM` adding the recording of activity instances and their state changes, and finally `MAXIMUM` adding the recording of variables and their content. Not everything is recorded, such as the truth value of links or the contents of correlation sets. The SWoM also tries to minimize the number of state changes that are recorded for activities; for example, an extra event type is used for recording the fact that an activity instance has been created and then executed right way, instead of using distinct event types, one for the creation of the activity, one for the execution, and one for the finishing of the activity instance.

The SWoM offers no facilities to manage the audit trail, such as removing entries that are no longer needed or moving older entries to some archive store; this is left to the user. Only rudimentary query functions are provided to illustrate what could be done with the audit trail.

The performance impact of audit trailing is quite considerable, in particular if the `MAXIMUM` option is being chosen, as is shown in Figure 11.2. It shows for the benchmark process, for each of the different audit trail amounts, the number of SQL calls that are written to the audit trail and the performance degradation that is observed as a result of the audit trailing.

The benchmark process is run using the best achievable performance (see Section 8.12). In this case, the impact of audit trailing is at peak, since the number of SQL calls used for navigation is minimal and thus the ratio of audit trail related SQL calls to navigation related SQL calls, which represents the overhead, is at the maximum.

The performance numbers show the rather significant impact of audit trailing on the throughput of the SWoM. The SWoM therefore offers options, that allow the process modeler to tailor audit trailing to the particular needs of the process model (with a performance impact as minimal as possible) and whose results are listed in the two last rows in Figure 11.2.

Audit Options	Number of Audit Trail SQL Calls	Throughput (processes /min)	Performance Degradation (%)
None	-	1400	
Minimum	2	1344	3
Medium	16	1052	25
Maximum	21	1008	28
Batched – Medium	3	1342	4
Cached – Medium	< 1	1390	1
Cached- Minimum	< 1	1400	0

Figure 11.2.: Audit Trail Impact

#### 11.1.4. Batching Audit Trail Records

SWoM in its default configuration writes each audit trail record separately. Writing all audit trail records for each navigator transaction of a process instance helps improve performance the same way it does for the transaction cache. Listing 11.3 shows how this mode is selected.

```

1   <processDeploymentDescriptor>
2     <executionOptions>
3       <auditOptions>
4         <eventRecording>BATCHED
5       </eventRecording>
6     </auditOptions>
7   </executionOptions>
8 </processDeploymentDescriptor>
```

Listing 11.3: Event Recording

This optimization technique significantly brings the number of SQL calls for the audit trail down to three (one in each navigator transaction).

### 11.1.5. Event Type Selection

The default audit trail setting, as described in Section 11.1.3, provides only crude control over the information that is written. The SWoM provides finer control by allowing the user to specify explicitly for which event types an audit trail record should be written. Listing 11.4 shows the way the selection of event types is specified.

```
1 <processDeploymentDescriptor>
2   <executionOptions>
3     <auditOptions>
4       <amount>EVENT_TYPES_SELECTED</amount>
5       <eventTypes>
6         <eventType>001</eventType>
7         <eventType>002</eventType>
8       </eventTypes>
9     </auditOptions>
10  </executionOptions>
11 </processDeploymentDescriptor>
```

Listing 11.4: Event Type Selection

The `EVENT_TYPES_SELECTED` value for the `amount` element (Line 4) indicates that the user provides a list of event types for which audit trail records should be written. The list of event types is specified using the `eventTypes` element (Line 5); the individual event types are identified via an appropriate `eventType` element.

The advantage of selecting events is not only that the performance impact is reduced, but also, and more important, the amount of information that is written (and that need to be processed later when the audit log is processed or archived) is also reduced.

### 11.1.6. Context Based Selection

The SWoM writes audit trail records indiscriminately, that means without taking into consideration the importance of a particular process instance. For a loan process, for example, however, it may only be necessary to record information in the audit trail if the loan amount exceeds 100,000 €.

Listing 11.5 shows how one defines that audit trail records should only be written for process instances that handle a loan exceeding 100,000 €.

```
1 <processDeploymentDescriptor>
2   <executionOptions>
3     <auditOptions>
4       <condition>
5         <variable name="loanInformation">
6           <query>loanAmount > 100000</query>
7         </variable>
8       </condition>
9     </auditOptions>
10  </executionOptions>
11 </processDeploymentDescriptor>
```

Listing 11.5: Context Based Selection in Audit Trail

### 11.1.7. Multiple Audit Trails

The different performance options we have discussed so far were assuming a single audit trail; that means a single place to which the audit trail data is written. The advantage of a single audit trail is the ease of use with which the information in the audit trail can be processed. The disadvantage is the contention that is caused by having multiple navigators writing to the same audit trail (particularly if massive amounts of data are written).

The problems with a single audit trail can be solved through the support of multiple audit trails. In a first step, different audit trails are defined to the SWoM via the system deployment descriptor (SSDD); an example is shown in Listing 11.6.

```
1 <systemDeploymentDescriptor>
2   <auditTrails>
3     <auditTrail name="Audit1"/>
4     <auditTrail name="Audit2"/>
5     <auditTrail name="Audit3"/>
6   </auditTrails>
7 </systemDeploymentDescriptor>
```

Listing 11.6: Multiple Audit Trails

The `auditTrails` element (Line 2) is used to define a set of audit trails that are associated with the SWoM. Note that the default audit trail is not defined;

it is allocated when the SWoM is installed. Three audit trails are defined using the `auditTrail` element; for simplicity only the names are shown; obviously further information may be needed so that the audit trail manager has enough information for making the proper requests.

When the SWoM imports the SSDD, it checks whether the audit trail tables have already been allocated. If not, they are created and appropriate entries are defined in IBM WebSphere.

The process modeler associates a process model with a particular audit trail by adding an appropriate entry in the process deployment descriptor, as shown in Listing 11.7.

```
1 <processDeploymentDescriptor>
2   <auditOptions>
3     <auditTrail>Audit1</auditTrail>
4   </auditOptions>
5 </processDeploymentDescriptor>
```

Listing 11.7: Audit Trail Selection

The `auditTrail` element is used to specify the audit trail that should be used for writing audit trail records for instances of the associated process model.

Further research is needed to understand the ramifications of having multiple audit trails, such as the amount of work required to consolidate audit trails if needed.

#### 11.1.8. Audit Trail Caching

Another possibility to improve performance, if the loss of one or more audit trail records is tolerable, is the caching of audit trail records and writing them in larger batches.

```
1 <processDeploymentDescriptor>
2   <auditOptions>
3     < caching>YES</ caching>
4     < batchSize>10</ batchSize>
5   </ auditOptions>
6 </ processDeploymentDescriptor>
```

Listing 11.8: Audit Trail Caching

This function is implemented via the separate audit trail manager; this approach is architecturally cleaner than having it done by the audit trail manager in the navigator.

The caching element (Line 3) enables caching; the `batchSize` indicates after how many navigator transactions the audit trail is written. Figure 11.2 shows that the impact of auditing goes down quite nicely; in particular if caching is active for audit trail mode `MINIMUM` the impact of audit trailing is no longer measurable.

## 11.2. Process Queries

The SWoM supports, as many other WfMSs do, queries either for individual process instances or sets of process instances. A typical query is the request for the number of active process instances for a particular process model, for example the number of active loan processes.

The unoptimized SWoM answers this question by carrying out an appropriate SQL query. The performance of the query depends on a number of factors, such as indexes on the fields that are part of the query and the amount of process instances. Since they usually touch the process instance table, which is the key table for the navigator, one can expect that those queries are having a significant impact on the overall performance of the SWoM.

The easiest way of reducing the impact of queries is by saving the results of the queries in a cache, the *query result cache*. The cache is maintained the same way as all the other caches, such as the intra transaction cache. Listing 11.9 shows the specific properties of the query result cache.

```
1 <systemDeploymentDescriptor>
2   <queryResultCache>
3     <maxNumberQueries>100</maxNumberQueries>
4     <maxQueryResult>4KB</maxQueryResult>
5     <invalidationTime>PT1H</invalidationTime>
6   </queryResultCache>
7 </systemDeploymentDescriptor>
```

Listing 11.9: Query Result Cache Definition

The `QueryCache` element (Line 2) starts the definition of the query result cache. The `maxNumberQueries` defines the number of queries that should be maintained in the cache using a least recently used algorithm for removing older entries if space is needed.

The `MaxQueryResult` specifies the size that the result of a query may not exceed; that means if the size of a particular query result exceeds the specified size it is not cached. The `invalidationTime` specifies the amount of time a query should be answered out of the cache.

The advantage of the query result cache is the capability to answer the same or similar queries very efficiently by retrieving the information from the query result cache. The disadvantage of the approach is that additional cycles and memory are needed to maintain the cache.

No tests have been carried out to assess the performance improvements: the benchmark process is carried out so quickly that at any one time not more than 40 process instances are active. Furthermore all process instances are kept in IBM DB2s buffers, so that all queries can be answered by having IBM DB2 just accessing the buffers.

### 11.3. Process Instance Monitor

The process instance monitor, as described in Section 3.1.1.3, helps users to display the information of a particular process instance, such as the state of the individual activities or the actual contents of all variables. The support of process instance monitoring requires that the SWoM keeps all required information in the runtime database, even information not or no longer needed for processing a process instance.

This extra information that the SWoM must manage in the runtime database degrades the performance of the SWoM. Appropriate options, similar to the ones specified for the audit trail, help the user control the amount of information that should be kept. Listing 11.10 shows how the option is specified in the process deployment descriptor.

```

1 <processDeploymentDescriptor>
2   <executionOptions>
3     <monitoringOptions>
4       <amount>MINIMUM</amount>
5     </monitoringOptions>
6   </executionOptions>
7 </processDeploymentDescriptor>

```

Listing 11.10: Process Instance Monitor Amount Control

The SWoM supports a rather crude way of specifying the amount of information that the SWoM should keep. Valid entries are `NO` indicating that the SWoM must not keep any information, `MINIMUM` causing the current process instance information to be kept, such as the current value and state of variables, `MEDIUM` in addition keeps the complete execution history of the process and activities, and `MAXIMUM` adds to the medium option the keeping of the completed execution history of variables. It should be noted that all performance tests have been carried out with the `MINIMUM` setting. No performance tests have been carried out to assess the impact of the settings.

Obviously the same optimization techniques that have been developed for audit trail control, such as the event type selection specified in Section 11.1.5, can be applied as well.



## TOPOLOGIES

This chapter discusses how the architecture and implementation of the *SWoM* can be easily deployed into distributed infrastructures to deliver increased throughput. In particular the following topologies are presented:

- The single server structure used so far; the information is repeated here to make some points that have not yet been addressed.
- A two tier structure, where IBM DB2 is running on a separate server.
- A structure where several *SWoMs* are running on separate servers and are sharing the same database either on one of the servers or on a separate server.
- An IBM WebSphere cluster topology, where the *SWoM* is deployed to various IBM WebSphere nodes that are part of the cluster.
- An advanced messaging topology where the message queuing functions are delivered by a separate middleware component.

Note that the presented topologies show potential implementation techniques; no measurements have been carried out.

## 12.1. Single Server Structure

The single server topology shown in Figure 12.1 is the topology that has been used so far, and all benchmarks were run with this topology.

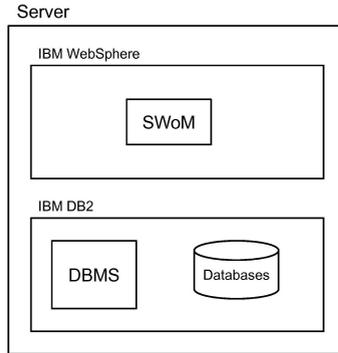


Figure 12.1.: Single Server Topology

All infrastructure components, such as IBM WebSphere and IBM DB2, are residing on the same server. The major advantage of this topology is its simplicity from an administration point of view. It is obvious that the infrastructure components and the SWoM are competing for the same set of resources. Throughput can only be increased by moving to a more powerful server. Eventually the throughput is limited by one or more of the following factors: no larger server is available, the system gets I/O-bound, or one of the infrastructure components or the SWoM does not scale anymore.

## 12.2. Two-Tier Structures

A simple way of increasing the throughput is by moving IBM DB2 to a separate server, as shown in Figure 12.2.

This approach has the advantages that the cycles that IBM DB2 spends are now available to IBM WebSphere and the SWoM.

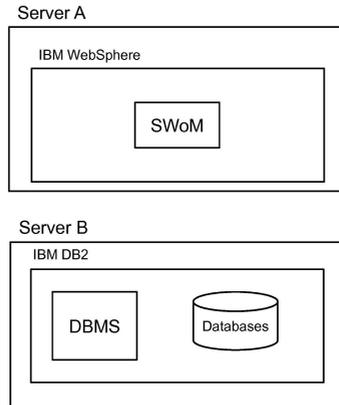


Figure 12.2.: Two Tier Structure

The off-loading of IBM DB2 cycles to another server does not buy as much as one would possibly expect. The amount of cycles spent by IBM DB2 usually makes up, as the benchmark results show, only 10 to 30 % of the overall system load (at least in a fully optimized system).

### 12.2.1. Shared Database

The SWoM addresses the IBM DB2 offloading limitation by allowing multiple SWoM instances running on different servers to share an IBM DB2 instance, as shown in Figure 12.3.

Since the information maintained by all SWoM instances is kept in the same set of databases, the administration functions provided by the SWoM operate directly on IBM DB2; the information about process instances can be obtained from the administrative console of every participating SWoM.

The biggest problem with this approach is the level of administration that is necessary to manage the deployment of processes onto the various servers. Conceptually, it would be possible to deploy the same process model onto different servers; however each process model would have its interface at a different endpoint, something that may be feasible but is hardly easy to manage.

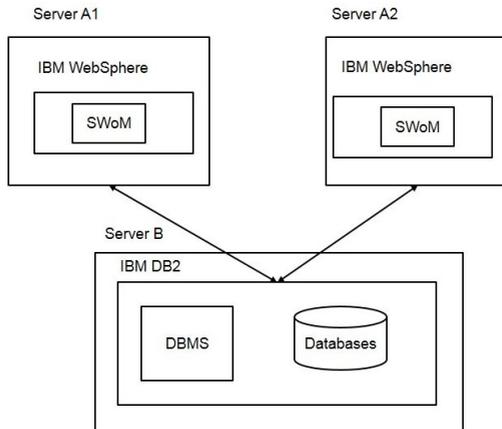


Figure 12.3.: Shared Database

So one usually ends up with having a process model only deployed onto a single server. The net result is a set of process models spread over several servers, making the proper balancing of load to the individual servers cumbersome. For example, if a server can not sustain the load anymore, process models must be moved from one server to another.

Several features of the SWoM, such as intra transaction or correlation caching, are dependent on the affinity to a server (or more precisely a SWoM instance). Additional techniques are required to further benefit from the performance improvements delivered by those features; they are described later in Section 12.5.

### 12.3. IBM WebSphere Cluster

The administration burden of the shared database approach can be leveraged by having the different IBM WebSphere installations on the various servers grouped into an IBM WebSphere cluster as shown in Figure 12.4.

All requests entering the cluster are forwarded to appropriate instances of the

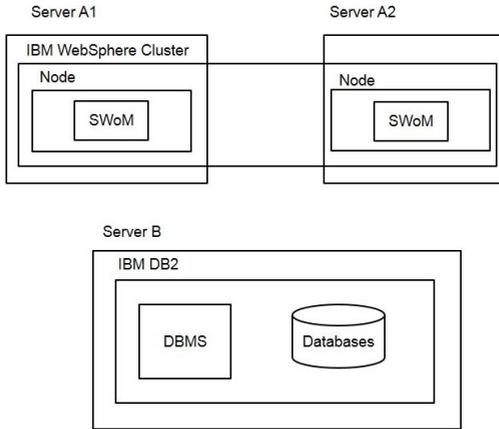


Figure 12.4.: IBM WebSphere Cluster

SWoMs residing in the different nodes of the cluster. The cluster determines the optimal node based on information, either supplied by the cluster administrator or determined through load balancing algorithms. Furthermore, the cluster provides fail-over-support so that work is moved to another server if necessary.

Since IBM WebSphere now controls the allocation of requests to servers, it is not possible to know which SWoM processes a request for a particular process instance. In fact, a particular process instance may be carried out by many different WfMS instances; this situation occurs if two parallel paths of the process instance are processed at the same time.

This approach suffers from the same disadvantage as the previous one with respect to certain SWoM optimization techniques; the solutions introduced in Section 12.5 also help to overcome the disadvantages.

## 12.4. Advanced JMS Messaging

The JMS messaging used in IBM WebSphere is provided by an internal messaging engine, whose persistence is provided either through the supplied

Derby database or through an enterprise class database, such as IBM DB2 with supposedly improved performance, in particular if managed on an extra server.

Another approach goes even further. Instead of just replacing the database with an enterprise class database, the complete messaging engine is replaced with an enterprise class messaging engine. Figure 12.5 reflects the new system structure.

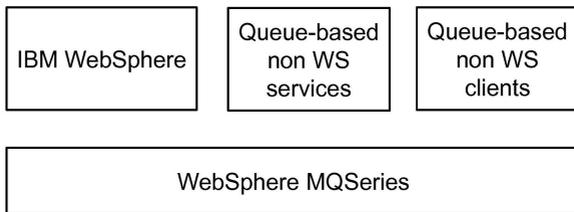


Figure 12.5.: Advanced JMS Messaging

IBM WebSphere MQSeries delivers an enterprise messaging bus that can be used by all applications. IBM WebSphere provides the capability to use queues that are defined in IBM WebSphere MQSeries. The move from the IBM WebSphere internal messaging engine to the usage of IBM WebSphere MQSeries is just, besides the added costs, an administrative change, that does not require any change to SWoM.

## 12.5. Cluster-Level Caching

Both caches, the intra transaction and the correlation cache, no longer work properly in a cluster environment, since different pieces of a process instance may be executed on different nodes in the cluster, so that there is no longer an affinity between a process instance and a server.

The SWoM uses two strategies to cope with this situation: the usage of a shared intra transaction cache and the usage of node spheres.

### 12.5.1. Shared Intra Transaction Cache

The inter transaction cache is changed so that the ITCM is shared between the different navigators running on the different nodes. Figure 12.6 shows the appropriate topology.

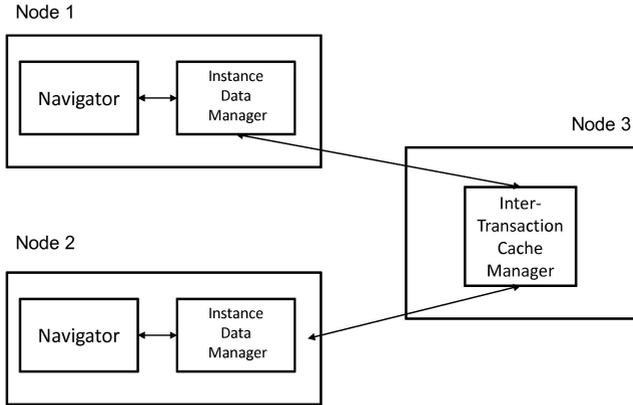


Figure 12.6.: Shared Intra Transaction Cache

The processing is the same as described earlier. However, each of the interactions between the navigator and the ITCM are carried out as remote procedure calls. This requires that the cache objects need to be serialized and de-serialized when exchanged between the navigator and the ITCM. Additional work is needed to determine how much of the performance improvements that the Intra Transaction Cache delivers are offset by the additional cycles needed for serialization/deserialization and cross-system communication.

### 12.5.2. Node Spheres

Node spheres are an approach to optimize the processing of business processes in a clustered environment by having a complete business process or even a part of a business process to be carried out on a particular node within a cluster. One of the purposes of assigning a business process to a particular node is to

have some affinity between the SWoM and the invoked Web Services. Another purpose is the reuse of the process instance information stored in the ITC on the selected node.

Listing 12.1 illustrates how one would assign the execution of instances of a particular process model to a particular node.

```
1 <processDeploymentDescriptor>
2   <clusterExecution>
3     <executionNode mode="nodeFixed">
4       <node name="node1"/>
5     </executionNode>
6   </clusterExecution>
7 </processDeploymentDescriptor>
```

Listing 12.1: Assign Process Model to a Fixed Node

The `clusterExecution` element provides for the definition of properties that apply to the handling of the process in a cluster environment. The enclosed `executionNode` element specifies how the assignment of processes to a node should be done. The `mode` attribute specifies how the node is selected, the value `nodeFixed` indicates that the processes should be carried out on the node(s) identified through enclosed `node` element. Another option of selecting a node is shown in Listing 12.2.

```
1 <processDeploymentDescriptor>
2   <clusterExecution>
3     <executionNode mode="nodeInitiated">
4     </executionNode>
5   </clusterExecution>
6 </processDeploymentDescriptor>
```

Listing 12.2: Assign Process Model to an Initial Node

The `mode` attribute value is now set to `nodeInitiated` which specifies that the process should be carried out on the node on which the process instance was created. In this case, the process instance is always executed on the node that keeps the process information cached.

In the two previous approaches, the definitions were applied to complete processes; Listing 12.3 shows how a node sphere could be defined for a part of a process.

```

1   <processDeploymentDescriptor>
2     <clusterExecution>
3       <nodeSphere>
4         <executionNode mode="nodeInitiated">
5           </executionNode>
6         <activities>
7           <activity name="A"/>
8           <activity name="B"/>
9           <activity name="C"/>
10        </activities>
11      </nodeSphere>
12    </clusterExecution>
13  </processDeploymentDescriptor>

```

Listing 12.3: Defining a Node Sphere

The `nodeSphere` assigns to a set of activities a node sphere. The `executionNode` identifies where the activities in the node sphere identified by the enclosed `activities` element should be executed. In the example, the activities A, B, and C are executed on the node on which the first of the three activities was carried out.

```

1   CREATE TABLE CORRELATION_INSTANCE
2   (
3     CTID          CHAR (12)          NOT NULL ,
4     PMID          CHAR (12)          NOT NULL ,
5     NODE_ID      INTEGER ,
6     PIID         CHAR (12)          NOT NULL
7               REFERENCES PROCESS_INSTANCE
8               ON DELETE CASCADE,
9     VALUE        BLOB (2000k) ,
10  ) ;

11  CREATE UNIQUE INDEX ON CORRELATION_INSTANCE
12  WITH (VALUE,PIID,PMID,CTID) ;

```

Listing 12.4: Node-Aware Correlation Instance Table

The newly added `NODE_ID` field holds the node identifier of the node that created the node sphere (or the process instance). When a new message is received, independent of the mechanism being used, a navigator is invoked as usual. It contacts as usual the correlation manager to obtain the process instance identifier.

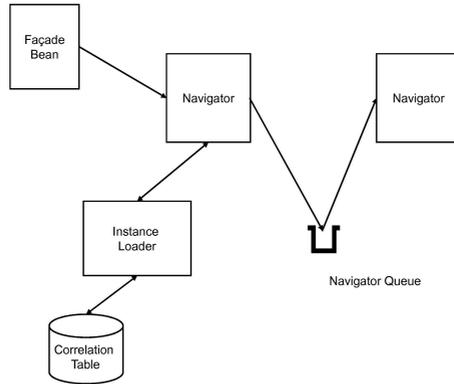


Figure 12.7.: Node Sphere Processing

If the node identifier in the entry is set to some node identifier and the navigator is running on a different node, it inserts an appropriate message into the navigation queue of the appropriate node. The message is a modification of the standard message as it already contains the process instance identifier, so there will be no need for the called navigator to fetch the correlation information again.

The disadvantage of the approach is the additional transaction with two message calls that is required when the request must be dispatched to a different node. On the other hand, there is no need to carry out serialization and deserialization of objects. In addition, if the process is continued by a SWoM internal message, then the message can be put into the queue that is local to the node. In this case, no extra transaction is required.

It should be noted that the notion of node spheres goes beyond the reuse of process instance information in subsequent navigation steps. The outlined approach assumes that the initial node is selected randomly and that all subsequent navigation steps within the affinity spheres are carried out on the same node. Another option would be to tag the node sphere to a particular node; reasons for doing so may be the locality of resources that are used within the node sphere.

Significant more work is needed to understand the ramifications and benefits of the node sphere approach.



## SUMMARY AND OUTLOOK

### 13.1. Summary

This thesis presented the architecture of a high throughput, non-distributed WfMS; the implementation was labeled appropriately Stuttgarter Workflow Maschine (SWoM) to honor its birth place. The basic architecture was that of a stateless server running navigation and service invocation as transactions, storing the results of the transactions in a persistent storage, and using stratified transactions to complete processes run as transactions [LR00c].

This thesis introduced a set of novel concepts for the performance optimization of a WfMS.

- The notion of transaction flows as the base for optimizing the execution of flows. Transaction flows consist of a set of transactions whose stratified execution causes the transacted execution of process instances.
- The concept of caching has been driven into virtually all areas of the different components that make up the SWoM: (1) The caching of process models, so that process model information needs to be read only once from the database, (2) the caching of process instance information during

the execution of a transaction, so that process information is read only once and can be updated using SQL batch update, reducing the number of SQL calls quite significantly, (3) the caching of correlation information to eliminate the need for query of the database via correlation information when using asynchronous messaging for high speed interactions with Web Services, and (4) the caching of process instance data between two subsequent processing steps of a process instance, eliminating not only the need for fetching the information from the database but also the re-construction of the internal process instance representation from the database representation.

- The extensive exploitation and dynamic adaptation of the underlying infrastructure, IBM WebSphere for the application server and IBM DB2 as the database environment, resulting in significant performance improvements without any impact on the robustness of the SWoM.
- The notion of flow configuration options, that help the process modeler to improve performance by telling the SWoM to exploit for a particular process model a set of pre-defined optimization approaches. It includes, for example, the direct invocation of Web Services without using the service invocation component reducing the number of transactions and messages that need to be processed, or the efficient handling of LOBs in the database, making the processing of variables or correlation information significantly more efficient.
- A flow optimizer that optimizes the execution of the transaction flows with respect to cache, database, and CPU cycle usage using data flow and transaction analysis techniques. Optimization is done based on user recommendations and statistical information that the SWoM collects during execution, providing significant performance improvements, in particular the approach of persisting the serialized process instance transaction level cache to the database which brings down the number of SQL calls in a transaction to not more than two or three.
- Several optimization techniques have been added that achieve the im-

proved performance by executing process instances without the normal robustness. These techniques include the usage of non-persistent queues for exchanging information between the different transactions that make up the execution of process instance or the execution of a process instance totally in memory.

The performance of the SWoM has been calibrated using a publicly available benchmark for WS-BPEL engines with the net result that the SWoM outperforms the second-best engine.

Since this calibration benchmark is a very simple process that a smart workflow management system carries out as a micro flow, a new benchmark process has been developed that more realistically represents real world scenarios, such as correlation processing or processes that contain join activities. Only such a process can show the impact of the performance optimizations that have been developed.

The results of the individual optimization techniques show an overall performance improvement of approximately 60 % on an already high performant workflow engine. Note that the benchmark used is short-running, so larger performance improvements can be expected for long-running processes, where the impact of the database on the overall performance is larger so that the database related optimization techniques play a larger role.

## 13.2. Outlook

The work done for the thesis just forms the base of significant work that lays ahead. Section 8.14 defines a road map with the following major areas:

- *Cost-Based Transaction Flow Type Selection* introduces the first step in the direction of cost-based optimization by assigning costs to the actions within a transaction, such as SQL calls or message processed. This information allows the flow optimizer to determine the best transaction flow type.

- *Statistics-supported Flow Optimization* drives the statistics support forward by automatically determining if the statistical information changes and automatically modifying the appropriate flow execution plans. The necessary versioning support is augmented by appropriate version migration, so that the changed statistical information can be exploited by running process instances.
- *Feature Completion* extends the existing support of flow optimization to all constructs of the WS-BPEL specification. Of particular importance is the support of compensation spheres, that requires additional statistical information for any decent optimization.
- *Dynamic Transaction Boundaries* extend the fixed transaction flow structures defined by the transaction flow types to the dynamic cost-based construction of transaction boundaries.
- *Application-Specific Optimization* addresses optimization techniques that are important for certain application types, such as applications that use massive amounts of data.
- *Business Goals Optimization Goals* extend the optimization objective from throughput to business goals, such as the minimizing of the external costs associated with a business process, for example the costs of the goods that are ordered in a purchasing process.

The author believes that optimization in workflow management systems is far more complex than the optimization of relational database management systems. So we are talking here about significant efforts to address just a few pieces in the road map. Given the number of people involved in database research, including the various research groups of commercial database vendors, support this statement.



# BENCHMARK

The purpose of this appendix is to provide detailed information about the benchmark that is used for evaluating the performance improvements associated with the different optimization techniques and that has been introduced in Chapter 4.

This appendix shows the WSDL, the WS-BPEL definition, the process deployment descriptor, and the flow execution plan for the main process in the benchmark as well as one of the processes that implements the Web Service that the main process invokes via an `invoke` activity and whose response the main process processes via a subsequent `receive` activity.

The purpose of the listing here is to provide a reference when explaining some of the optimization techniques in the main sections. Only the most important pieces of information are shown to minimize the number of pages required.

## A.1. Structure

Figure 4.2 shows only the basic structure of the main process that invokes a set of Web Services. Figure A.1 shows a more detailed picture using BPEL4Chor

introduced by [DKLW09].

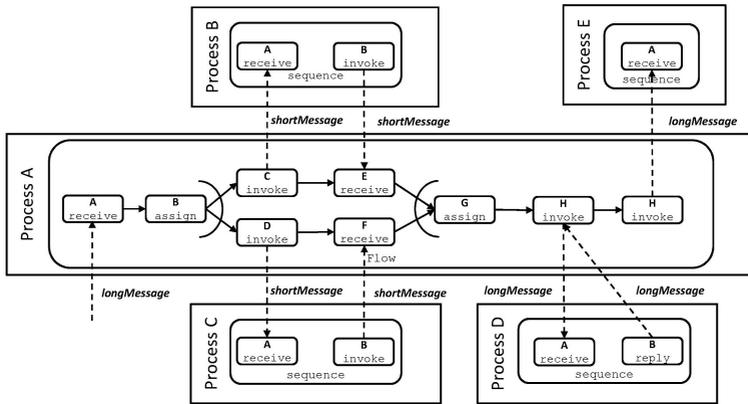


Figure A.1.: Benchmark Choreography

Process A is the core of the benchmark: It is activated by a `longMessage` request; the message contains two fields. The assign activity B creates two variables of message type `shortMessage`, containing a single field, by moving the first field of the received message into the first variable, and the second field into the second variable. The first variable is then used in the invoke activity C to start Process B; the second variable in the invoke activity E to start Process C. Both activities create appropriate correlation set instances, which are then used when processing the messages returned by the called processes in the receive activities to locate the proper process instance.

## A.2. Process A

The information provided for the main process is the following:

Section A.2.1 presents the WSDL. For ease of use and referencing, the WSDL information for the called processes is factored in the WSDL main process. Note that only the logical parts of the WSDL are shown; the physical parts,

such as the service and port type definitions are generated by the SWoM upon deployment of the process.

Section A.2.2 shows the WS-BPEL definition that is associated with the process. The process is defined using links; the provided information is somewhat shortened, for example, of the two identical parallel path only one of them is shown.

Section A.2.3 lists the associated process deployment descriptor. It provides the binding and service information that is needed for the SWoM to carry out the appropriate Web Service invocations. Flow configuration options are specified for variables, such as the length of the variable, activities, such as the invocation mode, and for correlation sets , such as the length of the correlation sets.

Section A.2.4 demonstrates the structure of the associated flow execution plan that the flow optimizer generates by exploiting the following capabilities: intra transaction caching, transaction flow type ULTIMATE, correlation information caching, optimized variable access, XPath processing optimization, direct invocation, variable length, and correlation length optimization.

### A.2.1. WSDL Definition

The WSDL for the benchmark process starts, as shown in Listing A.1 with the definition of the name of the WSDL and the appropriate name spaces. The name space for the WSDL is the one specified in the targetNamespace. The data types that are defined in the WSDL for use by the process model are defined in their own name space.

```
1      <?xml version="1.0" encoding="UTF-8"?>
2      <wsdl:definitions
3          name="ProcessA"
4          xmlns:soap=
5              "http://schemas.xmlsoap.org/wsdl/soap/"
6          xmlns:tns=
7              "http://iaas.perfTest.org/wsdl/ProcessA/"
8          xmlns:wSDL=
9              "http://schemas.xmlsoap.org/wsdl/"
10         xmlns:xsd=
11             "http://www.w3.org/2001/XMLSchema"
12         xmlns:plnk=
```

```

9         "http://docs.oasis-open.org/wsbpel/2.0/plnktype"
    xmlns:vprop=
10        "http://docs.oasis-open.org/wsbpel/2.0/varprop"
    targetNamespace=
11        "http://iaas.perfTest.org/wsdL/ProcessA/"
    xmlns:dt=
        "http://iaas.perfTest.org/datatypes">

```

Listing A.1: WSDL Definition for Process A

The benchmark process uses WSDL messages, so the appropriate types need to be specified as shown in Listing A.2. Two messages are defined, meaningfully labeled `longMessage` for a message structure that contains two fields and `shortMessage` for a message structure that contains a single field.

```

1  <wsdl:types>
2  <xsd:schema
      targetNamespace=
        "http://iaas.perfTest.org/datatypes"
      xmlns:dt=
        "http://iaas.perfTest.org/datatypes">
3  <xsd:element name="longMessage">
4  <xsd:complexType>
5  <xsd:all>
6  <xsd:element name="field1"
          type="xsd:string"/>
7  <xsd:element name="field2"
          type="xsd:string"/>
8  </xsd:all>
9  </xsd:complexType>
10 </xsd:element>
11 <xsd:element name="shortMessage">
12 <xsd:complexType>
13 <xsd:all>
14 <xsd:element name="field"
          type="xsd:string"/>
15 </xsd:all>
16 </xsd:complexType>
17 </xsd:element>
18 </xsd:schema>
19 </wsdl:types>

```

Listing A.2: Type Definitions

These two messages structure are then used in creating two messages, similarly meaningful labeled `longMessage` and `shortMessage` as shown in Listing A.3.

```

1 <wsdl:message name="longMessage">
2   <wsdl:part element="dt:longMessage"
      name="longMessage"/>
3 </wsdl:message>
4 <wsdl:message name="shortMessage">
5   <wsdl:part element="dt:shortMessage"
      name="shortMessage"/>
6 </wsdl:message>

```

### Listing A.3: Message Definitions

The benchmark process uses correlation processing, so appropriate property definitions must be provided. Listing A.4 The correlation property is of type `String` and is extracted from the `shortMessage` using the XPath-Query specified in `vprop:query`; in fact, it extracts the contents of the field `field`.

```

1 <vprop:property name="correlationProperty"
      type="xsd:string"/>
2 <vprop:propertyAlias
      propertyName="tns:correlationProperty"
      messageType="tns:shortMessage"
      part="shortMessage">
3   <vprop:query>//field</vprop:query>
4 </vprop:propertyAlias>

```

### Listing A.4: Correlation Property Definitions

Listing A.5 shows the port types that the benchmark process exposes. The first one (Line 1) is the port type used to start the process. The two other ones (Line 2 and Line 3) are here to receive the response from two Web Services that have been invoked.

```

1 <wsdl:portType name="ProcessAPT">
   <wsdl:operation name="start">
     <wsdl:input message="tns:longMessage"/>
   </wsdl:operation>
</wsdl:portType>

2 <wsdl:portType name="ProcessBCallbackPT">
   <wsdl:operation name="receiveResponseFromProcessB">
     <wsdl:input message="tns:shortMessage"/>
   </wsdl:operation>
</wsdl:portType>

3 <wsdl:portType name="ProcessCCallbackPT">
   <wsdl:operation name="receiveResponseFromProcessC">

```

```

        <wsdl:input message="tns:shortMessage"/>
    </wsdl:operation>
</wsdl:portType>

```

### Listing A.5: Process Port Type Definitions

Listing A.6 shows the partner link type definitions for the process. The first one is, named ProcessALT is the partner link type that is used for connecting the initial requester with the process.

The two other partner link types are the links for the interaction with Process B and process C, or more accurately the Web Services that the processes represent.

```

1  <plnk:partnerLinkType name="TTProcALT">
2      <plnk:role name="ProcessA"
           portType="tns:ProcessAPT"/>
3  </plnk:partnerLinkType>

4  <plnk:partnerLinkType name="ProcessBLT">
5      <plnk:role name="ProcessB"
           portType="tns:TTProcBPT"/>
6      <plnk:role name="CallbackForProcessB"
           portType="tns:ProcessBCallbackPT"/>
7  </plnk:partnerLinkType>

8  <plnk:partnerLinkType name="ProcessC">
9      <plnk:role name="ProcessC"
           portType="tns:ProcessCPT"/>
10     <plnk:role name="CallbackForProcessC"
           portType="tns:ProcessCCallbackPT"/>
11 </plnk:partnerLinkType>

```

### Listing A.6: Partner Link Type Definitions

No binding and service information is provided in the WSDL, since the SWoM automatically generates appropriate binding and service information when the process is deployed into IBM WebSphere. The final WSDL can be obtained by using the administration console of IBM WebSphere. It should be noted, that it would be desirable to allow the process modeler to specify the binding and service information; however this would be a significant development whose investment can only be justified for a commercial product. It does not add any new findings to the purpose of the thesis of finding performance optimization techniques.

## A.2.2. BPEL Definition

The benchmark process is presented in several listings so that it is easier to comment the different pieces without losing sight or using cumbersome references.

Listing A.7 shows the basic definition of the process model providing the name, the associated namespaces and the location (see Line 9), via the `import` element, of the WSDL that is associated with the process model.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2   <process name="ProcessA"
3     suppressJoinFailure="yes"
4     targetNamespace=
5       "http://iaas.perfTest.org/ProcessA"
6     xmlns=
7       "http://docs.oasis-open.org/wsbpel/2.0/process/executable"
8     xmlns:bpws=
9       "http://docs.oasis-open.org/wsbpel/2.0/process/executable"
10    xmlns:tns=
11      "http://iaas.perfTest.org/wsd/ProcessA/"
12    xmlns:xsd=
13      "http://www.w3.org/2001/XMLSchema">
14
15   <import location="ProcessA.wsdl"
16     namespace=
17       "http://iaas.perfTest.org/wsd/ProcessA/"
18     importType=
19       "http://schemas.xmlsoap.org/wsdl/">
```

Listing A.7: BPEL Process

Listing A.8 shows the partner links that the process model needs. The first one in Line 2 is the partner link for the client that invokes the process model and whose request is handled by the first receive activity. The next two partner links (Line 3 and Line 4) are for those two Web Services that are invoked in activities C and D and whose response is handled by the receive activities E and D (and that are used to test correlation processing). Line 5 shows the partner link for the Web Service that is invoked synchronously in activity H. Finally, Line 6 shows the partner link for the last Web Service that is invoked asynchronously and who call the process back which is then handled via appropriate receive activities. The last one in Line 6 shows the partner link for the fire-an-forget Web Service called in the last activity.

```

1 <partnerLinks>
2   <partnerLink name="ProcessAPL"
      myRole="ProcessA"
      partnerLinkType="tns:ProcessALT"/>
3   <partnerLink name="processBPL"
      myRole="CallbackForProcessB"
      partnerRole="ProcessB"
      partnerLinkType="tns:ProcessBLT"/>
4   <partnerLink name="ProcessCPL"
      myRole="CallbackForProcessC"
      partnerRole="ProcessC"
      partnerLinkType="tns:ProcessCLT"/>
5   <partnerLink name="ProcessDPL"
      partnerRole="ProcessD"
      partnerLinkType="tns:ProcessDLT"/>
6   <partnerLink name="ProcessEPL"
      partnerRole="ProcessE"
      partnerLinkType="tns:ProcessELT"/>
7 </partnerLinks>

```

Listing A.8: Partner Link Definitions

The benchmark uses a set of variables that are defined using the messages defined in the WSDL as shown in Listing A.9. It should be noted that more variables have been defined than what would be actually needed. This has been done to reflect more heavily the amount of CPU cycles needed for data manipulation in assign activities. Variables that are the target of copy operations in assign activities are appropriately initialized (the WS-BPEL specifications are slightly weak in this respect). As can be seen in Line 2, the `outRequest1` variable, for example, is initialized with the `fIELD`. Without such an initialization, a WS-BPEL fault is thrown, when the `fIELD` is used in an appropriate query for the `to` specification (see Line 10).

```

1 <variables>
2   <variable name="inRequest"
      messageType="tns:longMessage"
      <variable name="outRequest1">
        messageType="tns:shortMessage"
        <from>
          <literal><![CDATA[<field/>]]</literal>
        </from>
      </variable>
3   <variable name="outRequest2"
      messageType="tns:shortMessage"/>
      <from>

```

```

        <literal><![CDATA[<field/>]]></literal>
    </from>
</variable>
4   <variable name="inResponse1"
      messageType="tns:shortMessage"/>
5   <variable name="inResponse2"
      messageType="tns:shortMessage"/>
6   <variable name="inInvoke"
      messageType="tns:longMessage">
    <from>
        <literal><![CDATA[<field1/></field2>]]></literal>
    </from>
    </variable>
7   <variable name="outInvoke"
      messageType="tns:longMessage"/>
8   </variables>

```

Listing A.9: Variable Definitions

Listing A.10 shows the two correlation sets that are needed for correlating the messages exchanged between Process A and Process B and Process C, respectively.

```

1   <correlationSets>
2   <correlationSet name="correlation1"
      properties="tns:correlationProperty"/>
3   <correlationSet name="correlation2"
      properties="tns:correlationProperty"/>
4   </correlationSets>

```

Listing A.10: Correlation Set Definitions

The process model is using flow constructs exclusively, so that appropriate links must be defined as shown in Listing A.11. It should be noted, that one could have also used the algebraic constructs to describe the process model; however the author is, for obvious reasons, more familiar with the graph notation.

```

1   <flow>
2   <links>
3   <link name="linkAB"/>
4   <link name="linkBC"/>
5   <link name="linkBD"/>
6   <link name="linkCE"/>
7   <link name="linkDF"/>
8   <link name="linkEG"/>

```

```

9         <link name="linkFG"/>
10        <link name="linkGH"/>
11        <link name="linkHI"/>
12    </links>

```

### Listing A.11: Link Definitions

Listing A.12 shows the initial receive activity. It receives a `longMessage` which contains two field and stores it in the variable `inRequest`. The activity is source of one outgoing link as shown in Line 3; the appropriate target is activit B.

```

1    <receive createInstance="yes"
        name="A"
        operation="start"
        partnerLink="TTProcASPPL"
        portType="tns:TTProcASPPT"
        variable="inRequest">
2        <sources>
3            <source linkName="linkAB"/>
4        </sources>
5    </receive>

```

### Listing A.12: Initial Receive Activity

Listing A.13 shows the assign activity that extracts each field in the variable that has been received and creates appropriate variables from the fields. These variables are later used as input to the activities C and E, respectively.

```

1    <assign name="B">
2        <targets>
3            <target linkName="linkAB"/>
4        </targets>
5        <sources>
6            <source linkName="linkBC"/>
7            <source linkName="linkBD"/>
8        </sources>
9        <copy>
10           <from variable="inRequest" part="longMessage">
11               <query>//field1</query>
12           </from>
13           <to variable="outRequest1" part="shortMessage">
14               <query>//field</query>
15           </to>
16        </copy>
17        <copy>
18           <from variable="inRequest" part="longMessage">
19               <query>//field2</query>

```

```

20         </from>
21         <to variable="outRequest2" part="shortMessage">
22             <query>//field</query>
23         </to>
24     </copy>
25 </assign>

```

Listing A.13: Assign Activity

The queries in Line 10 and Line 18 extract the different fields; the appropriate queries in the target are needed so that the extracted field values can be assigned to the field in the variable which is of type `shortMessage`.

As can be seen, the assign activity is the source of two links, resulting in the execution of two different, parallel, paths.

Listing A.14 shows one of the two paths, which are basically identical, except for the Web Service that is being invoked. The path consists of two activities: (1) an invoke activity that calls the Web Service that is implemented by the appropriate WS-BPEL process and creates the appropriate correlation set instance and (2) a receive activity that receives the answer from the called Web Service and uses the correlation set information in the response to find the appropriate process instance.

```

1     <invoke name="C"
        operation="start"
        partnerLink="ProcessBPL"
        portType="tns:ProcessBPT"
        inputVariable="outRequest1">
2     <targets>
3         <target linkName="linkBC"/>
4     </targets>
5     <sources>
6         <source linkName="linkCE"/>
7     </sources>
8     <correlations>
9         <correlation initiate ="yes" set="correlation1"/>
10    </correlations>
11 </invoke>
12 <receive name="E"
        operation="receiveResponseFromProcessB"
        partnerLink="ProcessBPL"
        portType="tns:ProcessBCallbackPT"
        variable="inResponse1">
13 <targets>
14     <target linkName="linkCE"/>

```

```

15     </targets>
16     <sources>
17         <source linkName="linkEG"/>
18     </sources>
19     <correlations>
20         <correlation set="correlation1"/>
21     </correlations>
22 </receive>

```

Listing A.14: Parallel Path

The two parallel paths are then joined together in the assign activity shown in Listing A.15. The assign activity takes the responses returned by the two Web Services and creates a new variable `inInvoke` that is used later by an invoke activity. Note, that the assign activity actually mirrors the processing of the assign activity B earlier in the process.

```

1     <assign name="G">
2         <targets>
3             <target linkName="linkEG"/>
4             <target linkName="linkFG"/>
5         </targets>
6         <sources>
7             <source linkName="linkGH"/>
8         </sources>
9         <copy>
10            <from variable="inResponse1" part="shortMessage">
11                <query>//field</query>
12            </from>
13            <to variable="inInvoke" part="longMessage">
14                <query>//field1</query>
15            </to>
16        </copy>
17        <copy>
18            <from variable="inResponse2" part="shortMessage">
19                <query>//field</query>
20            </from>
21            <to variable="inInvoke" part="longMessage">
22                <query>//field2</query>
23            </to>
24        </copy>
25    </assign>

```

Listing A.15: Join Assign Activity

The newly created variable is now used in a synchronous invoke activity (Listing A.16, that accepts the variable as input and returns the variable unchanged,

which is then stored in a new variable outInvoke.

```
1      <invoke name="H"
        operation="start"
        partnerLink="ProcessDPL"
        portType="tns:ProcessDPT"
        inputVariable="inInvoke"
        outputVariable="outInvoke">
2      <targets>
3      <target linkName="linkGH"/>
4      </targets>
5      <sources>
6      <source linkName="linkHI"/>
7      </sources>
8      </invoke>
```

Listing A.16: Synchronous Invoke Activity

The final activity is, as shown in Listing A.17 a one way invoke activity that is carried out but does return any message. This activity completes the process model.

```
1      <invoke name="I"
        operation="start"
        partnerLink="ProcessEPL"
        portType="tns:ProcessEPT"
        inputVariable="outInvoke">
2      <targets>
3      <target linkName="linkHI"/>
4      </targets>
5      </invoke>
6  </flow>
7 </process>
```

Listing A.17: One Way Invoke Activity

### A.2.3. Process Deployment Descriptor

Each process model in the SWoM is associated with a process deployment descriptor, that provides two major pieces of information: (1) the end points for the Web Service that are called from within the process and (2) configuration options that help improve performance.

Listing A.18 shows the header of the process deployment descriptor: the processModel defines the process model with which the process deployment

descriptor is associated with, `http://shared.swom.iaas/spdd/` identifies the namespace of the process deployment descriptor.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <tns:DeploymentDescriptor processModel="ProcessA"
3     xmlns:tns=
4         "http://shared.swom.iaas/spdd/"
5     xmlns:wSDL=
6         "http://schemas.xmlsoap.org/wSDL/"
7     xmlns:soap=
8         "http://schemas.xmlsoap.org/wSDL/soap/"
9     xmlns:wsa=
10        "http://schemas.xmlsoap.org/ws/2004/08/addressing"
11    xmlns:xsi=
12        "http://www.w3.org/2001/XMLSchema-instance">
```

### Listing A.18: Process Deployment Descriptor Header

The process deployment descriptor consists, as described in Section 3.7.2, of two parts: a part that specifies information for the individual partner links so that the related Web Services can be invoked, and a second part that specifies flow configuration information for the process model. Listing A.19 shows, exemplarily, the entry for the partner link `ProcessBPL` which associates the `invoke` activity `C` with the appropriate Web Service `ProcessB`.

```
1 <PartnerLinks>
2 <PartnerLink name="ProcessBPL">
3 <PartnerRole>
4 <BindingInformation>
5 <soap:binding style="document"
6     transport="http://schemas.xmlsoap.org/soap/http"/>
7 <operation name="start">
8 <soap:operation style="document"/>
9 <wSDL:input>
10 <soap:body use="literal"/>
11 </wSDL:input>
12 </operation>
13 </BindingInformation>
14 <ServiceInformation>
15 <wsa:EndpointReference
16     xmlns:w="http://iaas.perfTest.org/wSDL/ProcessB/">
17 <wsa:Address>http://localhost:9080/ProcessB/
18     ProcessBPTProcessBPTBindingPort</wsa:Address>
19 <wsa:PortType>ProcessBPT</wsa:PortType>
20 <wsa:ServiceName PortName="ProcessBPTBindingPort">w:ProcessB</
21     wsa:ServiceName>
```

```

20         </wsa:EndpointReference>
21     </ServiceInformation>
22 </PartnerRole>
23 </PartnerLink>
24 </PartnerLinks>

```

Listing A.19: Partner Link Definitions

Each partner link is identified via the `PartnerLink` element as shown in Line 2. A partner link is usually associated with two roles, the role that the process plays and the role that the partner plays. One can specify information for both role types; however usually only the specification for the partner role is needed which is identified via the `PartnerRole` element shown in Line 3. The information provided follows the structure used in WSDL: the binding information and the service information. The binding information is identified via the `BindingInformation` element (Line 4). It should be noted that the binding in the example is provided for completeness only; it is not needed as all settings are the default ones. More important is the service information, identified via the `ServiceInformation` element in Line 14, which, in contrast to WSDL, is provided as an endpoint reference using WS-Addressing [W3C06]. The SWSM uses the partner link information to carry out the appropriate SOAP/HTTP request.

The second part, the flow configuration part, provides flow configuration options as described in Chapter 7. Listing A.20 shows only a subset of the information to limit the amount of space needed; for example, configuration settings are only provided for one variable instead of providing them for all variables.

```

1     <executionOptions>
2         <variableOptions>
3             <variable name="inRequest">
4                 <length>200</length>
5             </variable>
6         </variableOptions>
7
8     <activityOptions>
9         <activity name="H">
10            <invocationMode>SYNCHRONOUS_INLINE</invocationMode>
11        </activity>
12        <activity name="I">

```

```

12         <invocationMode>ASYNCHRONOUS_INLINE</invocationMode>
13     </activity>
14 </activityOptions>

15     <correlationSetOptions>
16         <correlationSet name="correlation1">
17             <hashAlgorithm>STRING_SHORT</hashAlgorithm>
18             <hashUnique>YES</hashUnique>
19         </correlationSet>
20     </correlationSetOptions>
21 </executionOptions>

```

Listing A.20: Flow Configuration Options

The `variableOptions` (Line 2) defines the length of the individual variables in the process, so that the SWoM can use the variable table structure that provides the best performance. The appropriate approach is described in Section 7.5.

The `activityOptions` (Line 7) defines the processing that the SWoM should carry out when a particular activity is executed. The `invocationMode` tells the SWoM how to process the invocation of the associated Web Service; a detailed description of this performance options can be found in Section 7.1.2.

The `correlationSetOptions` (Line 15) defines the length of the individual correlation sets, so that the SWoM can select the best hashing method for correlation sets. More about this performance improvement approach can be found in Section 7.6.

#### A.2.4. Flow Execution Plan

The flow execution plan is constructed by the flow optimizer and stored in the buildtime database. The flow execution plan shown is based on the transaction flow type `ULTIMATE` using standard database storage. It provides the following optimization techniques: intra transaction caching, optimized variable access, XPath processing optimization, direct invocation, and variable/correlation length optimization.

Listing A.21 shows the header of the flow execution plan. The `processModel` identifies the process model with which the flow execution plan is associated with; this information is needed when a flow execution plan is imported that has been generated by some other means than by the flow optimizer. The

planIdentifier allows to maintain different plans for the same process model (this information is displayed in the SWoM administration console). The namespace of the flow execution plan is `http://shared.swom.iaas/fep/`.

```
1      <?xml version="1.0" encoding="UTF-8"?>
2      <tns:FlowExecutionPlan processModel="ProcessA"
           planIdentifier="FEP1"
3          xmlns:tns=
           "http://shared.swom.iaas/fep/"
4          xmlns:wSDL=
           "http://schemas.xmlsoap.org/wSDL/"
5          xmlns:soap=
           "http://schemas.xmlsoap.org/wSDL/soap/"
6          xmlns:wsa=
           "http://schemas.xmlsoap.org/ws/2004/08/addressing"
7          xmlns:xsi=
           "http://www.w3.org/2001/XMLSchema-instance">
```

Listing A.21: Flow Execution Plan Header

The flow execution plan, as presented in Section 8.2, consists of two major parts, the definition of the transaction flow and the definition of the execution options. Listing A.22 shows the start of the transaction flow, which is identified via the `transactionFlow` element (Line 1). The `links` section (Line 2) defines the links that are connecting the different transactions in the transaction flow; the `multiPhase` links connect activities that are processed in two transactions, such as receive activities, the `join` links connects multi-transaction join activities with the transaction that contains the target of the join activity link.

```
1      <transactionFlow>
2          <links>
3              <link name="MPT1T2" type="multiPhase"/>
4              <link name="MPT1T3" type="multiPhase"/>
5              <link name="JLG4" type="join" activity="G"/>
6          </links>
```

Listing A.22: Transaction Links

Listing A.23 shows the first transaction within the transaction flow. Each transaction within the transaction flow is given a unique name identified by the `ID` attribute (Line 2). The definition of the transaction starts with the basic execution properties of the transaction. Line 4 through Line 10 control the usage of the intra transaction cache.

```

1      <transactions>
2          <transaction ID="T1">
3              <executionProperties>
4                  <getFromIntraTransactionCache>NO
5                      </getFromIntraTransactionCache>
6                  <storeInIntraTransactionCache>YES
7                      </storeInIntraTransactionCache>
8                  <associatedCache>0
9                      </associatedCache>
10                 <cacheDisposable>NO
11                     </cacheDisposable>
12                 <parallelTransaction>NO
13                     </parallelTransaction>
14                 <executionMode>UNCONDITIONAL
15                     </executionMode>
16             </executionProperties>

```

Listing A.23: Transaction Execution Properties

Line 12 indicates that no transactions are executed parallel to this one. This information is not used in the current transaction, since the transaction is the first one in the transaction flow, however it used for all subsequent transactions when specified. The navigator can, if no parallel transactions are carried out, use a simple non-locking SQL call to obtain the process instance root. If parallel transactions are carried out, the navigator generally needs to use a locking SQL call so that the same piece of the process instance is not processed in parallel (with unpredictable results).

Line 14 tells the navigator that no transition conditions are attached to the links between the activities that make up the transaction. The navigator uses this information to optimize the navigation of the activities, see Section 8.4.3 for more information.

Listing A.24 specifies the links which leave the current transaction. As can be seen, two links are leaving the transaction, resulting in the execution of two parallel transactions.

```

1      <sources>
2          <source linkName="MPT1T2"/>
3          <source linkName="MPT1T3"/>
4      </sources>

```

Listing A.24: Source of Parallel Transactions

The flow execution plan provides for each of the different pieces of a process model that are referenced in a transaction appropriate information that helps optimize their processing; Listing A.25 shows the information that is provided for each of the activities within the transaction.

```
1    <activitiesInTransaction>
2      <activityInTransaction name="A">
3        <position>START</position>
4        < caching>NO</ caching>
5        < persist>NO</ persist>
6      </activityInTransaction>
7      <activityInTransaction name="B">
8        <position>MIDDLE</position>
9        < caching>NO</ caching>
10       < persist>NO</ persist>
11     </activityInTransaction>
12     <activityInTransaction name="C">
13       <position>MIDDLE</position>
14       < caching>NO</ caching>
15       < persist>NO</ persist>
16     </activityInTransaction>
17     <activityInTransaction name="D">
18       <position>MIDDLE</position>
19       < caching>NO</ caching>
20       < persist>NO</ persist>
21     </activityInTransaction>
22     <activityInTransaction name="E">
23       <position>MIDDLE</position>
24       < caching>YES</ caching>
25       < persist>NO</ persist>
26     </activityInTransaction>
27     <activityInTransaction name="F">
28       <position>END</position>
29       < caching>YES</ caching>
30       < persist>NO</ persist>
31     </activityInTransaction>
32   </activitiesInTransaction>
```

Listing A.25: Activities

The activities are carried out in the sequence that is specified. Line 2 shows the appropriate definition for the first activity in the transaction, the initial receive activity. The position in Line 3 specifies that this activity is the first one in the transaction. Note that this information in fact is superfluous, however it maintained for completeness of information (well, also for debugging). Line 4

defines via the caching element, whether the activity should be cached, that means maintained in the intra transaction cache. Finally, the persist element as used in Line 5 specifies whether the activity should be persisted or not.

Listing A.26 shows the appropriate definitions for variables. Each variable is defined via an appropriate variableInTransaction entry as shown in Line 2.

```
1      <variablesInTransaction>
2          <variableInTransaction name="inRequest">
3              <startState>NEW</startState>
4              <creationActivity>A</creationActivity>
5              <endState>DELETE</endState>
6              <caching>NO</caching>
7              <persist>NO</persist>
8          </variableInTransaction>
9          <variableInTransaction name="outRequest1">
10             <startState>NEW</startState>
11             <creationActivity>B</creationActivity>
12             <endState>DELETE</endState>
13             <caching>NO</caching>
14         </variableInTransaction>
15         <variableInTransaction name="outRequest2">
16             <startState>NEW</startState>
17             <creationActivity>B</creationActivity>
18             <endState>DELETE</endState>
19             <caching>NO</caching>
20         </variableInTransaction>
21     </variablesInTransaction>
```

Listing A.26: Variables

Line 3 defines via the startState that the variable is created in the transaction, so that there is no need for the instance cache manager to check whether the variable exists; that means no SQL call needs to be carried out. The creationActivity in Line 4 specifies that the variable is created by activity A. Incidentally this information is not directly used by the navigator, but is here for completeness of information. Line 5 specifies via the endState that the variable does not survive the life of the transaction. In fact, the inRequest variable is a temporary variable, something which has been proposed in [IBM04], as it is created and deleted in the transaction. Finally, Line 6 and Line 7 specify that the variable should not be cached in the intra transaction cache, nor should it be persisted, which is obvious, since the variable is kind of temporary variable. It should be noted that the endState information is used for delete processing

when ongoing deletion is active.

Finally, information about the correlation sets in the transaction is provided via the `correlationSetInTransaction` element within the `correlationSetsInTransaction` as shown in Listing A.27.

```
1 <correlationSetsInTransaction>
2   <correlationSetInTransaction name="correlation1">
3     <startState>NEW</startState>
4     <creationActivity>C</creationActivity>
5     <endState>KEEP</endState>
6     <storeInCorrelationCache>YES
7       </storeInCorrelationCache>
8     <deleteFromCorrelationCache>NO
9       </deleteFromCorrelationCache>
10    <getFromCorrelationCache>NO
11      </getFromCorrelationCache>
12    <persist>YES</persist>
13    <caching>YES</caching>
14  </correlationSetInTransaction>
15  <correlationSetInTransaction name="correlation2">
16    <startState>NEW</startState>
17    <creationActivity>D</creationActivity>
18    <endState>KEEP</endState>
19    <storeInCorrelationCache>YES
20      </storeInCorrelationCache>
21    <deleteFromCorrelationCache>NO
22      </deleteFromCorrelationCache>
23    <getFromCorrelationCache>NO
24      </getFromCorrelationCache>
25    <persist>YES</persist>
26    <caching>YES</caching>
27  </correlationSetInTransaction>
28 </correlationSetsInTransaction>
29 </transaction>
```

Listing A.27: Correlation Sets

The definitions in Line 3 through Line 5 specify the life cycle of the correlation set within the transaction: The `startState` element specifies that the correlation set is created in the transaction, the `creationActivity` that activity D creates the correlation, and `endState` indicates that the correlation set is to be kept. Line 6 through Line 10 control the handling of the correlation set with respect to the correlation cache. Line 12 defines that the correlation set needs to be persisted in the database; Line 13 indicates that the correlation set should be

moved to the correlation cache.

Listing A.28 shows the information that applies to the second transaction. Note that the third transaction is not shown; it is just a mirror of this one.

```
1 <transaction ID="T2">
2   <executionProperties>
3     <getFromIntraTransactionCache>YES
4     </getFromIntraTransactionCache>
5     <storeInIntraTransactionCache>YES
6     </storeInIntraTransactionCache>
7     <parallelTransaction>YES</parallelTransaction>
8     <executionMode>UNCONDITIONAL_JOIN_END
9     </executionMode>
10    <rootUpdateRequired>NO</rootUpdateRequired>
11    <joinTransaction>YES</joinable>
12    <associatedCache>0</associatedCache>
13    <sourceCache>0</sourceCache>
14    <cacheDisposable>NO</cacheDisposable>
15    <cacheImage>FULL</cacheImage>
16  </executionProperties>
17  <targets>
18    <target linkName="MPT1T2"/>
19  </targets>
20  <sources>
21    <source linkName="JLG"/>
22  </sources>
23  <activitiesInTransaction>
24    <activityInTransaction name="E">
25      <position>START</position>
26      < caching>NO</ caching>
27      < persist>NO</ persist>
28    </activityInTransaction>
29    <activityInTransaction name="G">
30      <position>END</position>
31      < caching>YES</ caching>
32      < persist>YES</ persist>
33    </activityInTransaction>
34  </activitiesInTransaction>
35  <variablesInTransaction>
36    <variableInTransaction name="inResponse1">
37      <startState>NEW</startState>
38      <creationActivity>E</creationActivity>
39      <endState>KEEP</endState>
40      < caching>YES</ caching>
41      < persist>YES</ persist>
42    </variableInTransaction>
43    <variableInTransaction name="inResponse2">
44      <startState>UNKNOWN</startState>
```

```

45         <endState>KEEP</endState>
46         <caching>YES</caching>
47         <persist>NO</persist>
48     </variableInTransaction>
49     <variableInTransaction name="inInvoke">
50         <startState>NEW</startState>
51         <creationActivity>G</creationActivity>
52         <endState>KEEP</endState>
53         <caching>YES</caching>
54         <persist>NO</persist>
55     </variableInTransaction>
56 </variablesInTransaction>
57 <correlationSetsInTransaction>
58     <correlationSetInTransaction name="correlation1">
59         <endState>DELETE</endState>
60         <storeInCorrelationCache>NO
61             </storeInCorrelationCache>
62         <deleteFromCorrelationCache>YES
63             </deleteFromCorrelationCache>
64         <getFromCorrelationCache>YES
65             </getFromCorrelationCache>
66         <persist>NO</persist>
67         <caching>NO</caching>
68     </correlationSetInTransaction>
69 </correlationSetsInTransaction>
70 </transaction>

```

Listing A.28: Second Transaction

The only interesting part are the execution properties, that control the transaction, particularly in light of the fact, that a second transaction is running in parallel.

Line 3 and Line 5 control the usage of the intra transaction cache. Since single cache processing is used, the process instance is obtained from the intra transaction cache and replaced at the end.

Line 7 specifies not only that the activities should be carried out sequentially, as there are no transition conditions associated with the links but also specifies that the last activity is a join activity. In this case, processing of the outgoing links is not carried out until all other transactions that are having the same join activity are processed.

Line 8 is used to control the processing when the last join transaction has been processed. When set to NO, the last join transaction completes as normal.

However, when set to YES, the current transaction is not completed, but the outgoing link is followed and the target transaction is processed as part of the current transaction (see the ULTIMATE transaction flow type description in Section 6.1).

Line 12 through 15 control the details of the intra transaction cache handling. Line 12 defines the cache ID that is the cache ID used for the transaction; Line 13 identifies the cache that is used as the source for establishing the transaction cache from the intra transaction cache; Line 14 indicates what to do with the cache at the end of the transaction; and Line 15 tells the instance cache manager, whether the cache contains all information or not. A detailed description all of these parameters and their effects are found in Section 8.9.

Listing A.29 shows the fourth, the last transaction. This is indicated by, as seen in Line 6, by setting the lastTransaction indicator to YES.

```
1      <transaction ID="T4">
2          <executionProperties>
3              <storeInIntraTransactionCache>NO</storeInIntraTransactionCache>
4              <parallelTransaction>NO</parallelTransaction>
5              <executionMode>UNCONDITIONAL</executionMode>
6              <lastTransaction>YES</lastTransaction>
7          </executionProperties>
8          <targets>
9              <target linkName="JLG"/>
10         </targets>
11         <activitiesInTransaction>
12             <activityInTransaction name="H">
13                 <position>START</position>
14                 <caching>NO</caching>
15                 <persist>NO</persist>
16             </activityInTransaction>
17             <activityInTransaction name="I">
18                 <position>END</position>
19                 <caching>NO</caching>
20                 <persist>NO</persist>
21             </activityInTransaction>
22         </activitiesInTransaction>
23         <variablesInTransaction>
24             <variableInTransaction name="inInvoke">
25                 <endState>DELETE</endState>
26             </variableInTransaction>
27             <variableInTransaction name="outInvoke">
28                 <startState>NEW</startState>
29                 <creationActivity>H</creationActivity>
```

```

30         <endState>DELETE</endState>
31     </variableInTransaction>
32 </variablesInTransaction>
33 </transaction>
34 </transactions>
35 </transactionFlow>

```

Listing A.29: Last Transaction

Listing A.30 starts the second part of the flow execution plan, which contains information/settings that apply to the whole process model. The first section within that part is identified via the `baseOptions`, which as the name suggests, contains the basic options that the flow optimizer has determined.

```

1  <executionOptions>
2    <baseOptions>
3      <transactionType>ULTIMATE</transactionType>
4      <maxParallelTransactions>2
5      </maxParallelTransactions>
6      <persistenceMode>DATABASE</persistenceMode>
7      <cacheMode>SINGLE</cacheMode>
8      <correlationCacheSupport>YES
9      </correlationCacheSupport>
10     <intraTransactionCacheSupport>YES
11     </intraTransactionCacheSupport>
12 </baseOptions>

```

Listing A.30: Base Options

Line 3 defines the transaction flow type to be `ULTIMATE`, indicating that the flow optimizer has come up with this as being the best one (or the one requested by the process modeler via an appropriate entry in the process deployment descriptor). Line 4 specifies the number of parallel transactions to be two; this information is used by the navigator to use the correct SQL call when fetching the process instance root. Line 6 defines that the process instance is persisted in the database using the standard mechanism of storing each object in a separate table. Line 7 defines that only one copy of the transaction cache is used in the intra transaction cache. Line 8 requests the usage of the correlation cache, Line 10 the usage of the intra transaction cache.

Listing A.31 shows the definitions for the activities. In particular, it defines the invocation mode, that means it defines whether the service invoker is called

directly or via message. The information is generally copied from the process deployment descriptor.

```
1 <activities>
2   <activity name="C">
3     <invocationMode>ASYNCHRONOUS_EXTERN
4     </invocationMode>
5   </activity>
6   <activity name="D">
7     <invocationMode>ASYNCHRONOUS_EXTERN
8     </invocationMode>
9   </activity>
10  <activity name="H">
11    <invocationMode>SYNCHRONOUS_INLINE
12    </invocationMode>
13  </activity>
14  <activity name="I">
15    <invocationMode>ASYNCHRONOUS_INLINE
16    </invocationMode>
17  </activity>
18 </activities>
```

Listing A.31: Activities

Next, the properties of the variables are defined as shown in Listing A.32.

```
1 <variables>
2   <variable name="InRequest">
3     <length>100</length>
4   </variable>
5   <variable name="OutRequest1">
6     <length>100</length>
7   </variable>
8   <variable name="OutRequest2">
9     <length>100</length>
10  </variable>
11  <variable name="InResponse1">
12    <length>100</length>
13  </variable>
14  <variable name="InResponse2">
15    <length>100</length>
16  </variable>
17  <variable name="InInvoke">
18    <length>100</length>
19  </variable>
20  <variable name="OutInvoke">
21    <length>100</length>
22  </variable>
23 </variables>
```

## Listing A.32: Variables

The FEP finishes off with the appropriate definitions for the correlation sets as shown in Listing A.33.

```
1    <correlationSets>
2        <correlationSet name="correlation1">
3            <hashAlgorithm>STRING_SHORT</hashAlgorithm>
4            <hashUnique>YES</hashUnique>
5        </correlationSet>
6        <correlationSet name="correlation2">
7            <hashAlgorithm>STRING_SHORT</hashAlgorithm>
8            <hashUnique>YES</hashUnique>
9        </correlationSet>
10    </correlationSets>
11    </executionOptions>
12 </tns:FlowExecutionPlan>
```

## Listing A.33: Correlation Sets

Of interest are the two definitions for the correlation sets: Line 3 defines the hash algorithm to be used, such as `STRING_SHORT` which indicates that a String should be used, and Line 4 which defines whether the hash algorithm is unique (if so, the number of SQL calls is minimal). Further information can be found in Section 7.6.

## A.3. Called Processes

All called processes are rather simple. Listing A.34 shows, for example, the WS-BPEL definition of Process B and Process C, respectively. Note that the algebraic notation of WS-BPEL is used and all non-execution relevant information is left off.

```
1    <process name="ProcessB">
2
3        <partnerLinks>
4            <partnerLink name="ProcessBPL"
5                myRole="ProcessB"
6                partnerRole="CallbackForProcessB"
7                partnerLinkType="ProcessBLT"/>
8        </partnerLinks>
9
10       <variables>
```

```

9         <variable messageType="shortMessage" name="request" />
10    </variables>

11    <sequence>

12        <receive createInstance="yes"
13            name="receive"
14            operation="start"
15            partnerLink="ProcessBPL"
16            portType="ProcessBPT"
17            variable="request">
18    </receive>

19        <invoke name="invoke"
20            operation="receiveResponseFromProcessB"
21            partnerLink="ProcessBPL"
22            portType="ProcessAPT"
23            inputVariable="request">
24    </invoke>
25    </sequence>
26 </process>

```

Listing A.34: BPEL Process Definition for Process B

The processes just receive a message with a single field and return this message to the calling Process A via the invoke activity. The main purpose of the two processes is to test correlation processing: a correlation set is defined on the field in the message in Process A.

Listing A.35 shows the definition of Process D.

```

1    <process name="ProcessD">

2        <partnerLinks>
3            <partnerLink name="ProcessDPPL"
4                myRole="ProcessD"
5                partnerLinkType="ProcessDLT"/>
6    </partnerLinks>

7    <variables>
8        <variable messageType="shortMessage"
9            name="request" />
10    </variables>

11    <sequence>

12        <receive createInstance="yes"
13            name="receive"

```

```

14         operation="start"
15         partnerLink="ProcessDPL"
16         portType="ProcessDPT"
17         variable="request">
18     </receive>

19     <reply name="reply"
20         operation="start"
21         partnerLink="ProcessDPL"
22         portType="ProcessDPT"
23         variable="request">
24     </reply>

25 </sequence>
26 </process>

```

Listing A.35: BPEL Process Definition for Process D

The process implements a synchronous Web Service, that receives a field in a message and just returns the message to the calling Process A. Its main purpose is to evaluate the performance of synchronous invocation.

Listing A.36 shows the definition of Process E.

```

1     <process name="ProcessE" >

2     <partnerLinks>
3         <partnerLink name="ProcessEPL"
4             myRole="ProcessE"
5             partnerLinkType=ProcessELT"/>
6     </partnerLinks>

7     <variables>
8         <variable messageType="shortMessage" name="request"/>
9     </variables>

10    <sequence>

11        <receive createInstance="yes"
12            name="receive"
13            operation="start"
14            partnerLink="ProcessEPL"
15            portType="ProcessEPT"
16            variable="request">
17    </receive>

18    </sequence>
19 </process>

```

### Listing A.36: BPEL Process Definition for Process E

The process is a simple fire-and-forget process that just processes a received message.

The content of the associated process deployment descriptors and flow execution plans is obvious. The only interesting definition is the one exemplarily shown for Process E in Listing A.37: the definition of the process as a microflow.

```
1      <DeploymentDescriptor processModel="ProcessE">
2          <executionOptions>
3              <baseOptions>
4                  <microFlowMode>YES</microFlowMode>
5              </baseOptions>
6          </executionOptions>
7      </DeploymentDescriptor>
```

### Listing A.37: Microflow Definition of Called Processes



# IMPLEMENTATION

This appendix presents the implementation of the *SWoM*. It discusses, in particular, the infrastructure in which the *SWoM* is executing, the project structure of the *SWoM*, and the environment that is used for developing the *SWoM*.

## B.1. Implementation Scope

The development of a full-fledged workflow management system is a rather large undertaking. The reason for this significant development effort is very simple: in addition to the actual core components, navigator and service invoker, that actually implement the WS-BPEL, additional components, such as administration or non-standard functions, such as auditing, are required to make such a system usable in a customer environment.

It is obvious that it is impossible to have all functions prescribed by the WS-BPEL standard implemented in the *SWoM*. Only those functions have been implemented that help to verify that the goal of the thesis, namely the architecture of a high-performance Workflow Management System, has been achieved. The most important activities have been implemented, such as flow, sequence,

receive, assign, reply and invoke as well as correlation processing. All other activities, including scopes, have been left out. Furthermore no compensation and event handlers have been implemented. Still, the development of the SWoM comprises a significant development effort, as evidenced by the number of modules and lines of code as listed in Section B.3. It should be noted that the architecture allows the implementation of these WS-BPEL functions without jeopardizing the stability and robustness of the implementation.

## B.2. Infrastructure Exploitation

The SWoM is implemented as a standard JEE application. The appropriate JEE functionality is delivered by the application server component of IBM WebSphere Version 7.0. Any data that needs to be made persistent is stored in IBM DB2 Version 9.5.

All components of the SWoM are implemented as EJBs, which provides for the necessary transactional properties. The SWoM uses the EJB Version 3 capabilities of dependency injection to provide for the efficient interaction between the different components. This provides, for example, the navigator, implemented as a stateless session EJB with the capability to simply call the statistics manager, also implemented as a stateless session EJB.

The core components, such as the navigator or the service invoker, come in two flavors: as a stateless session bean to be called via the appropriate Java invocation mechanism, and as message-driven bean, that is started by messages in the associated queues. The actual navigator and service invoker functions are encapsulated in appropriate classes that are loaded by the beans.

### B.2.1. IBM WebSphere Setup

This section shows the most important definitions in IBM WebSphere that are needed for the SWoM.

Figure B.1 shows the definitions for the databases. Each database is defined as a data source to IBM WebSphere.

The connection pool size specifies the number of connections that can be

Database	Usage	Connection Pool Size	Statement Cache Size
SWOMAU	Audit database	50	5
SWOMBT	Buildtime database	5	20
SWOMRT	Runtime database	50	100
SWOMSY	System database	5	5
SWOMSR	Scheduler database	2	5
SWOMST	Statistics database	5	10

Figure B.1.: Data Sources

established as a maximum. Only the runtime and the audit database need a significant amount of connections; all other ones are only rarely accessed.

The statement cache size defines the maximum number of statements held in the cache. The figure reflects the amount of different statements that are being carried. Note that the size for the buildtime database has been set rather low, since the buildtime database is rarely used during process execution.

Figure B.2 shows the definitions for the queues that the SWoM needs.

Queues	Processing component	Inserting components
NavigationQueue/ NavigationNPQueue	Navigator	Navigator Service Invoker Scheduler Administration
InvocationQueue/ InvocationNPQueue	Service Invoker	Navigator
DeleteServerQueue	Delete Server	Navigator
DeadLetterQueue	Administration	WebSphere

Figure B.2.: Queues

Two versions exist for the navigator and the service invoker queues, one supporting persistent, the other (indicated via the letters NP), non-persistent queues.

The dead letter queue is managed by IBM WebSphere and is used when IBM WebSphere can not, for whatever reason, deliver a message to one of the SWoM queues. The inserted message triggers the activation of an administrative component that analyzes the message, and if possible repairs it, and sends it to the proper queue again. If not possible, it is inserted into the system activity table in the system database (the same table in which any SWoM internal errors are written).

### B.3. Project Structure

The overall SWoM project has been divided into several smaller projects. Each of these projects implements a particular part of the SWoM.

#### B.3.1. Administration

The administration project contains all components that provide the administration functions for the SWoM. All components, with the exception of the buildtime component, are implemented as EJBs (stateless session beans).

Table B.1.: Administration Components

<b>Subcomponent</b>	<b>Functions</b>
Buildtime	Basic functions for managing the buildtime information. Includes the schema definitions for the flow execution plan, the process deployment descriptor and the system deployment descriptor

<b>Subcomponent</b>	<b>Functions</b>
Cache Manager	Manages the appropriate requests to the cache managers, such as query or reset
Data Management	Contains all functions to access the build-time and runtime database. Called by all other administration components.
Import/Export	Import/exports SWoM archives, which include for a particular process model the appropriate WS-BPEL definition, the related WSDL definitions and the associated process deployment descriptor.
System Optimizer	Uses information collected by the statistics manager to tune IBM WebSphere and IBM DB2.
Monitoring	Manages and displays the information generated by the audit manager.
Optimization	Flow Optimizer
Performance Monitor	Obtains IBM WebSphere performance indicators, such as the JVM load or connection pool usage.
Process Deployment	Deploys a process model by generating the appropriate façade beans and deploying them into WebSphere.

<b>Subcomponent</b>	<b>Functions</b>
Process Instance Management	Provides process instance management functions, such as terminate, suspend or resume.
Process Model Management	Provides process model management functions, such as lock or unlock process model.
Queue Management	Manages the queues that the SWoM maintains.
Statistics Management	Manages the appropriate requests to the statistics manager, such as query or reset.
Systems Management	Imports and exports the system deployment descriptor.
System Activity Monitoring	Manages the appropriate requests to the system activity monitor.
User Manager	Manages the users in the system, including password maintenance

### B.3.2. Administration Interface

This project implements the administration interface, which provides the front end for the functions provided by the administration component. Java Server Faces with Java Server Pages provides the underlying technology. In addition, the AJAX conformant SPRY framework from Adobe is used.

Table B.2.: Administration Interface Components

<b>Subcomponent</b>	<b>Functions</b>
Cache Manager	Manages the screens associated with the information provided by the cache manager
Check Credential Listener	Generates appropriate credentials when a user has signed on and checks these credentials when the user issues a new request
Performance Monitor	Manages the screens associated with the information provided by the performance monitor.
Navigation	Manages the navigation between the different administration screen groups
Process Instance Management	Manages the screens associated with process instance management.
Process Model Management	Manages the screens associated with process model management.
Queue Management	Manages the screens associated with queue management.
Optimizer	Manages the screens associated with the flow optimizer and the management of the flow execution plan.

<b>Subcomponent</b>	<b>Functions</b>
Systems Management	Manages the screens associated with systems management.
System Activity Management	Manages the screens associated with system activity monitoring.
Web Resources	Contains all resources that are associated with the graphical end user interface, such as JSPs.

### B.3.3. Process Execution

This project contains all components that are used for executing process instances. Only the two components, Service Invoker and the Navigator, are externally exposed as EJBs. All other components are loaded by the navigator and service invoker instance when needed. For example, each navigator instance creates an instance in the instance cache for its particular use.

The exchange of information between the various components is by means of process execution context class, that maintains references to all service components that are packaged into the Shared EJB project as well as the information about the current process instance that is processed.

Table B.3.: Process Execution Components

<b>Subcomponent</b>	<b>Functions</b>
Audit Manager	Manages the audit trail, an optional feature of the SWoM.
Correlation Manager	Determines correlation information on outgoing messages as well as incoming messages.

<b>Subcomponent</b>	<b>Functions</b>
Compilation Unit Cache	Manages the compilation units that have either been provided by the user or generated by the flow optimizer.
Instance Cache	Instance Cache Manager
Instance Memory Cache	A version of the instance cache manager that supports the execution of process instances in memory only execution.
Service Invoker	Service Invocation Manager
Navigator	Navigator
Shared	Contains functions that are shared by the different process execution components, for example, the message handler, which puts messages into the various SWoM queues.

#### B.3.4. Shared

This project contains a set of components that are shared by SWoM components. Most classes are of definitional type; the XPath processor is the only class that performs any functions.

Table B.4.: Shared Components

<b>Subcomponent</b>	<b>Functions</b>
---------------------	------------------

<b>Subcomponent</b>	<b>Functions</b>
Messages	Definition of the structure of messages that are exchanged between the different components
XPath processor	Evaluates XPath expressions for the different components.
Properties	Manages properties in property files such as the text for messages
Types	Defines the values for all properties that are used in the SWoM.

### B.3.5. Shared EJBs

This project contains those components that are exploited by other SWoM components. They are exposed as EJBs.

Table B.5.: Shared EJBs Components

<b>Subcomponent</b>	<b>Functions</b>
Correlation Cache Manager	Manages the correlation cache
Intra Transaction Cache Manager	Manages the intra transaction cache
Model Cache Manager	Manages the process models
SSDD Cache Manager	Manages the system deployment descriptor

<b>Subcomponent</b>	<b>Functions</b>
System Statistics Manager	Obtains and manages SWoM related system-relevant statistics.
System Activity Monitor	Records and manages errors that are encountered by the SWoM. The collected information can be analyzed by system administrators so that corrective actions can be taken.
Trace Mode Manager	Is used by the generated façade beans to determine which traces should be written
UUID Generator	Generates UUIDs for usage within the SWoM.
Delete Server	Deletes process instances
Timer Manager	Provides timer services for the statistics manager
WSDL Cache Manager	Provides efficient access to the WSDLs.
Audit Manager	Provides cached and non-transacted handling of auditing.

### B.3.6. Statistics

The following table shows, for each project, the number of lines of code, classes, methods, and packages. The values have been determined using the *Metrics* 2 plugin for Eclipse<sup>1</sup>.

---

<sup>1</sup><http://metrics2.sourceforge.net/>

Table B.6.: Implementation Statistics

<b>Project</b>	<b>LoC</b>	<b>Classes</b>	<b>Methods</b>	<b>Packages</b>
Administration	28189	142	1619	23
Administration Interface	3155 (Java) 4322 (JSP)	16	333	1
Process Execution	27187	102	1200	16
Shared	8919	207	991	16
Shared EJBs	21987	230	2300	23
Installer	10175	73	548	5
<b>Total</b>	<b>103925</b>	<b>770</b>	<b>6991</b>	<b>82</b>

#### B.4. Development Environment

IBM Rational Application Developer V 8.0 [IBM11c] has been used as the development environment for the SWoM. It is the obvious choice as it is perfectly integrated into the IBM WebSphere environment in which the SWoM is running.

# BIBLIOGRAPHY

- [A<sup>+</sup>85] Anon, et al. A measure of transaction processing power. *Datamation*, 1985.
- [AAD<sup>+</sup>07a] A. Agrawal, M. Amend, M. Das, M. Ford, C. Keller, M. Kloppmann, D. König, F. Leymann, R. Müller, G. Pfau, et al. Web Services Human Task (WS-HumanTask), Version 1.0. [http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel4people/WS-HumanTask\\_v1.pdf](http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel4people/WS-HumanTask_v1.pdf), 2007.
- [AAD<sup>+</sup>07b] A. Agrawal, M. Amend, M. Das, M. Ford, C. Keller, M. Kloppmann, D. König, F. Leymann, R. Müller, G. Pfau, et al. WS-BPEL Extension for People (BPEL4People), Version 1.0. [http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel4people/BPEL4People\\_v1.pdf](http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel4people/BPEL4People_v1.pdf), 2007.
- [Act09] Active Endpoints Inc. ActiveVOS <sup>™</sup> Server Architecture. [http://www.activevos.com/indepth/f\\_technicalNotes/activeVOSArchitecture/ActiveBPELArchitecture](http://www.activevos.com/indepth/f_technicalNotes/activeVOSArchitecture/ActiveBPELArchitecture), 2009.
- [Act11] Active Endpoints Inc. ActiveVOS - Performance Tuning. <http://www.activevos.com/cp/640/activevos-performance-tuning>, 2011.

- [AKL<sup>+</sup>08] A. Arning, M. Kloppmann, F. Leymann, G. Pfau, D. Roller, A. Schmitz, F. Schwenkreis, C. Zentner. Dynamic Determination of Transaction Boundaries in Workflow Systems. <http://patft.uspto.gov/netahtml/PTO/srchnum.htm>, 2008. United States Patent US 7,386,577.
- [All01] C. Allinson. Information Systems Audit Trails in Legal Proceedings. *Computers & Security*, 20(5):409–421, 2001.
- [Apa11] Apache Software Foundation. Apache ODE. <http://ode.apache.org/>, 2011.
- [Apa12] Apache Software Foundation. Apache ODE - Architectural Overview. <http://ode.apache.org/developerguide/architectural-overview.html>, 2012.
- [Bas06] The Base16, Base32, and Base64 Data Encodings. <http://tools.ietf.org/html/rfc4648>, 2006.
- [BBC<sup>+</sup>07] A. Berglund, S. Boag, D. Chamberlin, M. Fernandez, M. Kay, J. Robie, J. Simeon. XML Path Language (XPath) Version 2.0. <http://www.w3.org/TR/xpath20/>, 2007.
- [BBD10a] D. Bianculli, W. Binder, M. Drago. Automated performance assessment for service-oriented middleware: a case study on BPEL engines. In *Proc. 19th International Conference on World Wide Web (WWW 2010)*, Raleigh, NC, USA. 2010.
- [BBD10b] D. Bianculli, W. Binder, M. Drago. SOABench: Performance Evaluation of Service-Oriented Middleware Made Easy. In *Proc. 32th International Conference on Software Engineering (ICSE)*, Cape Town, South Africa. 2010.
- [BF07] T. Baeyens, M. V. Faura. The Process Virtual Machine. <http://docs.jboss.com/jbpm/pvm/article/>, 2007.

- [BHL95] B. Blakely, H. Harris, R. Lewis. *Messaging and Queuing Using the MQI: Concepts & Analysis, Design & Development*. McGraw-Hill Inc., 1995.
- [Bon05] L. Bond. Understanding locking in DB2 Universal Database. <http://www.ibm.com/developerworks/data/library/techarticle/dm-0511bond/>, 2005.
- [BSR95] A. Bonner, A. Shrufi, S. Rozen. LabFlow-1: a Database Benchmark for High-Throughput Workflow Management. In *Proc. Fifth International Conference on Extending Database Technology (EDBT), (Avignon, France)*. 1995.
- [CARW97] A. Cichocki, H. A. Ansari, M. Rusinkiewicz, D. Woelk. *Workflow and Process Automation: Concepts and Technology*. Springer, 1997.
- [Cda99] J. Clark, S. deRose, et. al. XML Path Language (XPath) Version 1.0. <http://www.w3.org/TR/xpath>, 1999.
- [Cha98] S. Chaudhuri. An Overview of Query Optimization in Relational Systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington*, pp. 34–43. ACM Press, 1998.
- [Cha04] D. A. Chappel. *Enterprise Service Bus*. O'Reilly Media, 2004.
- [Che08] S. Chen. Improvement of Correlation Processing Efficiency in SWoM II. Student Thesis: University of Stuttgart, Institute of Architecture of Application Systems, 2008.
- [dK76] F. deRemer, H. Kron. Programming-in-the-Large Versus Programming-in-the-Small. In *Transactions on Software Engineering Vol SE-2, No. 2*. IEEE, 1976.
- [DKLW09] G. Decker, O. Kopp, F. Leymann, M. Weske. Interacting services: From specification to execution. *Data & Knowledge Engineering*, 68(10):946–972, 2009.

- [DP02] D. Davis, M. Parashar. Latency Performance of SOAP Implementations. In *CCGRID*, pp. 407–412. IEEE Computer Society, 2002.
- [Dug11] S. Duggirala. WebSphere Application Server Top 10 Tuning Recommendations. <http://wasdynacache.blogspot.de/2011/05/websphere-application-server-top-10.html>, 2011.
- [E. 99] E. C. M. A. International. ECMA-262: ECMAScript Language Specification. <http://www.ecmascript.org/>, 1999.
- [EN94] R. Elmasri, S. B. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings Publishing Company, Redwood City, California, 1994.
- [FL08] S. Faulhaber, K. Luttenberger. WebSphere Process Server operational architecture: Part 1: Base architecture and infrastructure components. [http://www.ibm.com/developerworks/websphere/library/techarticles/0809\\_faulhaber/0809\\_faulhaber.html](http://www.ibm.com/developerworks/websphere/library/techarticles/0809_faulhaber/0809_faulhaber.html), 2008.
- [FLF09] S. Faulhaber, K. Luttenberger, A. Faulhaber. Using WebSphere Process Server operational architecture to design your applications: Part 2: Implementation: SCA runtime, Business Process Choreographer, and supporting services. [http://www.ibm.com/developerworks/websphere/library/techarticles/0901\\_faulhaber/0901\\_faulhaber.html](http://www.ibm.com/developerworks/websphere/library/techarticles/0901_faulhaber/0901_faulhaber.html), 2009.
- [Gel85] D. Gelernter. Generative Communication in Linda. In *ACM Transactions on Programming Languages and Systems*, 7, 1985.
- [GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Publishing Company, 1995.
- [GHS95] D. Georgakopolus, M. Hornick, A. Sheth. An overview of workflow management: From process modeling to infrastructure for

automation. *Journal on Distributed and Parallel Database Systems*, 3(2):119–153, 1995.

- [GMW00] M. Gillmann, R. Mindermann, G. Weikum. Benchmarking and Configuration of Workflow Management Systems. In *Proc. 5th International Conference on Cooperative Information Systems (CoopIS)*, pp. 186–197. 2000.
- [HB11] D. Hare, C. Blythe. Case Study: Tuning WebSphere Application Server V7 and V8 for performance. [http://www.ibm.com/developerworks/websphere/techjournal/0909\\_blythe/0909\\_blythe.html](http://www.ibm.com/developerworks/websphere/techjournal/0909_blythe/0909_blythe.html), 2011.
- [HC94] M. Hammer, J. Champy. *Reengineering the Corporation*. Addison-Wesley Publishing Company, 1994.
- [HK04] R. Hauser, J. Koehler. Compiling Process Graphs into Executable Code. In *GPCE*, pp. 317–336. 2004.
- [IBM] IBM Corporation. IBM WebSphere Process Server V 6.0.2: Structure of the audit trail database view for business processes. <http://publib.boulder.ibm.com/infocenter/dmndhelp/v6r2mx/index.jsp?topic=/com.ibm.websphere.bpc.620.doc/doc/bpc/rg5attbl.html>.
- [IBM98] IBM Corporation, Armonk, New York. *IBM MQSeries Workflow: Concepts and Architecture*, 1998. Available through IBM branch offices.
- [IBM04] IBM Corporation. Temporary BPEL Variables. <http://ip.com/IPCOM/000030084>, 2004. IP Disclosure.
- [IBM05a] IBM Corporation. Extended Wait Activities. <http://ip.com/IPCOM/0000126252>, 2005. IP Disclosure.
- [IBM05b] IBM Corporation. Finish Activities. <http://ip.com/IPCOM/000126251>, 2005. IP Disclosure.

- [IBM05c] IBM Corporation. *IBM Process Server Tuning*, 2005.
- [IBM06] IBM Corporation, Armonk, New York. *DB2 Version 9 for Linux, UNIX, and Windows: Administration Guide: Planning*, 2006.
- [IBM10] IBM Corporation, Armonk, New York. *WebSphere Business Process Management (BPM) V7 Performance*, 2010.
- [IBM11a] IBM Corporation. CICS Family. <http://www-01.ibm.com/software/http/cics/>, 2011.
- [IBM11b] IBM Corporation. Correlation sets in BPEL processes. <http://www-01.ibm.com/support/docview.wss?uid=swg21171649>, 2011.
- [IBM11c] IBM Corporation. Rational Application Developer for WebSphere Software. <http://www-01.ibm.com/software/awdtools/developer/application/>, 2011.
- [IBM13] IBM Corporation. *IBM Process Server V 7.0 Information Center*, 2013.
- [Int11] Intalio. Intalio|BPMS. <http://www.intalio.com/bpms/server>, 2011.
- [JBo11] JBoss Community. jBPM. <http://www.jboss.org/jbpm>, 2011.
- [JBo12] JBoss Community. *Hibernate Getting Started Guide*, 2012.
- [Joh04] C. Johnson. IBM WebSphere Developer Technical Journal: WebSphere Enterprise Scheduler planning and administration guide. [http://www.ibm.com/developerworks/websphere/techjournal/0404\\_johnson/0404\\_johnson.html](http://www.ibm.com/developerworks/websphere/techjournal/0404_johnson/0404_johnson.html), 2004.
- [JSO13] Introducing JSON. <http://json.org>, 2013.
- [Kha08] R. Khalaf. *Supporting Business Process Fragmentation While Maintaining Operational Semantics: A BPEL Perspective*. Ph.D. thesis, University of Stuttgart, 2008.

- [Kim90] W. Kim. *Introduction to Object-Oriented Databases*. MIT Press, 1990.
- [KKS<sup>+</sup>06] D. Karastoyanova, R. Khalaf, R. Schroth, M. Paluszek, F. Leymann. BPEL Event Model. Technical Report Computer Science 2006/10, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, 2006. URL [http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTR/NCSTR\\_view.pl?id=TR-2006-10&engl=1](http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTR/NCSTR_view.pl?id=TR-2006-10&engl=1).
- [KL06] R. Khalaf, F. Leymann. Role-based Decomposition of Business Processes using BPEL. In *Proc. IEEE International Conference on Web Services (ICWS)*. 2006.
- [KLS<sup>+</sup>03] E. Kwan, S. Lightstone, B. Schiefer, A. Storm, L. Wu. Automatic Database Configuration for DB2 Universal Database: Compressing Years of Performance Expertise into Seconds of Execution. In *BTW 2003: Datenbanksysteme für Business, Technologie und Web, 10. BTW-Konferenz 26.-28. Februar 2003 in Leipzig*. Bonner Köllen Verlag, 2003.
- [KRL09] R. Khalaf, D. Roller, F. Leymann. Revisiting the Behavior of Fault and Compensation Handlers in WS-BPEL. In R. Meersman, T. Dillon, P. Herrero, editors, *On the Move to Meaningful Internet Systems: OTM 2009*, volume 5870 of *Lecture Notes in Computer Science*, pp. 286–303. Springer Berlin Heidelberg, 2009. doi: 10.1007/978-3-642-05148-7\_20.
- [L. 04] L. Pay. RUNSTATS in DB2 UDB Version 8.2: Guidelines and examples. <http://www.ibm.com/developerworks/data/library/techarticle/dm-0412pay/>, 2004.
- [LDA06] Lightweight Directory Access Protocol (LDAP): Technical Specification Road Map. <http://www.ietf.org/html/rfc4519>, 2006.
- [Ley97] F. Leymann. Transaktionsunterstützung für Workflows. *Informatik in Forschung & Entwicklung*, 12(1), 1997.

- [Ley01] F. Leymann. Managing Business Processes Via Workflow Technology, 2001. Tutorial at VLDB 2001, Rome, Italy.
- [Ley10] F. Leymann. Workflow Management. University Lecture, 2010.
- [LLOW91] C. Lamb, G. Landis, J. Orenstein, D. Weinreb. The ObjectStore Database System. In *Communications of the ACM* 34, pp. 50–63. 1991.
- [LR94] F. Leymann, D. Roller. Business Process Management with Flowmark. In *Proc. compcon 94 (San Francisco) , 28 August–3 September 1994*, pp. 230–234. IEEE, 1994.
- [LR97] F. Leymann, D. Roller. Workflow-based Applications. *IBM Systems Journal*, 36(1), 1997.
- [LR00a] F. Leymann, D. Roller. Method and Computer System for Generating Process Management Computer Programs from Process Models. <http://patft.uspto.gov/netahtml/PTO/srchnum.htm>, 2000. United States Patent US 6,011,917.
- [LR00b] F. Leymann, D. Roller. Method of Stratified Transaction Processing. <http://patft.uspto.gov/netahtml/PTO/srchnum.htm>, 2000. United States Patent US 6,012,091.
- [LR00c] F. Leymann, D. Roller. *Production Workflow: Concepts and Techniques*. Prentice-Hall, Upper Saddle River, New Jersey, 2000.
- [LR04] F. Leymann, D. Roller. Staging Objects in Workflow Management Systems. <http://patft.uspto.gov/netahtml/PTO/srchnum.htm>, 2004. United States Patent US 6,772,407.
- [LR05] F. Leymann, D. Roller. Context Based Execution Prioritization in Workflow-Management-Systems. <http://patft.uspto.gov/netahtml/PTO/srchnum.htm>, 2005. United States Patent US 6,976,257.

- [LR06] F. Leymann, D. Roller. Managing workload within workflow management systems. <http://patft.uspto.gov/netahtml/PTO/srchnum.htm>, 2006. United States Patent US 7,024,669.
- [LR10] F. Leymann, D. Roller. Resource Scheduling in Workflow Management Systems. <http://patft.uspto.gov/netahtml/PTO/srchnum.htm>, 2010. United States Patent US 7,74,786.
- [M. 07] M. Saraswatipura and S. Prasad. Understanding the advantages of DB2 9 autonomic computing features. <http://www.ibm.com/developerworks/data/library/techarticle/dm-0709saraswatipura/>, 2007.
- [Mar83] J. Martin. *Managing the Data Base Environment*. Prentice-Hall, 1983.
- [Mis07] S. K. Mishra. Web services tip: Use asynchronous beans to improve Web services performance. <http://www.ibm.com/developerworks/webservices/library/ws-tip-beans/index.html>, 2007.
- [MLR03] V. Markl, G. Lohmann, V. Raman. LEO: An autonomic query optimizer for DB2. *IBM Systems Journal*, 42(1), 2003.
- [MPGM08] T. Muehlfriedel, G. Pfau, J. Grundler, D. Meyer. WebSphere Process Server V6.1 - Business Process Choreographer: Performance Tuning Automatic Business Processes for Production Scenarios with DB2. <http://www-01.ibm.com/support/docview.wss?uid=swg27012639&aid=1>, 2008.
- [NIS02] NIST. FIPS 108-2 : Secure Hash Standard. <http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf>, 2002.
- [OAS07] OASIS. Web Services Business Process Execution Language (WS-BPEL) Version 2.0. <http://docs.oasis-open.org/wsbpel/2.0/wsbepl-v2.0.html>, 2007.

- [OAS09] OASIS. Web Services Atomic Transaction (WS-AtomicTransaction). <http://docs.oasis-open.org/ws-tx/wstx-wsat-1.2-spec-os/wstx-wsat-1.2-spec-os.html>, 2009.
- [Ope] Open SOA. Service Component Architecture Specifications. <http://www.osoa.org/display/Main/Service%2BComponent2BArchitecture2BSpecifications>.
- [ORA08] ORACLE Corporation, Redwood Shores. *Oracle Application Server Performance Guide 10g Release 3 : Chapter 7 Oracle BPEL Process Manager Performance Tuning*, 2008.
- [ORA11] ORACLE Corporation, Redwood Shores. *Oracle Fusion Middleware Performance and Tuning Guide 11g Release 1*, 2011.
- [ORA12] ORACLE. Java API for XML Processing. <http://jaxp.java.net/>, 2012.
- [Par08] D. Parmenter. *Key Performance Indicators (KPI): Developing, Implementing, and Using Winning KPIs*. John Wiley & Sons, Inc., 2008.
- [Pas13] PassMark Software. CPU Benchmarks. <http://www.passmark.com/index.html>, 2013.
- [Pau09] C. Pautasso. *Compiling Business Process Models into Executable Code*, chapter 15. IGI Global, 2009. URL <http://www.igi-global.com/reference/details.asp?ID=33287&v=tableOfContents>.
- [PPBB13] A. Peternier, C. Pautasso, W. Binder, D. Bonetta. High-performance execution of service compositions: a multicore-aware engine design. *Concurrency and Computation: Practice and Experience*, 2013. doi:10.1002/cpe.2948. URL <http://dx.doi.org/10.1002/cpe.2948>.

- [PWYT07] I. Popivanov, S. Walkty, A. Yang, B. Tang. Automatic statistics collection in DB2 for Linux, UNIX, and Windows. 2007.
- [QR04] W. Qiao, S. Rangaswamy. IBM WebSphere Developer Technical Journal: Writing PMI applications using the JMX interface. [http://www.ibm.com/developerworks/websphere/techjournal/0402\\_qiao/0402\\_qiao.html](http://www.ibm.com/developerworks/websphere/techjournal/0402_qiao/0402_qiao.html), 2004.
- [Rah10] E. Rahm. Datenbank-Benchmarks. University Lecture, 2010. URL <http://dbs.uni-leipzig.de/file/idbs1-ws10-kap8.pdf>.
- [RFT<sup>+</sup>05] F. Y. Ran, R. Fang, Z. Tian, H. Srinivasan, E. Lane, L. Hei, T. Banks. Message Oriented Middleware Cache Pattern – a Pattern in a SOA Environment. In *Fourth "Killer Examples" for Design Patterns and Objects First Workshop (OOPSLA 05)*. 2005.
- [RHAM06] N. Russell, A. H. M. ter Hofstede, W. M. P van der Aalst, N. Mulyar. Workflow Control-Flow Pattern: A Revised View. In *BPM Center Report BPM-06-22*. BPMcenter.org, 2006.
- [Ron92] Ronald Rivest. The MD5 Message-Digest Algorithm. <http://www.ietf.org/rfc/rfc1321.txt>, 1992.
- [SHLP05] M.-T. Schmidt, B. Hutchison, P. Lambros, R. Phippen. The Enterprise Service Bus: Making service-oriented architecture real. *IBM Systems Journal*, 44(4), 2005.
- [Sma12] SmartBear Software. soapUI. The Swiss-Army Knife of Testing. <http://soapui.com/About-SoapUI/what-is-soapui.html>, 2012.
- [SS04] S. Shastry, M. Saraswatipura. DB2 performance tuning using the DB2 Configuration Advisor. <http://www.ibm.com/developerworks/data/library/techarticle/dm-0605shastry/>, 2004.

- [Sun05] Sun Microsystems. JavaBusiness Integration (JBI) 1.0. <http://jcp.org/aboutJava/communityprocess/final/jsr208/index.html>, 2005.
- [SWSS03] C. Schuler, R. Weber, H. Schuldt, H. Schek. Peer-to-Peer Process Execution with OSIRIS. In *Proc. International Conference on Service-Oriented Computing (ICSOC)*. 2003.
- [SWSS04] C. Schuler, R. Weber, H. Schuldt, H. Schek. Scalable Peer-to-Peer Process Management - The OSIRIS Approach. In *Proc. International Conference on Web Services*. 2004.
- [The06] The Apache Software Foundation. *Apache Axis2 User's Guide*, 2006.
- [The11] The Apache Software Foundation. *Apache Derby - Getting Started with Derby*, 2011.
- [The12] The Apache Software Foundation. *Apache OpenJPA 2.2 User's Guide*, 2012.
- [Tra13] Transaction Processing Performance Council. <http://www.tpc.org>, 2013.
- [W3C] W3C. Web Services Definition Language (WSDL). <http://http://www.w3.org/TR/2006/CR-wsdl20-20060327>.
- [W3C05] W3C. Document Object Model (DOM). <http://www.w3.org/DOM/>, 2005.
- [W3C06] W3C. Web Services Addressing 1.0 - Core. <http://www.w3.org/TR/2006/REC-ws-addr-core-20060509>, 2006.
- [WRK<sup>+</sup>13] S. Wagner, D. Roller, O. Kopp, T. Unger, F. Leymann. Performance Optimizations for Interacting Business Processes. In *Proceedings of the first IEEE International Conference on Cloud Engineering (IC2E 2013)*. IEEE Computer Society, 2013.

- [Wut10] D. Wuttke. *Eine Infrastruktur für die dezentrale Ausführung von BPEL-Prozessen*. Ph.D. thesis, University of Stuttgart, 2010.
- [Xue10] J. Xue. Caching web services to improve the performance of business solutions in WebSphere Process Server. <http://www.ibm.com/developerworks/webservices/library/ws-caching/index.html>, 2010.
- [ZRL<sup>+</sup>04] D. Zilio, J. Rao, S. Lightstone, G. Lohmann, A. Storm, C. Garcia-Arellano, S. Fadden. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *Proceedings of the 30th VLBD Conference, Toronto, Canada*. 2004.

All links have been last followed on March 26, 2013.



# LIST OF FIGURES

1.1.	Two-Level Programming . . . . .	6
1.2.	Recursive Composition . . . . .	7
1.3.	Web Services Application Structure . . . . .	8
1.4.	Software Stack . . . . .	9
1.5.	Base Architecture . . . . .	11
1.6.	Calibration Performance Results . . . . .	16
1.7.	Performance Achievements for Benchmark Process . . . . .	17
2.1.	ODE Architecture . . . . .	29
2.2.	PVM Meta Model . . . . .	30
2.3.	ActiveVOS Architecture . . . . .	32
2.4.	WebSphere Process Server Architecture . . . . .	34
3.1.	Basic Architecture of SWoM . . . . .	39
3.2.	Transaction Boundaries . . . . .	44
3.3.	Hot Pooling . . . . .	45
3.4.	Process Instance Life Cycle . . . . .	47
3.5.	Activity Life Cycle . . . . .	49
3.6.	Basic Processing Within The SWoM . . . . .	50
3.7.	Basic Synchronous Processing Within The SWoM . . . . .	51

3.8.	Synchronous Invocation . . . . .	52
3.9.	Asynchronous Invocation . . . . .	53
3.10.	Import Processing . . . . .	59
3.11.	Basic Navigator Structure . . . . .	68
3.12.	Correlation Manager . . . . .	78
3.13.	Time Dependent Processing . . . . .	84
3.14.	OID Generation . . . . .	86
3.15.	Deployment of a Business Process . . . . .	88
3.16.	Scalability by CPU load . . . . .	89
3.17.	Scalability by Number of Parallel Requests . . . . .	90
4.1.	TPC-C Benchmark Improvements . . . . .	94
4.2.	Benchmark . . . . .	95
5.1.	Model Cache . . . . .	103
5.2.	Instance Cache Processing . . . . .	106
5.3.	Instance Cache Structure . . . . .	107
5.4.	System Deployment Descriptor Cache . . . . .	109
6.1.	Transaction Flow Types . . . . .	112
6.2.	Short Transaction Type . . . . .	114
6.3.	Medium Transaction Type . . . . .	115
6.4.	Long Transaction Type . . . . .	116
6.5.	Ultimate Transaction Type . . . . .	117
6.6.	Number of Transactions . . . . .	118
6.7.	Number of SQL Calls . . . . .	119
6.8.	Number Of Messages . . . . .	119
6.9.	Throughput . . . . .	121
6.10.	Transaction Flow Structure . . . . .	122
6.11.	Transaction Flow Physical Execution . . . . .	122
6.12.	Internal Queue . . . . .	124
7.1.	Inline Invocation for Synchronous Invocation . . . . .	132
7.2.	Inline Invocation for Asynchronous Invocation . . . . .	133

- 7.3. Transaction and SQL Calls Savings for Inline Invocation . . . . . 134
- 7.4. Microflow Processing . . . . . 135
- 7.5. Microflow Performance Improvements . . . . . 136
- 7.6. Correlation Cache . . . . . 137
- 7.7. Process Instance Deletion Server . . . . . 150
- 7.8. Compilation Unit Preparation . . . . . 153
- 7.9. Compilation Unit Execution . . . . . 156
- 7.10. Performance Improvements . . . . . 156
- 7.11. Statistics Manager . . . . . 158
  
- 8.1. Flow Execution Plan . . . . . 163
- 8.2. Navigator Structure . . . . . 182
- 8.3. Variable Usage Optimization Results . . . . . 191
- 8.4. XPath Processing Strategies . . . . . 199
- 8.5. Intra Transaction Cache Architecture . . . . . 205
- 8.6. Parallel Cache Usage . . . . . 209
- 8.7. Intra Transaction Cache Statistics . . . . . 211
- 8.8. Multi Cache Mode Persistence Statistics . . . . . 215
- 8.9. Single Cache Mode Persistence Statistics . . . . . 215
- 8.10. Optimization Performance Improvements . . . . . 222
- 8.11. Statistic-Supported Optimization Processing . . . . . 225
  
- 9.1. System Optimizer . . . . . 230
- 9.2. System Statistics Manager . . . . . 231
- 9.3. Running the IBM DB2 Configuration Advisor . . . . . 240
- 9.4. Index Optimization . . . . . 247
  
- 10.1. Caching Synchronous Web Services Invocations . . . . . 253
- 10.2. Performance Improvements for Service Request Caching . . . . . 255
- 10.3. Performance Improvements for Memory Execution . . . . . 256
  
- 11.1. Audit Manager Architecture . . . . . 259
- 11.2. Audit Trail Impact . . . . . 261

12.1. Single Server Topology . . . . .	270
12.2. Two Tier Structure . . . . .	271
12.3. Shared Database . . . . .	272
12.4. IBM WebSphere Cluster . . . . .	273
12.5. Advanced JMS Messaging . . . . .	274
12.6. Shared Intra Transaction Cache . . . . .	275
12.7. Node Sphere Processing . . . . .	278
A.1. Benchmark Choreography . . . . .	286
B.1. Data Sources . . . . .	317
B.2. Queues . . . . .	317

# LIST OF TABLES

B.1. Administration Components . . . . .	318
B.2. Administration Interface Components . . . . .	321
B.3. Process Execution Components . . . . .	322
B.4. Shared Components . . . . .	323
B.5. Shared EJBs Components . . . . .	324
B.6. Implementation Statistics . . . . .	326



# INDEX

- IBM DB2
  - Optimizations, 237
- IBM WebSphere clustering, 272
- OID
  - Generation, 85
- Process Deployment Descriptor, 55
- System Deployment Descriptor, 54
- ActiveEndpoints ActiveVOS, 31
- Activity
  - Assign, 81
  - Functions, 48
  - Life cycle, 48
  - Processing, 80
  - Retrieve, 80
  - States, 48
  - Wait, 84
- Advanced JMS Messaging, 273
- Apache ODE, 27
- Architecture
  - Basic Processing, 49
  - Build time, 42
  - Databases, 42
  - Engine, 11
  - Hot Pooling, 45
  - Infrastructure, 43
  - Run time, 41
  - Synchronous Request Processing, 51
  - System Profile, 42
- Assign
  - Processing, 81
- Assign Activity
  - Part Processing, 83
- Audit
  - Context Based Selection, 262
- Audit Trail, 257
  - Audit Manager, 259
  - Event Type Selection, 262
  - Multiple Audit Trails, 263
- Audit trail
  - Table, 258

- Base Architecture, 38
- Batching database operations
  - audit trail, 261
- Benchmark, 26
  - Main process, 286
  - Structure, 285
- Benchmark Process
  - WSDL, 287
- Bufferpool
  - Setup, 243
- Buildtime tables
  - Web Services, 63
- Cache
  - Instance Cache, 105
  - Model, 102
  - Multi Cache Execution, 208
  - Single Cache Execution, 211
- Cache Persistence, 212
- Caching
  - Cluster, 274
  - Correlation Caching, 136
  - Service request result, 253
- Component
  - Build time, 42
  - Run time, 41
- Configuration, 54
  - Database, 238
  - Databases, 139
- Connection, 232
  - Connection Pool Size, 232
- Context
  - Audit Trail, 262
- Control
  - Audit trail, 259
- Correlation, 142
  - Processing, 78
- Correlation Set
  - Usage Optimization, 196
- Database
  - Buildtime database, 58
  - Indices, 244
  - Query Optimizer, 240
    - RUNSTATS, 240
  - Runtime database, 64
  - Setup, 241
  - System Database, 56
- Databases, 56
  - Configuration, 139
- Deployment
  - Process, 88
- Engine
  - Architecture, 11
  - Implementation, 12
- Execution history, 68
- Flow Compilation, 153
- Flow Execution Plan, 162
  - Management, 223
  - Process A, 300
- Flow Optimizer, 162
- Functions
  - Activity, 48
  - Process, 46

- IBM Process Server, 33
- Implementation
  - Engine, 12
- Index, 246
  - Cluster index, 245
- Indices
  - Setup, 244
- Infrastructure
  - Selected, 43
- Intra Engine Binding, 130
- Intra Transaction Cache, 204
- Intra-Transaction Cache, 275
- Invocation
  - Inline Invocation, 132
  - Intra Engine Binding, 130
  - SOAP/HTTP Bypassing, 130
- Java Virtual Machine, 236
- JBoss jBPM, 30
- JMS Message Engine, 236
- Large Objects, 141
- Life cycle
  - Activity, 48
  - Process, 46
- LOB, 141
- Management
  - Process, 40
  - Process instances, 40
- Memory execution, 255
- Messages
  - Persistence, 250
- Micro Flow, 135
- Model
  - Cache, 102
- Navigation, 67
  - Advanced Architecture, 182
- Navigator, 50
  - Basic Structure, 68
- Operation administrator, 41
- Optimization
  - Application Specific, 226
  - Correlation Set, 196
  - External Costs, 227
  - Road Map, 223
- Optimizations
  - IBM DB2, 237
- Optimizer
  - System Optimizer, 230
- Performance
  - Improvements, 221
  - Relevant Factors, 92
- Primary key, 140
- Process
  - Functions, 46
  - Life cycle, 46
  - Management, 40
  - Queries, 265
  - Query, 41
  - States, 46
  - Table, 59
- Process based case, 5
- Process Deployment Descriptor, 297
- Process Instance

- Deletion, 148
  - Completion Deletion, 149
  - Deferred Deletion, 149
  - Ongoing deletion, 152
- Process instance
  - Management, 40
- Process instance monitor, 266
- Processing
  - Transaction-less, 251
- Queries
  - Process, 265
- Query
  - Process, 41
- Referential Integrity, 139
- Reliability, 87
- Retrieve
  - Processing, 80
- Scheduler, 233
- Service
  - Invocation, 52, 85
- Service Invoker
  - Request Processing, 75
- Setup
  - Bufferpools, 243
  - Databases, 241
  - Indices, 244
  - Tablespaces, 242
- Shared Database Topology, 271
- SOAP/HTTP
  - bypassing, 130
- Software
  - Stack, 9
- Spheres
  - Node, 275
- State
  - Activity, 48
  - Process, 46
- Statistics, 157
  - StatisticsManager, 157
  - System Statistics, 231
- System
  - Administrator, 41
- System Deployment Descriptor, 108
- Tables
  - Process table, 59
- Tablespaces
  - Setup, 242
- Topology
  - Single Server, 270
- Transaction
  - Internal Processing, 185
- Transaction Flow
  - Types, 112
- Two-level programming, 5
- Two-tier structure, 270
- Variable
  - Load Optimization, 195
  - LOB, 141
  - Usage Optimization, 190
- Wait Activities, 233
- Wait activity, 84
- Web Services

- Buildtime tables, 63
- WebSphere
  - Optimizations, 232
  - Prepared Statement Cache Size, 233
- Workflow Management System
  - Basic Processing, 49
  - IBM Process Server, 33
  - Synchronous Request Processing, 51
- Workflow Management Systems, 27
  - ActiveEndpoints ActiveVOS, 31
  - Apache ODE, 27
  - JBoss jBPM, 30
- XPath, 198