Universidad de los Andes
Faculty of Engineering
Department of
Systems and Computational
Software Construction Group

Vrije Universiteit Brussel
Faculty of Science
Department of
Computer Science
Software Languages Lab

Monitoring and Analysis of Workflow Applications: A Domain-specific Language Approach

Oscar González

September 2010

A dissertation submitted in partial fulfilment of the requirements for the degree of Doctor of Sciences at Vrije Universiteit Brussel and Doctor of Engineering at Universidad de los Andes

Promoters:
Prof. Dr. Rubby Casallas
Prof. Dr. Viviane Jonckers
Dr. Dirk Deridder

Jury:
Prof. Dr. Wolfgang De Meuter
Prof. Dr. Jorge Villalobos
Prof. Dr. Mario Südholt
Dr. Eric Tanter
Prof. Dr. Beat Signer
Dr. Dario Correal

Abstract

The increased focus of companies on improving their operational efficiency has raised a demand for advanced systems to support the continuous monitoring and analysis of business activities. These business activities are typically automated into workflow applications which are modeled, implemented and executed by workflow management systems (WFMS). The monitoring and analysis solutions aim at giving organizations the opportunity to focus on process improvement by detecting problematic properties of workflow applications. The focus of this research is to raise the level of abstraction for workflow developers for monitoring and analyzing workflow applications during their execution.

Contemporary solutions require a big effort from workflow developers to materialize the needs of workflow analysts. First of all, these solutions fall short because they do not treat monitoring and analysis as a first class entity in the workflow implementation. As a result, workflow developers need to build ad hoc infrastructures to instrument the workflow implementation with monitoring and analysis concerns. This results in crosscutting and entangled code that affects the maintainability of the workflow application and the monitoring and analysis implementation. Another consequence is that the implementation of monitoring and analysis concerns is not reusable across different workflow platforms. Secondly, workflow variables and measurement variables are difficult to localize, use and share between workflow developers. As a result, there is limited support to base the monitoring and analysis on the data of the workflow application domain (*e.g.,* Banking). Thus, the expressiveness of monitoring and analysis concerns is limited to generic properties defined in terms of the operational state of the workflow engine (*e.g.,* the time a workflow is running).

This dissertation presents two main contributions to solve these problems. Our first contribution is the creation a domain-specific language named MonitA to specify monitoring and analysis concerns in a uniform and workflow technol-

ogy independent way. MonitA specifications make extensive use of the data available in the workflow application and its constituents to tackle the lack of expressiveness in these specifications. Our second contribution is a strategy to generate monitoring and analysis code into different existing workflow platforms. MonitA specifications generate monitoring code that is composed with the existing workflow implementation in an automated fashion. The resulting code artifacts correspond to an executable workflow implementation instrumented with monitoring and analysis concerns that can be executed in a particular workflow engine.

We provide a flexible approach that offers workflow developers the possibility to target different workflow platforms and different workflow applications. The monitoring and analysis specifications are not tightly coupled to the workflow implementation since they make use of the process model concepts. The MonitA specifications are modularized to ease their maintainability and to be shared between workflow developers. This has a positive impact on evolvability since these specifications can be re-generated and re-composed with a new workflow implementation.

We have validated our approach by specifying monitoring and analysis concerns into well known workflow applications by using MonitA. We used our implementation strategy to automate the implementation of monitoring and analysis concerns in two different workflow platforms (*i.e.,* JPDL, BPEL). These experiments evidence improvements in the time and effort that workflow developers require to specify and implement monitoring and analysis concerns in an existing workflow application. They also improve the specification time in a new workflow application of a supported workflow platform. In addition, the code required to maintain is less in MonitA and it is modularized to ease its maintainability.

The MonitA language increases the expressiveness of monitoring and analysis concerns in terms of workflow relevant data to allow an analysis of the workflow application in terms of the specific domain it is modeling. The implementation of monitoring and analysis concerns is automatically generated and composed into a specific workflow platform. This increases the productivity of application developers and decreases the complexity and time required to implement monitoring and analysis concerns in workflow applications. This also has a favourable impact for workflow analysts since there is a time reduction to materialize their requirements and to get feedback required to take improvement decisions.

Resumen

El creciente interés de las empresas en el mejoramiento de su eficiencia operativa ha planteado una demanda de sistemas avanzados para apoyar el monitoreo y análisis continuo de sus actividades de negocio. Estas actividades de negocio suelen ser automatizados en aplicaciones de workflow las cuales son modeladas, implementadas y ejecutadas en workflow management systems (WFMS). Las soluciones de monitoreo y análisis dan a las organizaciones la oportunidad de centrarse en el mejoramiento de sus procesos mediante la detección de propiedades problemáticas en las aplicaciones de workflow. El objetivo de esta investigación es subir el nivel de abstracción para los desarrolladores del workflow para soportar el monitoreo y análisis durante la ejecución de estas aplicaciones.

Las soluciones actuales requieren un gran esfuerzo de los desarrolladores del workflow para materializar las necesidades de los analistas. En primer lugar, estas soluciones se quedan cortas porque no tratan el monitoreo y análisis como una entidad representativa en la aplicación de workflow. Como resultado, los desarrolladores necesitan construir infraestructuras diseñadas específicamente para instrumentar la aplicación de workflow con preocupaciones de monitoreo y análisis. Esto conlleva a código disperso y entrelazado que afecta la mantenibilidad de la aplicación de workflow y de la implementación de monitoreo y análisis. Otra consecuencia es que la implementación de las preocupaciones de monitoreo y análisis no es reutilizable a través de plataformas de workflow diferentes. En segundo lugar, las variables de workflow y las variables de medición son difíciles de localizar, utilizar y compartir entre los desarrolladores del workflow. Como resultado, el soporte para basar el monitoreo y análisis en los datos del dominio específico de la aplicación de workflow (*e.g.,* dominio bancario) es limitado. Por lo tanto, la expresividad de las preocupaciones de monitoreo y análisis se limita a propiedades genéricas acerca del estado del motor de workflow (*e.g.,* el tiempo que un workflow se está ejecutando).

Esta tesis presenta dos contribuciones principales para resolver estos prob-

lemas. Nuestra primera contribución es la creación de un lenguaje de dominio específico llamado MonitA para especificar preocupaciones de monitoreo y análisis de manera uniforme e independiente de tecnologías de workflow. Las especificaciones en MonitA hacen un amplio uso de los datos disponibles en la aplicación de workflow para hacer frente a la falta de expresividad de estas especificaciones. Nuestra segunda contribución es una estrategia para generar código de monitoreo y análisis en diferentes plataformas de workflow existentes. Las especificaciones en MonitA generan código de monitoreo que se integra con el código de la aplicación de workflow de forma automatizada. El código resultante corresponde a una aplicación de workflow ejecutable, la cual está instrumentada con preocupaciones de monitoreo y análisis y puede ser ejecutada en un motor de workflow concreto.

Proporcionamos un enfoque flexible que ofrece a los desarrolladores la posibilidad de seleccionar diferentes plataformas y aplicaciones de workflow. Las especificaciones de monitoreo y análisis no están estrechamente unidas a la implementación del workflow ya que hacen uso de los conceptos del modelo de proceso de alto nivel. Las especificaciones en MonitA son modulares para facilitar su mantenibilidad y para ser compartidas entre los desarrolladores del workflow. Esto tiene un impacto positivo en la capacidad de evolución ya que estas especificaciones pueden ser generadas y integradas con una nueva aplicación de workflow.

Hemos validado nuestro enfoque mediante el uso de MonitA para la especificación de preocupaciones de monitoreo y análisis en aplicaciones de workflow conocidas. Utilizamos nuestra estrategia generativa para automatizar la implementación de preocupaciones de monitoreo y análisis en dos plataformas de workflow diferentes (*i.e.,* JPDL, BPEL). El uso de las infraestructuras generativas creadas para estas plataformas muestra mejoras en el tiempo y esfuerzo que los desarrolladores requieren para especificar e implementar las preocupaciones de monitoreo y análisis para una aplicación de workflow existente. También mejora el tiempo de especificación para una nueva aplicación de workflow en una plataforma soportada por la infraestructura. Además, el código para mantener es menor en MonitA y es modular para facilitar su mantenimiento.

El lenguaje MonitA aumenta la expresividad de las preocupaciones de monitoreo y análisis en términos de los datos pertinentes al workflow. Esto permite un análisis en términos del dominio específico que la aplicación de workflow está modelando. La implementación de preocupaciones de monitoreo y análisis es automáticamente generada e integrada en una plataforma de workflow específica. Esto aumenta la productividad de los desarrolladores y reduce la complejidad y el tiempo necesarios para implementar las preocupaciones de monitoreo y análisis en las aplicaciones de workflow. Esto también tiene un impacto favorable para los analistas de workflow ya que hay una reducción del tiempo necesario para materializar sus necesidades y para obtener la retroalimentación necesaria para tomar decisiones de mejoramiento.

Samenvatting

Een belangrijk aandachtspunt voor bedrijven is de continue verbetering van hun operationele efficiëntie. Hierdoor is er een stijgende vraag naar geavanceerde systemen om bedrijfsprocessen te observeren, analyseren en controleren. Doorgaans worden bedrijfsprocessen geautomatiseerd door ze te modelleren, programmeren en uit te voeren met behulp van workflow management systemen (WFMS). De hulpmiddelen waarover men beschikt om deze workflows te observeren en te analyseren stellen organisaties in staat om problematische onderdelen te identificeren en vervolgens hun bedrijfsprocessen te verbeteren. Het doel van het onderzoek in dit proefschrift is het verhogen van het niveau van abstractie waarin ontwikkelaars deze vereisten voor observatie en analyse kunnen implementeren.

Hedendaagse WFMS vragen een grote inspanning van ontwikkelaars om de noden van workflow analisten te realiseren. In de eerste plaats behandelen bestaande ontwikkelingsomgevingen het observatie- en analyse aspect niet als volwaardige elementen van een WFMS. Hierdoor zijn ontwikkelaars genoodzaakt om zelf de nodige infrastructuur te bouwen voor het instrumenteren van workflow implementaties met de nodige observatie- en analysecode. Uiteindelijk bekomt men zo een workflowimplementatie waarin deze code verweven is met de code verantwoordelijk voor de definitie van het bedrijfsproces. Dit heeft nadelige gevolgen voor de onderhoudbaarheid en de herbruikbaarheid van het workflowsysteem in haar geheel. Ten tweede bieden bestaande WFMS geen ondersteuning om observaties en analyses te baseren op data en metingen die uitgedrukt zijn in termen van de waarden gebruikt in het bedrijfsproces (domein-specifieke data). Meestal blijft de geboden ondersteuning beperkt tot uitdrukkingen die gebruik maken van de operationele toestand van het WFMS systeem (bijv. de uitvoeringstijd, het aantal actieve processen). In het algemeen kan dus gesteld worden dat huidige WFMS onvoldoende expressiviteit bieden voor het uitdrukken van de analyse en observatienoden van workflow analisten.

Dit proefschrift bevat twee belangrijke bijdragen om de hogervermelde problematiek aan te pakken. Een eerste bijdrage bestaat in het aanbieden van een domein-specifieke taal (genaamd MonitA) waarin observatie- en analysegedrag kan worden uitgedrukt op een uniforme en technologie-onafhankelijke manier. Bovendien kunnen specificaties in MonitA uitgebreid gebruik maken van de domein-specifieke data die beschikbaar is binnen het workflowsysteem. Hierdoor biedt MonitA een oplossing voor het gebrek aan expressiviteit van bestaande WFMS. De tweede bijdrage omvat een generatiestrategie om, op basis van een MonitA specificatie, de benodigde observatie- en analysecode te genereren en te integreren in een bestaande workflowimplementatie. Uiteindelijk zorgt deze generatiestrategie er voor dat men op het einde van de rit een uitvoerbare implementatie heeft die uitgerust is met de nodige elementen voor observatie en analyse. De aangeboden generatiestrategie is opgebouwd op een manier die het mogelijk maakt om code te genereren voor verschillende WFMS platformen. Dit zonder dat er nadelige gevolgen zijn voor de bestaande MonitA specificaties. Bijgevolg kan men stellen dat de generatiestrategie verantwoordelijk is voor het technologie-onafhankelijke aspect van onze bijdrage.

De voorgestelde aanpak is gevalideerd door het opstellen van MonitA specificaties voor observatie en analyse en dit voor verschillende gevalstudies. Hierbij is de generatiestrategie succesvol ingezet om code te genereren voor verschillende WFMS platformen (JPDL, BPEL). Deze experimenten hebben een bewijs geleverd dat workflow ontwikkelaars op een snellere en minder arbeidsintensieve manier observatie- en analysegedrag kunnen toevoegen aan een bestaande workflowimplementatie. Ook is gebleken dat nieuwe WMFS platformen op een gemakkelijke manier ondersteund kunnen worden. Hierdoor kan men dezelfde MonitA specificatie gebruiken om code te genereren voor verschillende WMFS platformen. Hierdoor zijn workflow ontwikkelaars niet langer gebonden aan een bepaalde technologie bij het uitdrukken van de vereisten voor observatie en analyse. Verder is gebleken dat het gebruik van MonitA resulteerde in compactere en beter gemodulariseerde code. Dit leverde een niet te onderschatten voordeel op voor de algemene onderhoudbaarheid van de workflowsystemen.

Acknowledgements

These few lines won't be enough to express my thankfulness to all my advisers, colleagues, friends and family, who have assisted me one way or another during this long but enriching process. Therefore, I want to dedicate this thesis and all its positive results to all people who have supported me all along ...

... To my advisers Rubby Casallas, Dirk Deridder, and Viviane Jonckers, who gave me the opportunity to work with them as a PhD student. I am immensely grateful to Rubby for introducing me into the research world, for giving me all the support and advice required during these four years of research, and for providing me with valuable feedback to improve this thesis. I thank deeply Dirk for his critical advice, for his constant motivation that encouraged me to keep a good performance, and for the enormous time spent reading my texts in depth. I am grateful to Viviane for providing me with constructive feedback on my work and for the coordination of the activities to finish my thesis.

... To the committee members, for the significant time spent reading my thesis and their rigorous comments to improve it: Wolfgang De Meuter, Jorge Villalobos, Mario Südholt, Eric Tanter, Beat Signer, and Dario Correal.

... To the Flemish Interuniversity Council (VLIR) in the framework of the CARAMELOS project and to Colciencias for providing me with the financial support to carry out this work.

... To my colleagues and friends Andrés Yie and Mario Sánchez who gave me support and help during the time living in Belgium, I feel very much indebted.

... To the members of the TICSw research group for their collaboration in my research. In particular, I thank to Marcial Moreno and William Cano who collaborated in the implementation and validation of the MonitA generative infrastructure.

... To my colleagues at Software Languages Lab for the scientific and productive working environment. I thank deeply the experience and friendship

from my colleagues at the past SSEL lab: Bruno De Fraine, Dennis Wagelaar, Niels Joncheere, Bart Verheecke, Ragnhild Van Der Straeten, Mathieu Braemand, and Eline Philips. I particularly want to thank Eline and Andy Kellens to proof-read some of the chapters of my thesis, and Bruno and Dirk for helping me with the Dutch version of the abstract. Additionally, I thank to Thomas Cleenewerck for his guidance and discussions on DSLs, to Carlos Noguera and Coen De Roover for their suggestions and valuable feedback to improve my work, and to Elisa Gonzalez and Jorge Vallejos for their support to coordinate the practical arrangements of my public defense.

... To all my friends who supported me during the different periods of time living in Colombia and Belgium. Thanks to Rob Vanmeert, Adriana Manrique, Isabel Michiels, Frank van der Kleij, Agustina Cibran, Jorge Vallejos, Sonia Petitprez, Hugo Arboleda, Nicolas Cardozo, and Sebastian González for their interest in my research and all-round support.

... To my parents, Buenaventura González and Flor Rojas, for the right academic and personal guidance, and for their infinite support. Thanks to my brother, sister, and family for their love and unit to face any challenge.

... To Diana, my eternal girlfriend, for supporting me in these years of doctoral studies, for bearing my long periods of absence being abroad, and for helping me find the balance in my life.

Thanks a lot !!!

Table of Contents

List of Figures

List of Tables

Listings

List of Abbreviations

AOM	Aspect-oriented Modeling
AOP	Aspect-oriented Programming
AOSD	Aspect-oriented Software Development
AST	Abstract Syntax Tree
BAM	Business Activity Monitoring
BNF	Backus-Naur Form
BOM	Business Operations Management
BPA	Business Process Analysis
BPEL	Business Process Execution Language
BPI	Business Process Intelligence
BPMI	Business Process Management Institute
BPMN	Business Process Modeling Notation
COTS	Commercial Off-The-Shelf
DARE	Domain Analysis and Reuse Environment
DSL	Domain-specific Language
EBNF	Extended Backus-Naur Form
EMF	Eclipse Modeling Framework
FAST	Family-oriented abstractions, Specification, and Translation
FODA	Feature-oriented Domain Analysis
GPL	General Purpose Language
jBPM	Java Business Process Management
JPDL	jBPM Process Definition Language
M&A	Monitoring and Analysis
MDE	Model-driven Engineering
MMC	Monitoring, Measurement and Control
OCL	Object Constraint Language
ODE	Ontology-based Domain Engineering
OMG	Object Management Group
PPI	Process Performance Indicators

SoC	Separation of Concerns
SOC	Service-Oriented Computing
SOA	Service-oriented Architecture
WfMC	Workflow Management Coalition
WFMS	Workflow Management Systems
WSDL	Web Service Description Language
XML	Extensible Markup Language
XPath	XML Path Language
XPDL	XML Process Definition Language
XQuery	XML Query
XSD	XML Schema Definition
XSL	Extensible Stylesheet Language
XSLT	XSL Transformations
YAWL	Yet Another Workflow Language

Part I

Problem Statement and Background

Chapter 1

Introduction

1.1 Research Context

The increased focus of companies on measuring and improving their operational efficiency has raised a demand for advanced systems to support the continuous monitoring and analysis of their business activities. These business activities are typically automated into workflow applications which facilitate the coordination of human and technological sources of information in an organization according to a formally defined process model that contains a set of linked activities that fulfill a policy goal [vdAtHW03] [zM04]. Workflow applications are modeled, implemented, executed, and analyzed by workflow management systems (WFMS) through the use of software, running on one or more workflow engines [wfm99] [vdAvH04]. These four phases form a life-cycle that drives the evolution of workflow applications. Nevertheless, the focus of our research is on the workflow analysis phase. Workflow analysis solutions aim at giving organizations the opportunity to focus on process improvement by detecting problematic properties of workflow applications.

The analysis phase involves different monitoring and analysis techniques. The monitoring can be performed *a posteriori*, at runtime, or with a combination of both to capture measurement information. There are also different analysis techniques such as verification and validation of process models (*e.g.,* conformance checking, correctness), debugging and profiling, predictions, and the evaluation of measurement information. In that context, the type of analysis supported by this work is limited to the evaluation of measurement information against a set of expected quality properties defined in terms of the workflow data.

We center our research on raising the level of abstraction for workflow developers for monitoring and analyzing workflow applications at runtime. Our goal is to assess changes in the quality of workflow applications through strategic measures defined early to provide feedback for workflow analysts. The work-

flow monitoring and analysis at runtime decreases the time required to identify problematic aspects about critical properties in the workflow and to decide on workflow improvements (*e.g.,* re-assign resources, add activities). Continuous workflow improvements address the requirement of companies to be adaptable to internal and external changes. We consider that an accurate identification of problematic aspects at runtime in workflow applications can be supported by implementing monitoring and analysis concerns such as: monitoring, measurement and control. These monitoring and analysis concerns are defined for monitoring the actual workflow executions, building and managing measurement information based on the data manipulated by the workflow application, and applying notification actions based on the evaluation of those data.

The original goal of WFMS is to empower business users to make changes in the workflow applications. Nevertheless, in reality they still have to ask IT users to do it for them. The process modeling and workflow implementation phases define the main infrastructure to enact and analyze workflow applications. BPMN [OMG06a] is currently the de facto standard to facilitate *process experts* in the creation of process models at a conceptual level of abstraction and independently of workflow technologies. These process models are used to partially generate the workflow implementation towards a particular workflow platform. Then *workflow developers* complement this workflow implementation by defining the actual code of the activities, their data management, and the integration with external systems. Workflow developers create the executable workflow application by using different workflow technologies such as workflow languages (*e.g.,* XPDL [XPD], BPEL [IBM02], JPDL [JPD]) and workflow engines (*e.g.,* Apache ODE, jBPM). This workflow generation process defines the different levels of abstraction and stakeholders that have to be considered to support the workflow monitoring and analysis at runtime.

The importance of workflow monitoring and analysis has been demonstrated by the multiple tools and publications related to this topic [zMR00] [MHH07]. This is also reflected by multiple emerging terms related to monitoring and analysis of workflow applications such as Business Activity Monitoring (BAM), Business Operations Management (BOM), Business Process Intelligence (BPI), and Business Process Analysis (BPA).

Considering the relevance of workflow monitoring and analysis, workflow technology should provide a mechanism with suitable flexibility to monitor and analyze workflow applications. We consider that the main needs for a flexible workflow monitoring and analysis approach are: understandability, expressiveness, maintainability, reusability, and productivity. First, the specification of monitoring and analysis concerns must be expressed in terms of the workflow monitoring and analysis domain to improve *understandability*. Second, the *expressiveness* in the specification of monitoring and analysis concerns needs to be improved by involving the data managed by the workflow application to incorporate custom measurements specific to the domain of the workflow. Third,

in the same way that the workflow applications evolve continuously as a consequence of the business evolution, the monitoring and analysis specifications need to be maintained and should co-evolve with the workflow specification. Fourth, the monitoring and analysis specifications need to be *reusable* between different workflow platforms and therefore should avoid a tight coupling with the workflow implementation. Finally, whereas the goal of workflow analysts is to identify potential improvements in workflow applications based on the definition of custom monitoring and analysis needs, the goal of workflow developers is to implement these needs with low cost and high *productivity*.

The following sections illustrate the limitations that workflow technologies have to monitor and analyze workflow applications at runtime and the challenges we face to tackle these limitations. We also introduce our approach for a flexible and expressive workflow monitoring and analysis solution.

1.2 Problem Statement

This section describes the problems we have identified in WFMS to offer support to workflow developers for specifying monitoring and analysis concerns. We first discuss the problems in detail by using a case study and then we summarize these problems.

Despite the tools and techniques developed for monitoring and analysis (M&A) at runtime, workflow developers have to manually intervene in the workflow implementation to include code that implements custom M&A solutions. This is a complex task to instrument each workflow application for each different workflow platform. For example, several commercial workflow management products and architectures (*e.g.,* Intalio, IBM, Oracle) offer solutions for business activity monitoring (BAM) [MHH07]. These solutions offer rich dashboards to visualize predefined measurements and to write queries on demand to extract the information required for the reporting tools. Nevertheless, typically workflow developers have to instrument the workflow platforms and applications by adding adapters to generate custom workflow events, to add custom measurements, and to send this information to the BAM architecture.

In contrast with the fact that workflow applications evolve continuously as a consequence of business necessities, contemporary monitoring and analysis solutions do not co-evolve with the workflow specification. Workflow technology evidences the difficulty of analyzing the execution of workflow applications from a conceptual perspective [HLD+05]. As described above, the automation of a workflow application starts from a conceptual process model provided by a process expert followed by a workflow implementation done by a workflow developer. In contrast, the monitoring and analysis specifications are materialized by workflow developers directly in the workflow implementation, thus skipping the conceptual stage.

1.2.1 A Trouble Ticket Workflow Scenario

To illustrate the problems for specifying monitoring and analysis concerns in workflow applications, we present a small extract of code of a trouble ticket workflow application. The trouble ticket process model defines the activities required to manage the processing of problem claims in a software product within an organization [Nor98]. Figure 1.1 illustrates the trouble ticket process model using BPMN notation.

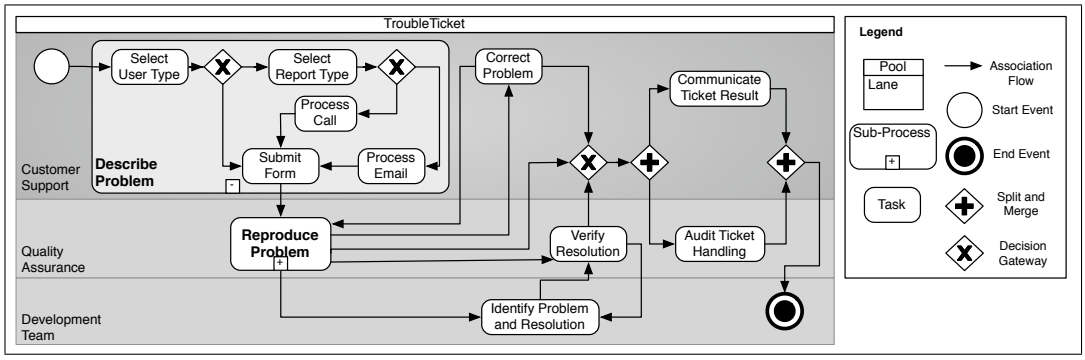


Figure 1.1: Simplified process model of a trouble ticket workflow application

A trouble ticket workflow involves multiple teams such as quality assurance, customer support, and developers. A ticket is created when a problem is detailed and recorded (*Describe Problem* sub-process). Then the record is checked until it can be reproduced (*Reproduce Problem* sub-process), and the intrinsic cause is identified and a resolution is provided (*Identify Problem and resolution* activity). Once the resolution is verified (*Verify Resolution* activity), the result is communicated to the originator (*Communicate Ticket Result* activity), and the problem and resolution are included in a knowledge repository (*Audit Ticket Handling* activity). Data entities such as Problem, Report, Originator, and Resolution are typically found in the trouble ticket application. For example, a Problem entity involves attributes such as id, description, product, expert, and area of expertise, with their data types. Consider for example a ticket created with a problem provided by a customer such as *"access denied in product X"*, which is associated to the area of expertise named *"area1"*.

The following example defines a set of monitoring and analysis concerns required to evaluate the workflow application at runtime and to provide feedback for workflow analysts:

- *Measuring.* Create measures to accumulate the number of problems reported by each area of expertise (*e.g.,* area1). These are custom measures that have to be defined and computed in terms of the workflow data and that span multiple workflow instances.

- *Monitoring.* Capture the area of expertise that is required to compute the measures mentioned above. This information must be captured right after the problem is recorded within the execution of the *Submit Form* activity and when the expert is reassigned within the *Identify Problem and Resolution* activity.

- *Controling.* Evaluate the number of problems reported by a specific area of expertise. If this measure is more than 10, then an alarm must be generated and visualized in a dashboard.

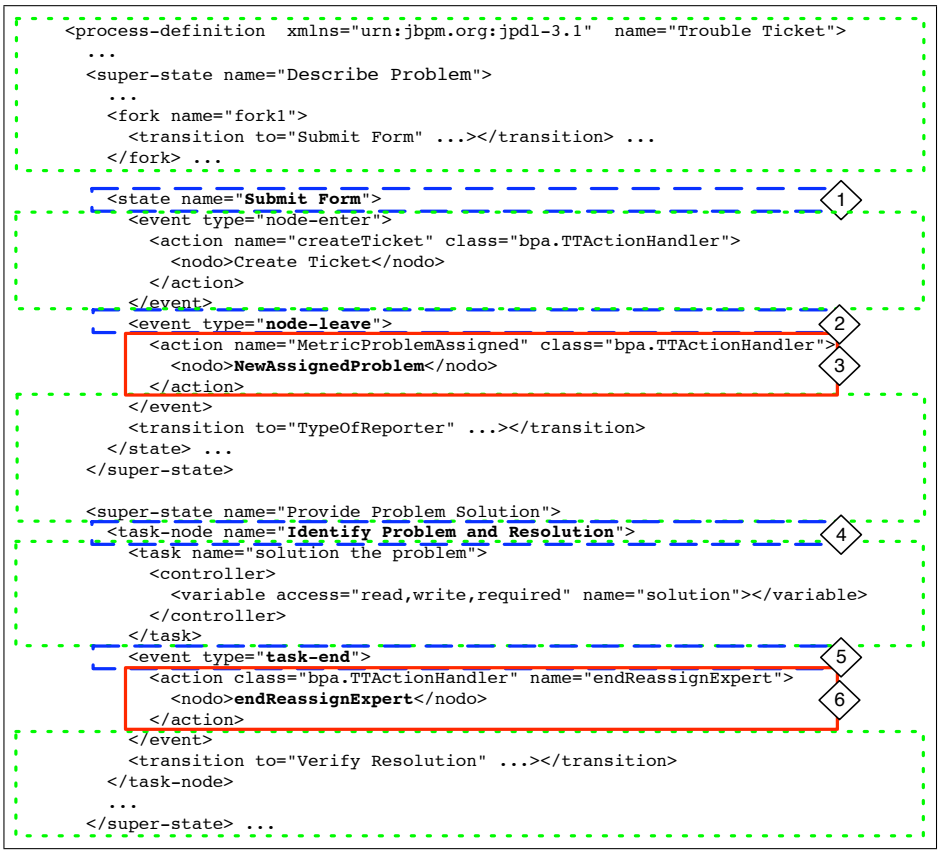
The goal of these concerns is to identify problematic areas of expertise where a high number of problems is reported. These monitoring and analysis concerns can be used to define potential improvements regarding a particular business goal. For example, for a high number of problems reported in a specific area of expertise, the manager of that area can visualize this information immediately and decide to stop the execution of the workflow application to include a new quality assurance activity. This new activity can for example be assigned to a new resource that can help to reduce the number of problems in that specific area.

1.2.2 The Need for Higher-level Mechanisms

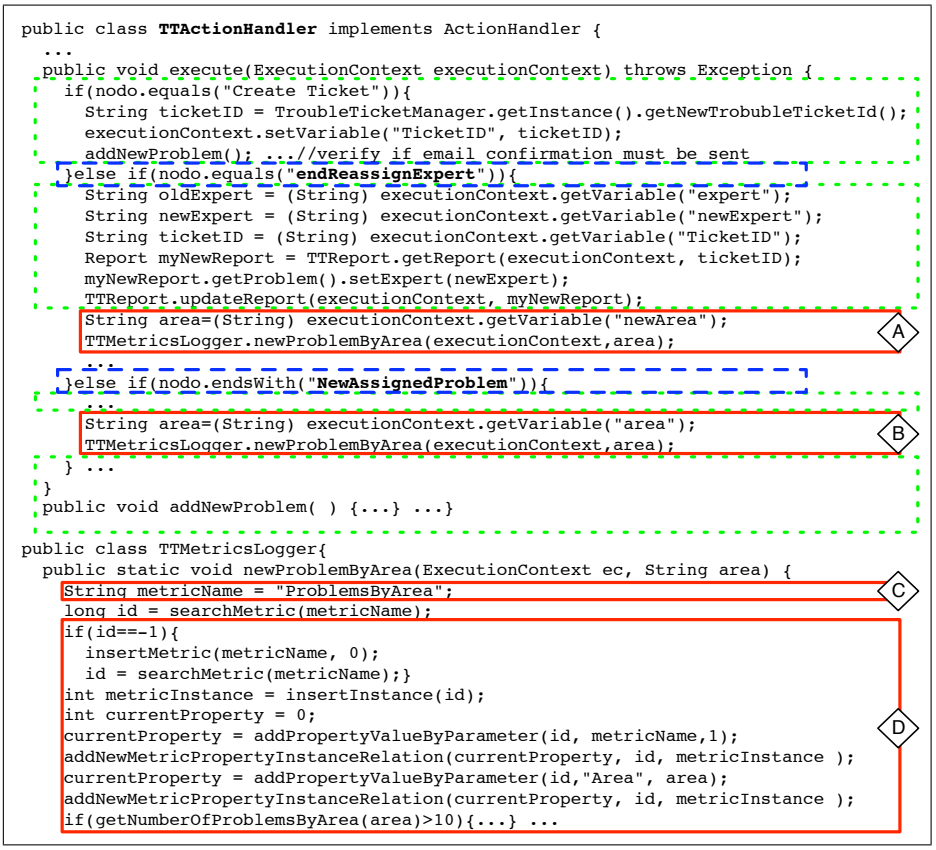
The Trouble Ticket process model presented above is implemented in the Java Process Definition Language (JPDL) [JPD]. As a result, the workflow developers have no choice but to do the monitoring and analysis concerns specification in JPDL as well. The JPDL workflow implementation consists of a *workflow definition* (an xml file) artifact that describes how workflow entities must interact, and the *underlying application code* (Java-classes) implementing these workflow entities.

Figure 1.2 illustrates a small extract of code from the workflow application implementation. In this code we distinguish between the workflow implementation code (dotted green line border) and analysis concerns (red line border) code. The blue code (dashed line border) illustrates glue code between the workflow implementation and the monitoring and analysis concerns.

This fragment of code shows that multiple monitoring and analysis concerns occur at different places in the workflow definition file. This phenomenon is commonly known as code scattering, which means that a single requirement affects multiple design and code modules [TOHJ99]. In addition, this fragment of code shows that analysis concerns are mixed with the workflow implementation. This phenomenon is commonly known as code tangling, which means that an element concerning multiple requirements is interleaved within a single module [TOHJ99]. The phenomenon of tangled and scattered code is known as crosscutting code in the area of Aspect-Oriented Programming (AOP) [KLM+97].



(a) Workflow Definition File



(b) Delegated Application Code

Figure 1.2: An example of crosscutting analysis concerns.

Crosscutting code compromises the comprehension and evolution of the workflow application and its monitoring and analysis concerns. Thus, when an analysis requirement changes, its implementation has to be repeated or adapted in multiple artifacts of the workflow implementation. Since monitoring and analysis concerns are added manually in several locations into a generated workflow implementation, the workflow generation mechanism cannot be used anymore because it overrides the analysis implementation. There are solutions that mark the manual code and avoid to override it. However, this code has to be adapted in several locations to fit the new requirements. Additionally, monitoring and analysis concerns can be seen as specific knowledge *about* the workflow application. Thus, it is not appropriate to entangle that knowledge in the workflow code where it is complex to identify and to reuse. Moreover, making changes to the workflow implementation or the analysis concerns implementation might break the functionality of the other one.

The code snippet in Figure 1.2 (a) is responsible for implementing the *monitoring* requirement of the trouble ticket application. This implementation intercepts workflow states where the required monitoring information has to be captured. For the selected workflow elements (Figure 1.2 (a) parts 1 and 4), a new workflow element (*e.g.,* event in Figure 1.2 (a) part 2 and 5) is created to intercept the desired state in the workflow execution (*e.g.,* node-leave). If the new workflow event already exists for the regular workflow execution, some properties (*e.g.,* action in Figure 1.2 (a) part 3 and 6) need to be added to this new element. These new workflow events are added in the workflow definition for each workflow interaction that is to be intercepted for implementing workflow and analysis code. A change done to these events can affect several elements supporting the workflow implementation and the monitoring and analysis implementation. These workflow execution states delegate the execution of custom monitoring and analysis concerns to a Java-class (Figure 1.2 (b)).

The code snippet in Figure 1.2 (b) is responsible for implementing the *measurement* and *control* requirements. In order to compute the metric (*i.e.,* number of problems reported by area of expertise), the actual area of expertise associated to the problem has to be queried and passed to the monitoring and analysis concerns implementation (Figure 1.2 (b) part A and B). However, this domain-specific information is not explicit in the workflow definition but is managed in the execution context. We can observe duplication of code. Thus, workflow developers require knowledge of the concrete workflow execution engine to be able to create and retrieve this information. Figure 1.2 (b) part C and D, represents the application code required to manage the specified control requirements such as generate an alarm upon the evaluation of an indicator. Note that the measurement variable is defined directly in the application code (Figure 1.2 (b) part C). This analysis knowledge is entangled with the workflow implementation, thereby it is complex to localize and reuse.

Adding new monitoring and analysis concerns to the workflow application requires knowledge of the workflow implementation. This requires manual adaptations in several locations of the workflow code. Nevertheless, adaptations done in the workflow application or in the analysis concerns should not affect each other. Furthermore, workflow developers have to deal with the complexity of technologies used in the workflow implementation (*i.e.,* workflow language, application code). The original goal of WFMS is to empower business users to make changes in the workflow applications, however, in reality workflow developers still have to do this task.

1.2.3 An Overview of the Problem

We can summarize the above problems as follows:

P1 Monitoring and analysis concerns result in an entangled low-level implementation. Contemporary solutions fall short because they do not treat M&A as a first class entity in the workflow implementation. As a result, workflow developers need to build ad hoc infrastructures and abstractions to instrument the workflow implementation with monitoring and analysis concerns. This low-level implementation results in crosscutting and entangled code that affects the maintainability of the workflow application and of the monitoring and analysis implementation. The lack of modularization has repercussions on evolvability since the monitoring and analysis concerns cannot co-evolve with the fast business change.

Additionally, fragments of the code implementing the workflow are generated. This complicates the implementation of monitoring and analysis concerns since it requires knowledge of the generated workflow elements such as their names, attributes and state. This also means that the monitoring and analysis implementation is tightly coupled to the naming conventions used by the selected platform.

A low-level implementation of monitoring and analysis concerns also requires experts in the workflow language (*e.g.,* JPDL, BPEL), in the underlying implementation language (*e.g.,* Java) used to implement the activities, and in the workflow engine (*e.g.,* jBPM, Apache ODE). For example, the way to specify a set of monitoring and analysis concerns in BPEL is different to the same specification in JPDL (*e.g.,* access workflow instances, control flow constructs). Thus, monitoring and analysis concerns are not reusable as specified across different workflow platforms.

P2 Workflow monitoring and analysis solutions do not provide support to base the analysis on the data used in the workflow application. In traditional monitoring environments, workflow developers can specify analysis concerns only in terms of predefined measure-

ments about the operational state of the workflow engine. For example, the time a workflow is running, the number of workflow instances, the utilization of workflow resources, and the current state of a workflow instance. Nevertheless, when the analysis concerns have to be specified in terms of the workflow relevant data, it is done a posteriori (*e.g.,* by data mining). This is because the internal workflow variables are not always explicitly represented in the process models nor in the workflow implementation. The workflow variables are encoded in the workflow implementation, thereby they are difficult to localize and query.

Current business scenarios require identifying potential workflow improvements faster, depending on the online evaluation of custom measurements. This need can be supported by increasing the expressiveness of monitoring and analysis concerns to custom measurements specified in terms of the particular domain of the workflow application (*e.g.,* Banking, Customer Support, Insurance). We refer to *application-specific measurements* as custom measurements defined in terms of the data managed by the workflow application. Application-specific measurements are required to evaluate a workflow application in terms of business goals. However, workflow developers have to encode new custom measurements in the workflow implementation. Thus, these custom measurements are difficult to localize, use, and share with other workflow developers that require to specify monitoring and analysis concerns based on this data.

1.3 Research and Assessment Goals

The main goal of this dissertation is to achieve improved separation of monitoring and analysis concerns in workflow applications. Within the domain of software engineering, the principle of separation of concerns is applied to modularize software systems such that various modules of the system can be treated in isolation of other system modules [Dij76]. Our main challenge is dealing with the monitoring and analysis concerns specification that is inherently changeable in workflow applications. Thus identifying and managing these monitoring and analysis concerns knowledge explicitly is crucial to maintain it according to the business needs and workflow application evolution.

The following are the specific goals we have defined and the challenges we have identified to tackle the problems outlined in the previous section.

G1 *Raise the level of abstraction for specifying monitoring and analysis concerns.* Workflow developers must be able to co-evolve the monitoring and analysis specification along with the workflow evolution (see Problem P1). The monitoring and analysis concerns must be specified in a uniform and workflow technology independent way to reuse them across

different workflow platforms. One challenge is to figure out how to map implementation concepts into modeling concepts.

Our solution must automate the implementation of monitoring and analysis concerns to ease the maintainability of these specifications. This requires to trace the mappings performed by the workflow generation process to specify monitoring and analysis concerns in terms of process models and also to incorporate them in the actual workflow implementation. The code that connects monitoring and analysis concerns with the workflow application must be identifiable to ease its maintainability.

G2 *Increase the expressiveness of monitoring and analysis in terms of workflow relevant data.* Our solution must be able to support the specification of monitoring and analysis concerns in terms of the domain the workflow application is modeling (see Problem P2). Thus, a mechanism to expose a projection of this internal data to the outside world has to be provided [Hel04]. In this way, a subset of the workflow variables and internal information can be accessed and shared to involve them in monitoring and analysis activities external to the regular workflow execution.

Our solution must be able to intercept not only workflow events in terms of flow entities (*e.g.,* activity finished) but also customize the instrumentation of workflow applications to intercept fine-grained workflow events in terms of data entities (*e.g.,* workflow variable changed). Another challenge is to allow workflow developers to specify and manage custom measurements that can be evaluated at runtime. The historic measurement information and workflow data must be managed independently of the workflow application.

The following summarizes a set of assessment goals to evaluate to which extent the goals presented previously can be met in our solution. These assessment goals correspond to an overview of the validation performed in this dissertation. Chapters 8 and 9 present the experimentation and validation conducted to achieve these evaluation goals in our solution.

AG1 Apply the proposed approach in different workflow platforms to evaluate to which extent it is applicable in a workflow technology independent way. It is necessary to evaluate our approach with respect to the time of adaptation to new workflow platforms. The time required to implement monitoring and analysis concerns by using our approach must be measured and compared with respect to the time in the context of a low-level implementation in each workflow platform. Our approach must be evaluated with respect to the effort of adaptation to the monitoring and analysis concerns, namely, the number of lines of code (LOCs) that have to be maintained. The evaluation of these criteria can determine to which extent we achieve goal G1.

AG2 Use multiple workflow applications to assess the expressiveness offered by the proposed approach in different domains. Multiple monitoring and analysis concerns must be specified for each workflow application to evaluate the expressiveness of these specifications in terms of workflow relevant data. These specifications must be easily identified and shared by multiple workflow developers. The evaluation of these criteria can determine to which extent we achieve goal G2.

1.4 Approach

In the previous sections we identified a set of problems and goals to specify monitoring and analysis concerns in workflow applications. The research presented in this dissertation concentrates on two fundamental topics: 1) the specification and 2) the implementation of monitoring and analysis concerns. The strategy used to conduct the research in these two topics presents a solution to the research goals described above.

Figure 1.3 illustrates our overall approach for specifying and implementing monitoring and analysis concerns in workflow applications.

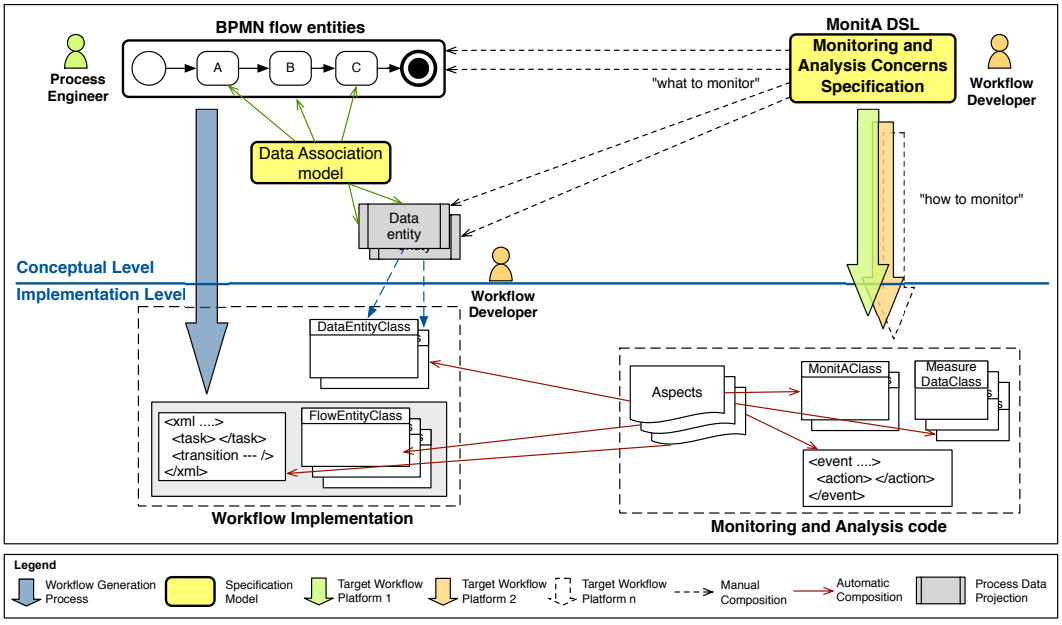


Figure 1.3: Overall Workflow Monitoring and Analysis Approach.

Higher-level Monitoring and Analysis Concerns Specification

The main element of our approach is the creation of a domain-specific language (DSL), named MonitA. MonitA eases the specification of monitoring

and analysis (M&A) concerns in terms of flow entities described in the process models (*i.e.,* at a conceptual level) and in terms of the data used by their activities. A detailed description of MonitA and the data modeling strategy is presented in chapter 4.

Workflow monitoring and analysis abstractions focus on what the monitoring and analysis concerns specify and not how they are implemented. We created a DSL to ensure that monitoring and analysis concerns can be specified in a uniform and technology-independent way. This is important since workflow developers, who implement a workflow application, should be able to write and reuse the monitoring and analysis concerns across different workflow platforms (*cf.* assessment goal AG1). MonitA allows workflow developers to specify monitoring and analysis concerns in a modularized fashion. This facilitates the localization and adaptation of monitoring and analysis concerns to ease their evolution according to the evolution of the workflow application. MonitA uses domain-specific notations from the beginning of the specification, raising the level of abstraction in the monitoring and analysis specifications (*cf.* goal G1).

To actually support monitoring and analysis specifications in terms of the data used by the workflow activities, we have defined a mechanism to model a projection of the data entities used by the workflow application and to model the new data required to measure it. The MonitA language also provides a data association model that workflow developers use to describe the workflow variables that are used by the flow entities and the operations that the flow entities perform on these variables. The data association model facilitates the monitoring of workflow relevant data being specified explicitly and captured selectively. The data association model is specified externally to the monitoring and analysis concerns model, thus, it can be reused in multiple MonitA specifications.

The data modeling strategy facilitates workflow data and measurement data to be localized, used, and shared by multiple workflow developers. This data-centric specification facilitates workflow developers to specify monitoring and analysis concerns driven by the domain the workflow is modeling (*cf.* goal G2 and assessment goal AG2). The monitoring concerns specified in terms of data entities customize the instrumentation of the workflow application to capture workflow events in terms of data entities. The custom measurements can be defined in terms of the workflow application domain and the control concerns can be defined in terms of application-specific measurements.

Generative Implementation Strategy

The second element of our approach is the definition and implementation of a strategy to integrate MonitA specifications with the implementation of workflow applications. One key characteristic in this strategy is to generate the implementation of MonitA specifications for existing workflow languages and

engines. In this way, our approach can be highly adopted and integrated with WFMS (*cf.* goal G1). A detailed description of the strategy for implementing M&A concerns is presented in chapter 7.

Our strategy assists workflow developers to enhance a given workflow technology to support the automated implementation of monitoring and analysis concerns and their composition with a workflow application. Our strategy to support the above solution is to use multiple generative programming approaches such as Model-Driven Engineering (MDE) and Aspect-Oriented Programming (AOP). We use MDE technology [Sch06] to generate the implementation of monitoring and analysis specifications in the workflow language (*e.g.,* JPDL) that implements the control flow of the workflow application and in a general-purpose language (*e.g.,* Java) for implementing the required underlying application code. The generated monitoring and analysis code is modularized in the workflow implementation through the use of AOP technology [KLM+97].

The elements in our generative strategy can be framed in three steps:

- The first step is to create a customized workflow generation process to generate a traceability model that stores the links between elements of a process model and its workflow implementation. This traceability model facilitates to infer the relation from source (BPMN) and target (workflow language) elements of the workflow application. This information is required in the model transformations to automatically determine the target of the monitoring and analysis specifications.

- The second step is to generate executable workflow code from the MonitA specifications. This executable workflow code comprises: a) aspect code to modularize the monitoring and analysis concerns implementation, b) workflow code with the monitoring and analysis concerns implementation, c) application code with a representation of measurement data, and d) application code required to manage the measurement information.

  A model transformation instruments automatically the workflow implementation with additional workflow elements (*e.g.,* events) required to support the monitoring and analysis specifications. This instrumentation is done through the information provided by the traceability model. We use AOP technology as a mechanism to keep the generated monitoring and analysis concerns modularized in the workflow implementation. We also use AOP as a mechanism to customize where to intercept workflow events in terms of flow entities as well as to intercept fine-grained workflow events in terms of data entities (*cf.* goal G2).

- Finally, the third step is to compose the generated monitoring and analysis code with the workflow implementation. We use AOP as a mechanism to

perform this composition automatically at the implementation level. The artifacts required to compose the workflow application and M&A concerns is generated according to the weaver engine provided by the aspect language. In this way, the workflow applications stay oblivious of the monitoring and analysis concerns and the existing workflow generation process can be used.

Once a MonitA generation infrastructure is created for a particular workflow platform, the specifications done by workflow developers are composed with the existing workflow implementation in an automated fashion. The resulting workflow application is instrumented with monitoring and analysis concerns and can be executed in a workflow engine (*e.g.,* Apache ODE, jBPM) that supports the targeted workflow language (*e.g.,* BPEL, JPDL).

Our generative implementation strategy facilitates workflow developers to target different target workflow platforms for implementing the workflow monitoring and analysis specifications (*cf.* assessment goal AG1). These specifications can be transformed into and composed with multiple executable workflow implementations using an automatic generation process. This makes our DSL and our overall approach reusable for a wide range of workflow platforms and applications (*cf.* assessment goal AG2).

We present the solution to our research goals in two separate parts in this document. Chapters 2, 4, and 5 present the rationale design to create our MonitA language, its concrete syntax, and its evaluation. Chapters 6, 7, and 8 present the rationale design to create the generative implementation strategy, the main elements involved in its architecture, and its evaluation.

1.5 Contributions

The main contributions of this research are:

- **An Architecture for Workflow Monitoring and Analysis**
  We present a flexible workflow monitoring and analysis architecture that serves as a common mechanism to specify and implement monitoring and analysis concerns in workflow applications. Our architecture offers the possibility a) to specify monitoring and analysis concerns independently of specific workflow platforms and in terms of the workflow relevant data, and b) to target these specifications into different workflow platforms and different workflow applications. This work has been presented in [GCD08] [GCD09a] [GCD10].

- **A Monitoring and Analysis Language**
  We created a domain-specific language named MonitA to ensure that monitoring and analysis concerns can be specified in a uniform and technology-independent way. MonitA specifications refer to BPMN process models for specifying monitoring and analysis concerns at a conceptual level. We present how workflow variables can be modeled to complement process models for supporting MonitA specifications in terms of *application-specific measurements* and *workflow data*. This work has been presented in [GCD09b] [GCD10].

- **A Generative Implementation Strategy**
  We present a generative strategy to create the infrastructure to enact the MonitA specifications on a specific workflow platform. This generative infrastructure is used to integrate automatically the MonitA specifications with the implementation of workflow applications. We also present how the generative strategy can be used to create a new MonitA infrastructure for different workflow platforms. This work has been presented in [GCD09a] [GCD10].

- **MonitA Execution Platform**
  We have developed the MonitA execution platform that implements monitoring and analysis concerns in workflow applications. The MonitA execution platform allows the definition, storage, management, and visualization of measurement information. In addition, we developed two different MonitA generative infrastructures (*i.e.,* MonitA-JPDL and MonitA-BPEL) to integrate automatically MonitA specifications with BPEL and JPDL workflow applications. The MonitA execution platform tools are available at `http://qualdev.uniandes.edu.co/bpa/` and are described in [GCD10].

1.6 Outline of the Dissertation

Figure 1.4 illustrates an overview of the structure of this dissertation. The following chapter presents a background on workflow management systems to provide the basis for our workflow monitoring and analysis approach. We present our solution in two main parts: 1) the specification, and 2) the implementation of monitoring and analysis concerns. Each one of these parts

contain three chapters which involve the background of the solution, the approach itself, and an evaluation on it. The next chapter presents the validation performed on our workflow monitoring and analysis approach. This is followed by a discussion our approach in comparison with related work. The final chapter discusses the conclusions of this dissertation.

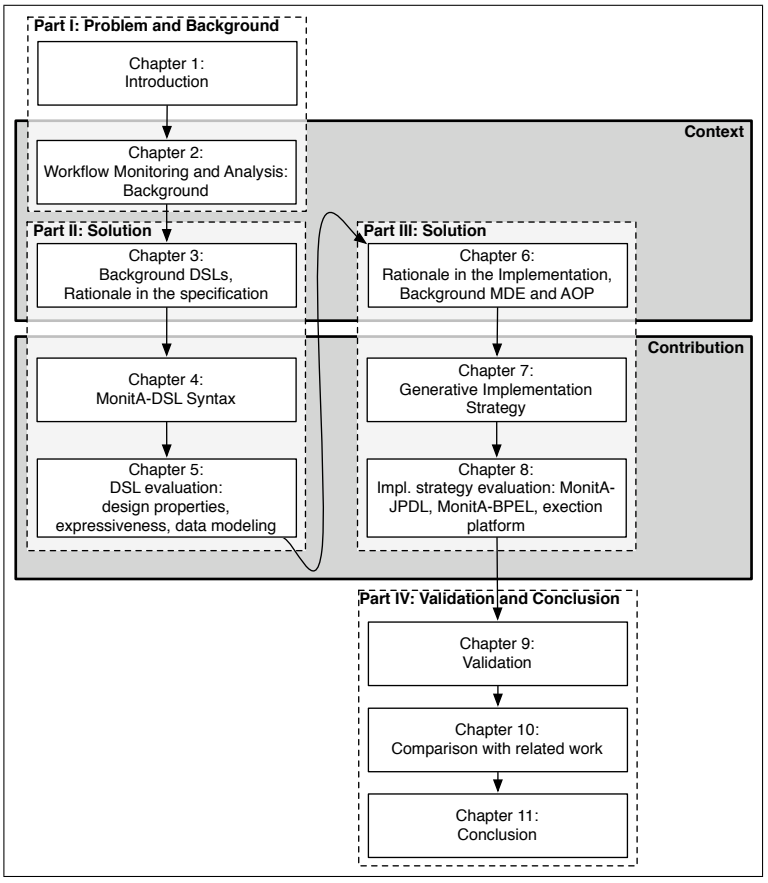


Figure 1.4: Dissertation Structure Overview

Chapter 2 details the main concepts involved in the domain of workflow monitoring and analysis. We start presenting the main elements of workflow applications as it is our application domain. We distinguish the different levels of abstraction and stakeholders involved in their development process, and how workflow monitoring and analysis fits in the life-cycle of the workflow management systems. We also present multiple definitions for workflow monitoring and analysis and discuss why having an explicit specification is beneficial for its implementation and evolution.

Chapter 3 presents the rationale and background behind the specification strategy of our workflow monitoring and analysis approach. We present the importance of a domain-specific language (DSL) tailored to an application domain, the phases for developing a DSL, and the criteria considered to evaluate

the design of a DSL. We also describes the main requirements considered to develop the MonitA DSL and the core concepts of the monitoring and analysis model. Finally, we present the design decisions adopted to develop our DSL.

Chapter 4 introduces MonitA, a domain-specific language we have developed for monitoring and analyzing workflow applications. MonitA was designed to specify monitoring and analysis concerns involving the data used by the workflow application in these specifications. We describe the approach and the characteristics for the specification of data entities used by a workflow application, and how this data specification is required to support the monitoring and analysis in terms of the workflow relevant data. We also describe the syntax of MonitA by presenting the abstractions required to monitor and analyze workflow applications during their execution.

Chapter 5 presents the studies we performed to evaluate different elements of the design and expressiveness of our domain-specific language. We also evaluate our data modeling approach in relation with workflow data patterns.

Chapter 6 presents the rationale and background behind the implementation strategy of our workflow monitoring and analysis approach. The implementation strategy is related to generate the implementation of monitoring and analysis concerns into different workflow platforms. We describe the main requirements and challenges considered to develop a MonitA generative infrastructure. We also introduce a set of definitions of Model-Driven Engineering for generation of code and separation of concerns at a conceptual level of abstraction. Finally, we introduce a set of definitions of Aspect-Oriented Programming for separation of concerns at an implementation level of abstraction.

Chapter 7 presents the generative strategy that we have defined to implement and execute the monitoring and analysis concerns specified with MonitA. This chapter also illustrates the two types of developers required to take abstract MonitA specifications to concrete implementations: application developers and MonitA infrastructure developers. We describe the process that application developers have to follow to specify and execute monitoring and analysis concerns for a workflow application. We also present the architecture and strategy that we have defined to generate and compose automatically monitoring and analysis code into an existing workflow application. We present our general strategy that can be applied to target diverse workflow languages and engines.

Chapter 8 presents the MonitA infrastructure that has been developed for implementing and executing monitoring and analysis concerns in workflow applications. We present the elements of the MonitA execution platform required to store and manage historic measurement information and to visualize this information. We also present the implementation of two different MonitA generative infrastructures (*i.e.,* MonitA-JPDL and MonitA-BPEL) created to validate our generative strategy.

Chapter 9 presents the evaluations performed using both qualitative and

quantitative criteria to determine to what extent we reached our research goals. We present the application of our approach in a set of workflow applications in order to validate its applicability for different domains. The application of MonitA in different workflow scenarios validates the goal of increasing the expressiveness of monitoring and analysis concerns in terms of workflow relevant data. We also discuss a number of case studies conducted to assess the goal of raising the level of abstraction for specifying monitoring and analysis concerns. We apply our approach to different workflow platforms to evaluate to which extent the MonitA specifications are applicable in a workflow technology independent way.

Chapter 10 presents a discussion about the main monitoring and analysis features offered by different approaches and their main missing features. We characterize current workflow monitoring and analysis approaches to determine their characteristics in analyzing workflow applications. We discuss the importance of data modeling in process models for higher-quality workflow monitoring and analysis. We present different approaches to model data on process models. We end this chapter by comparing our workflow monitoring and analysis approach against related work.

Chapter 11 summarizes the work and contributions presented in this dissertation. We discuss on the results and strengths of our research work. Finally, we discuss on limitations and future work.

Chapter 2

Background: Workflow Monitoring and Analysis

The monitoring and analysis of workflow applications can be performed in many ways. In order to position the contributions of this dissertation, we dedicate this chapter to detail the main concepts involved in the domain of workflow monitoring and analysis (M&A). We start presenting the main elements of workflow applications as it is our application domain. We also present multiple definitions for monitoring and analysis.

Section 2.1 introduces a set of definitions for the automation of business processes into workflow applications in relation with workflow management systems. We present multiple approaches typically used to model and implement workflow applications. We distinguish the different levels of abstraction and stakeholders involved in their development process. We also position how the monitoring and analysis phase fits in the life-cycle of the workflow management systems and the problems to specify M&A concerns.

Section 2.2 introduces a number of definitions for workflow monitoring and analysis. We position our approach with respect to the monitoring and analysis of custom measurements at runtime.

2.1 Workflow Management Systems

Workflow management systems (WFMS) support the effective execution (enactment) and improvement of business processes through the automated coordination of activities according to a formally defined process model [zM04]. These activities are transversal to the organizational areas (functional units) of companies. The coordination between activities is essential in every organization to adjust to changing market conditions such as, among others, the increasing market segmentation, shorter product life-cycles, non-repeating tasks fulfillment, and higher product quality. We refer to business processes

automated in a workflow management system as *workflow applications*.

Workflow applications are modeled, implemented, executed, and analyzed by workflow management systems (WFMS) through the use of software running on workflow engines [wfm99] [vdAvH04]. These four phases form a lifecycle that drives the evolution of workflow applications. A workflow application is constructed based on a process model specified by *process experts* using a process modeling notation. These process models are used to partially generate the executable workflow code into a particular workflow language. Then *workflow developers* complement this generated workflow code by defining the actual implementation of the activities, their data management, and the integration with external systems. Workflow developers create the executable workflow application by using different workflow technologies such as workflow languages and workflow engines. The resulting workflow application can be enacted by a workflow engine supporting the adopted workflow language. This workflow generation process defines the different levels of abstraction and stakeholders that have to be considered to support M&A at runtime. The specification of M&A concerns done by application developers is at the same level of abstraction than the specifications done by workflow developers.

From our point of view, the life cycle of a workflow application begins with modeling it using a high-level notation. Another perspective is to start the life cycle by the analysis phase, where event logs of existing applications are mined to extract the corresponding workflow specification (re-engineering approach).

In the next sections we describe in detail the different phases of the workflow generation process: process models specification, workflow implementation, workflow enactment, and workflow monitoring and analysis.

2.1.1 Perspectives on Workflow Applications

Curtis et al. [CKO92] present a conceptual framework with the basic building blocks of workflow applications: functional, organizational, behavioral, and informational. The functional perspective represents the flow entities (*e.g.,* activities, sub processes) which are performed during the execution of the workflow application. The organizational perspective represents where and by whom (*e.g.,* organizational unit, role, human, automatic resource) flow entities are performed. The behavioral perspective represents when (*e.g.,* sequencing) and how (*e.g.,* loops, decision criteria, decision-making conditions) flow entities are performed. The informational perspective represents the informational entities (*e.g.,* data, artifacts, products, objects) produced or manipulated by flow entities.

Typically process modeling languages allow the modeling of functional, organizational, and behavioral perspectives of workflow applications at a conceptual level of abstraction. The data involved in the informational perspective is typically specified directly at the workflow implementation level.

According to the Workflow Management Coalition (WfMC), three types of data in workflow management systems can be distinguished at runtime [wfm99]:

- *Application Data* is the data managed by external applications supporting the enactment of a workflow instance. Typically this data is not visible to the workflow management system. Examples of application data comprise documents, e-mails and database records, whose content is not relevant to the control flow of the workflow application.

- *Workflow Relevant Data*, also named process flow data, represents data used to determine the control flow of the workflow application. This data can be manipulated by the workflow application as well as by the workflow engine and can be passed between the workflow management system and related applications. Thus, workflow relevant data is made visible to elements in a workflow instance and to other workflow instances. Typed data allows the workflow management system to understand their structure and to ease their processing (*e.g.,* extract attribute values and determine a workflow participant). Untyped data can not be processed by the workflow management system but can be passed to the associated applications.

- *Workflow Control Data*, also named workflow engine state data, is the internal data managed by the WFMS system to represent the state of the workflow applications and their instances (*e.g.,* activity state changes, resource assignment). Typically this data is not visible to external applications, however, it is made persistent periodically to provide audit trail data and as a mechanism for recovery the workflow execution state after a failure. Workflow management systems typically make this information visible to external applications through application programming interfaces (APIs).

The functional, organizational, behavioral, and informational perspectives of workflow applications can be specified and implemented by using different modeling and workflow technologies. The following two sections describe these technologies, which are framed in the phases of the workflow generation process described previously: process models specification, and workflow implementation and enactment.

2.1.2 Process Models Specification

Within the process modeling phase, business *process models* can be constructed and optimized by using available business process modeling notations, such as UML activity diagrams (AD), Event-driven Process Chains (EPC), Petri Nets, Integrated Definition Method 3 (IDEF3), or Business Process Modeling Notation (BPMN). These modeling notations facilitate multiple stakeholders to align and create process models for the coordination of work that has to be

performed by multiple roles (resources and applications). A complete evaluation of these process modeling languages is presented in multiple research works [Hom04] [RRIG06] [zMI10].

We have used our approach for workflow applications specified with BPMN models since it is currently the *de facto* standard.

Business Process Modeling Notation (BPMN)

BPMN was specified by the Business Process Management Initiative (BPMI) [Whi04] with the goal of providing a standardized notation, which is easily readable and understandable by both technical and business users. A BPMN process model consists of a set of control flow constructs describing functional, organizational, and behavioral perspectives of workflow applications.

The simplicity of the BPMN notation satisfies the needs of process analysts, while its semantics satisfies the needs of the IT developers. It bridges the communication gap that frequently occurs between the design of a workflow application and its corresponding implementation. BPMN provides other important elements such as workflow patterns, and events that are powerful mechanisms to increase the expressiveness and semantics of the process models. BPMN facilitates *process experts* in the creation of process models in a conceptual level of abstraction and independently of particular workflow platforms [OMG06a]. BPMN enables interoperability and raises the level of abstraction of process description facilitating its communication and validation. Examples of BPMN process models are presented in sections 1.2, 9.2, and 9.3 which illustrate three different workflow scenarios used to motivate and validate our research.

However, there are multiple drawbacks if the objective of the organization is to automate a business process into an executable workflow application. Firstly, each workflow provider interested in using BPMN has to translate it in an ad-hoc manner since BPMN has no well-defined execution semantics [Dub04]. For instance, Oracle [Oraa] or Intalio [Int] translate BPMN into BPEL. Secondly, the data in BPMN is informally described through a graphical representation but there is no notion of type, scope or protection. Thus, BPMN leaves a gap for the definition of data manipulated by activities involved in a workflow application. This is problematic since these elements constitute a major source of information for monitoring and analysis. Thirdly, usually the automation of activities is defined in the underlying implementation language (*e.g.,* BPEL, Java) for a specific platform since BPMN does not include a language to define the actual implementation of its activities (see section 2.1.3). For example, a generated BPEL implementation only controls the workflow execution state but requires custom development to manage the data. In addition, BPMN requires advanced expressiveness mechanisms for handling complex modeling concepts [OMG06a] such as monitoring.

Now that we have presented the advantages and drawbacks of BPMN, the following presents the main elements of the BPMN metamodel that are available to create process models. Figure 2.1 illustrates the graphical elements of BPMN (version 1.1) used to model workflow applications.

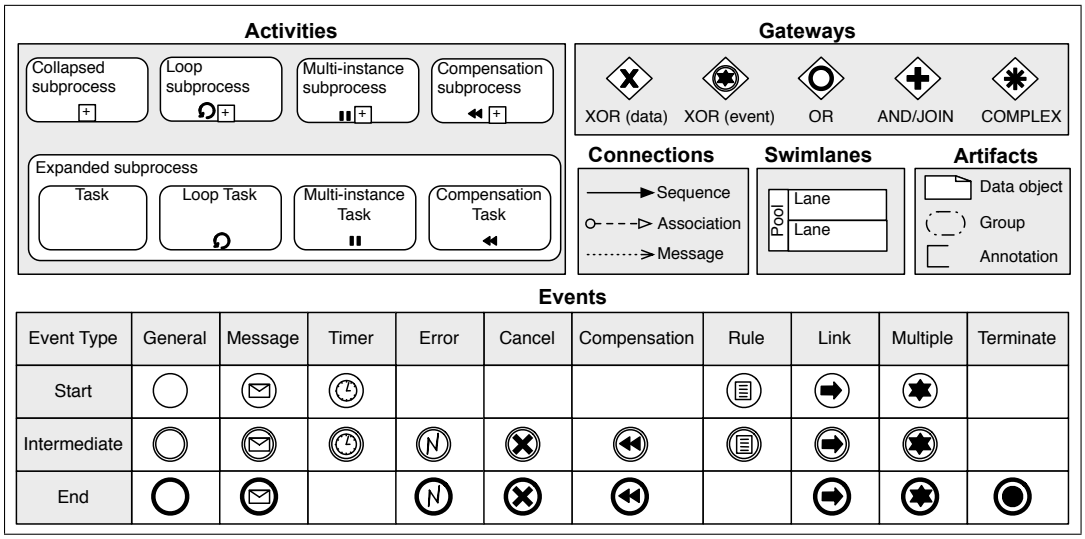


Figure 2.1: Graphical BPMN Elements

The BPMN metamodel, provided by the Eclipse Foundation [The08], comprises four different elements:

- **Flow objects** such as activities, events, and gateways define the functional and behavioral perspective of workflow applications.

  Activities are the main elements of a BPMN process model and represent particular pieces of work executed in the workflow application. The activities can be specialized in sub-processes and atomic tasks, which specify the workflow application at different levels of granularity. A `sub-process` represents an activity that is decomposed into other activities, whereas an atomic `task` is not decomposed into more specialized tasks. Atomic tasks and sub-processes can be specialized by adding specific attributes such as loop, multi-instance, or compensation. The loop attribute models cycles in a workflow application avoiding the use of gateways and connections, which are explained later on. The multi-instance attribute constraints the number of instances of an activity than can be executed. Finally, the compensation attribute indicates that an activity will be executed when a cancellation event is triggered by another activity [OMG06a].

  Events describe an interaction that happens in the execution of the workflow application such as the initialization or finalization of an activity or the timeout of an operation. Events are categorized into start, intermediate, and

end events. Start events trigger the execution of a workflow application by one of the following types: independently of particular events (`none`), based on the finalization of other workflow applications (`link`), as a result of the arrival of a message (`message`), dependent on multiple options (`multiple`), based on the evaluation of a condition (`rule`), or at a specific date and time (`time`). Intermediate events represent any event that may happen during the execution of the workflow application. Intermediate events such as message, timer, link, multiple, and rule are similar to the start event. In addition, a none event represents any event between the initialization and finalization of a workflow application, an `error` event receives a notification or notifies about the presence of an error, a `cancel` event indicates the cancellation of a transaction within an activity, and a `compensation` event initiates a compensation handling activity. In contrast to intermediate events, end events are generated at the finalization of the workflow application [OMG06a].

Gateways control the execution flow of a workflow application by defining how to diverge or converge points in the workflow when a decision needs to be made. Gateways are classified as exclusive, inclusive, parallel, or complex. Exclusive gateways (`XOR`) model points in the process when a selection of a single execution flow needs to be made. This selection is based on the evaluation of a condition associated with each option (data-based gateways) or on an event occurring at the execution time (event-based gateways). Inclusive gateways (`OR`) represent points in the process in which condition expressions are evaluated to take one or more alternatives. Parallel gateways (`AND`) represent diverging points where several activities can be executed concurrently or converging points where the concurrent activities need to be synchronized to follow the execution flow. `Complex` gateways model special situations that have to be considered in order to continue with the workflow execution [OMG06a].

- **Connecting objects** represent the connection between flow objects. Connecting objects are classified into three types: sequence flow, message flow, and associations. `Sequence` flows define the order in the execution of the activities defined in a workflow application. `Message` flows model the interchange of messages between workflow applications. `Association` flows associate flow objects with artifacts.

- **Swimlanes** define the organizational perspective of workflow applications. Swimlanes model private workflow applications or inter-organizational workflow applications in which multiple workflows participate in a collaborative manner [OMG06a]. Swimlanes are specialized into pools and lanes. A `pool` represents a participant as a workflow application, whereas a `lane` is a sub-partition within a pool that organizes and categorizes activities executed by different roles in the same workflow application.

- **Artifacts** define the informational perspective of workflow applications. BPMN distinguishes between three types of artifacts: `data objects`, text `annotations`, and `groups`. These artifacts provide information about the products used or produced by activities, but they do not have any direct effect on the execution of the workflow application [OMG06a].

Most of these BPMN elements can be found in the process model of the trouble ticket scenario (see Figure 1.1 in section 1.2).

2.1.3 Workflow Implementation and Enactment

Process models are implemented into executable workflow applications using a workflow language (*e.g.,* XPDL [XPD], BPEL [IBM02], JPDL [JPD], XPM [SJVD09]). The resulting workflow definition, that is obtained from a process model, is complemented with corresponding underlying application code (*e.g.,* Java, WSDL). There are different types of workflow languages: graph-based, script-based (workflow programming languages), state and activity charts, and petri net-based [WV99].

A workflow application consists of a set of constructs describing its perspectives namely control, data, and resources, which are required to execute an activity. The specific semantics of executing an activity can be a) the automatic call of a service implemented by some application, b) something a human being has to do manually, or c) something a human has to do using an external application.

A *workflow implementation* can be partially generated from a process model. For example, automatically creating a BPEL description using the translation described in [OvdADH]. Note that this transformation into the executable language is only partial which means that developers still need to add elements afterwards (*e.g.,* the particular class to be called if control is handed over to an activity). This is due to the fact that the process modeling standard (*i.e.,* BPMN) is not expressive enough to contain all these details.

The executable workflow applications are deployed into a workflow engine (*e.g.,* Apache ODE, jBPM, Cumbia), which enacts instances of these workflow applications. Workflow executions are stored by the workflow engine store system, which refers to it by a *workflow instance*. Despite the wide range of workflow languages developed so far, we present two well known workflow languages that were used to validate our approach.

Business Process Execution Language (BPEL)

The *Business Process Execution Language* for web services (BPEL) [IBM02] provides means to formally specify workflow applications and interaction protocols. BPEL is an XML programming language for specifying workflow applications behavior based on web services. BPEL applications export and import

functionality by using web service interfaces exclusively. A *workflow definition* in BPEL comprises a control flow diagram of activities where their execution can invoke operations of other services (underlying application code) and receive messages from external sources.

A BPEL specification defines the name of the workflow application and the name spaces used in its definition. A BPEL specification contains three main blocks: partner links, variables and activities.

Partner Links refer to the web services and other BPEL specifications that are involved in the orchestration defined in the workflow application. The main attributes of a partner link are: a name that identifies the participating service, a partnerLinkType that specifies the service offered, a processRole that describes the service implemented, and a partnerRole that defines the web service called by the workflow application.

Variables store the state of messages that are sent and received between partners. The variables include messages passed from/to BPEL applications, messages interchanged with external services, and local variables used in the control flow of the workflow application. Each data block is an XML document where the *type* of a variable can be a web service description language (WSDL) message type, a simple type of an XML Schema, or an XML Schema element.

Activities contain the actions to be performed in a workflow definition. There are two types of activities: structural and behavioral. *Structural activities* define the orchestration logic in the workflow application. The execution of a BPEL application can be defined by using a set of primitives such as a) a *sequence* for defining an execution order, b) a *switch* for conditional routing, c) a *while* for looping, d) a *pick* for executing conditions based on timing or external triggers, e) a *flow* for parallel routing, and f) a *scope* for grouping activities [IBM02]. *Behavioral activities* are classified into: receive, assign, invoke, reply. A receive activity executes workflow operations when it receives a message from a client (*e.g.,* create a new instance of the workflow application). An assign activity assigns the content of a variable into another one. These data can be manipulated and transformed by multiple languages such as XPath, XQuery, XSLT, or Java. An invoke activity enables a service invocation (*i.e.,* synchronous, asynchronous) and the operation it has to perform. A reply activity sends a message as a response to a received message through a receive activity.

A BPEL specification offers a clear execution semantics, and well defined storage and interchange formats. There are several products that support the definition and execution of BPEL applications [Orab] [Act]. Each workflow product provides a workflow engine, which supports passing messages between partners and the usage of events.

jBPM Process Definition Language (JPDL)

The *jBPM Process Definition Language* (JPDL) [JPD] is a graph-based language (XML specification) used to specify workflow applications. A JPDL implementation consists of multiple artifacts, which describe how workflow entities must interact (*workflow definition*) and the application code implementing these entities (*underlying implementation*). An example of the JPDL implementation of the trouble ticket scenario is illustrated in Figure 1.1 in section 1.2).

A *workflow definition* describes the control flow of workflow entities by mean of *nodes* and *transitions*. A token represents one path of execution that maintains a pointer to a node in the graph. The graph uses *actions* as an underlying implementation mechanism to add technical details outside of the workflow definition.

Each node has a type to determine the control flow in the workflow execution. There are five types of nodes: a) a *task node* represents a set of tasks performed by humans, b) a *state node* represents when a process waits for an external system execution (asynchronous communication), c) a *decision node* specifies a condition on the transitions, d) a *fork node* splits the execution path into multiple ones, and e) a *join node* merges all tokens created by the same parent (fork node). A transition relates a source node with a destination node.

Actions are pieces of Java code that are executed upon events in the workflow execution [JPD]. Actions include predefined or custom Java code for the execution of the workflow through delegation. The Java code is associated with the graph without changing its structure. Actions are executed upon workflow events in the process execution such as entering a node, leaving a node and taking a transition. Events are the hooks for actions. Actions located on events do not influence the control flow of the workflow application since it works as an observer pattern [GHJV95]. In contrast, actions located on a node propagate the workflow execution [JPD].

In addition to nodes, transitions and actions, a JPDL workflow definition supports other artifacts such as superstates to group nodes, exception handling managed by Java, and workflow composition.

A JPDL workflow implementation is executed in the jBPM engine [jBP], which fires events (*e.g.,* node-enter) during the graph execution. Events have a list of actions that must be executed when the jBPM engine fires an event.

2.1.4 Workflow Monitoring and Analysis

Despite the role of workflow management systems on workflow improvement, these systems are complemented with M&A tools and techniques to identify problems occurring in workflow applications. These monitoring and analysis techniques provide workflow management systems with the ability to continu-

ously inform and apply business process optimizations.

The complexity for implementing M&A concerns depends on the complexity of the workflow implementations. The evaluation of M&A solutions differs by the capabilities of the workflow products and by the complexity of the workflow applications depending the application domain (*e.g.,* banking & investment, healthcare, insurance). There is not a complete benchmark for evaluating M&A solutions in WFMS since there are many workflow products without a consensus to describe executable business processes. Nevertheless, the Workflow Patterns Initiative [vdAtHKB03] identifies workflow modelling scenarios and solutions, and provides evaluations for benchmarking various workflow products (commercial, open source, and proposed standards).

The workflow management systems and their associated M&A solutions have been highly adopted in companies with different domains [Kas06] [Pal09]. For example, healthcare processes (*e.g.,* organizational, medical treatment) have a wide range of offerings, involve multiple distributed organizational areas, involve complex set of possible diagnostic paths, and need to be highly structured. In this domain there is high demand for monitoring since healthcare providers require effectiveness, patient safety, high quality care, and extent of care. Hofstede et al. [tHvdAAR10] introduces a workflow application automated in YAWL for managing a gynecological oncology process. This process involved 230 tasks dependent on choice, and different perspectives (*i.e.,* control, data, resources).

The next section presents the scope of monitoring and analyzing workflow applications.

2.2 Monitoring and Analysis of Workflow Applications

The first wave of workflow management systems was focussed in re-engineering the software systems into process-oriented organizational structures. In this wave, the emphasis was on constructing process models and analyzing them. Current WFMS have moved up to a second wave in which business processes are executable to offer better control in the organizations. A third wave in WFMS is monitoring and analyzing the execution of business processes to focus on a continuous improvement process according to the business evolution.

The monitoring of workflow activities is required for auditing the operational efficiency of an organization. This fact is reflected by multiple emerging terms related to the monitoring of business activities such as BAM (Business Activity Monitoring), BOM (Business Operations Management), BPI (Business Process Intelligence), and Business Process Analysis (BPA). The importance of measurement and control has been demonstrated by multiple tools

and publications related to this topic [MHH07] [zM00]. The existing solutions are presented in chapter 10 where we present a comparison with our approach.

The following sections describe the main concepts related to the monitoring and analysis of workflow applications.

2.2.1 Workflow Monitoring and Analysis Taxonomy

Workflow monitoring and analysis (M&A) comprises multiple application fields such as verification or validation of process models (conformance checking), automatic generation of improved process models, predictions, and measurements. In addition, there are multiple techniques for monitoring and analysis such as analysis done *a posteriori* (*e.g.,* process mining), and workflow monitoring at runtime (*e.g.,* BAMs). In that context, the focus of our research is on the monitoring and analysis of workflow applications at runtime to measure and evaluate their execution. The M&A at runtime enables the assessment of changes in the quality of workflow applications. This is done through strategic measurements incorporated in the workflow applications to be analyzed during their execution to provide feedback for workflow analysts. The M&A at runtime decreases the time required to identify problematic aspects about critical properties in the workflow and to decide on workflow improvements (*e.g.,* re-assign resources, add activities).

We consider that an accurate identification of problematic aspects at runtime in workflow applications can be supported by implementing M&A concerns such as: monitoring, measurement and control. These M&A concerns are defined for monitoring the actual workflow executions, building and managing measurement information based on the data manipulated by the workflow application, and applying notification actions based on the evaluation of those data. The measurement data must be stored to support further analysis by accessing historical information.

Workflow M&A is a complex domain that requires a mapping between process models and workflow implementations. It involves multiple artifacts such as configuration files, descriptors, workflow code, and application code. In addition, the extraction of data from the internals of the workflow application for workflow analysis purposes is non-trivial since it involves information from multiple external systems. The M&A domain comprises multiple fields and meanings, thus its scope must be well defined.

The authors in [zMR00] outline some monitoring facilities provided by workflow management systems. We took this information as a starting point to create a customized taxonomy of types of M&A activities. Figure 2.2 illustrates this customized taxonomy and shows different associated application examples.

Broadly speaking there are two main perspectives:

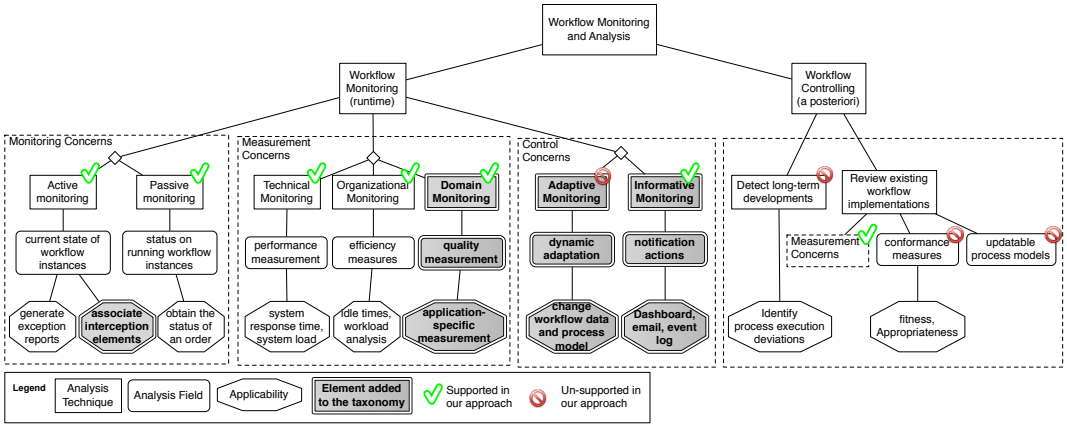


Figure 2.2: Taxonomy for Workflow M&A Adpated from [zMR00].

- **Workflow monitoring**, also named operative (interactive) process controlling, deals with the monitoring and analysis of workflow instances at runtime. The analysis in this technique can be characterized as active or passive, as technical, organizational or application-specific, and as adaptive or evaluative.

  Active/Passive Monitoring. Active monitoring provides information of the current state of workflow instances. This technique is useful for the generation of reports (*e.g.,* exception, warnings) with the actual and potentially overdue tasks (work items). We also consider this technique useful for associating interception elements that can be captured by external systems. Passive monitoring provides status information about running workflow instances upon request. This technique can be used to obtain the status of an order inquired by a customer and to visualize it (*e.g.,* dashboard).

  Technical/Organizational/Application-specific Monitoring. Technical monitoring is used for *performance* measurement such as system response time, and system load. Organizational monitoring measures the *efficiency* of the workflow application (*e.g.,* idle times, workload analysis). Application-specific monitoring measures the *quality* of the workflow application (*e.g.,* workflow data analysis).

  Adaptive/Informative Monitoring. Adaptive monitoring is used for dynamic adaptation of the workflow application (*e.g.,* modify the process model, change values of workflow variables). Informative monitoring is used to evaluate the historical data of workflow execution instances and to notify external entities of particular behaviors for identifying potential improvements (*e.g.,* create event log).

- **Workflow controlling**, also named strategic process controlling, aims at a posteriori (ex-post) analysis of the logged audit trail data captured during

the workflow enactment. It is useful a) to detect long-term developments in workflow enactment, b) to review already existing workflow implementations, and c) to identify deviations in the workflow execution by comparing audit trail data to target data derived from corresponding process models.

We limit the scope of our approach to the elements in the taxonomy that are market with the "√" symbol. In general, we focus on monitoring and analyzing workflow applications at runtime.

2.2.2 Workflow Monitoring and Analysis Views

Single workflow instances and the aggregation of them can be evaluated according to different views to analyze the historical data of workflow instance execution. Muehlen et al. [zMR00] identify three different but related views required to reduce the complexity for the analysis of workflow applications. These views comprise a) processes and functions, b) resources, and c) process objects (data entities).

Process view. The process view evaluates a) all performance measurements related to optimize the workflow application, and b) the conformance between process models and workflow instances information stored in event logs. The information gathered in this view can be used to change the elements associated to the different perspectives of the process models.

Performance measurements correspond to key performance indicators concerning time, cost and quality. The following are different evaluation examples in this view: a) average, maximum, and minimum process time, b) average, maximum, and minimum execution costs of a process, c) the number of failures or loops, which evaluates the quality of a process by establishing necessary rework.

Conformance measurements compare the process models and the workflow instances information. The workflow instances are compared only if they have the same execution path. For example, a workflow variant during the workflow execution has to be identified. Thus, the processing time can be predicted in an early stage of the workflow execution where the workflow variant is not yet identified.

Process objects are associated with a state in a given time. These state changes correspond to event types in the workflow management system (*e.g.,* when a process instance moves to a state *completed*).

Resource view. The resource view evaluates the usage of the available resources (*e.g.,* personnel). The evaluations in this view also correspond to criteria such as time, cost, and quality. The costs of the involved resources in a workflow application can be derived from applications associated to the

workflow. The time dimension evaluates the availability of the resources.

Object view. A business relevant object (*e.g.,* ticket, loan, order, inquire, invoice) is processed by the logical sequence of functions (activities) that conform a workflow application at the instance level. This view facilitates the identification of the value drivers of the workflow applications. A cost criteria is related to the evaluation of costs to handle an object (*e.g.,* calculation of costs and revenues). The time criteria informs about the typical processing time for these objects. Quality criteria measurements can inform about potential problems related with an object. Basic evaluations inform about the progress of a specific object.

We consider process and object views in our approach by evaluating time and quality criteria. A business object can be associated with multiple workflow elements. Thus, we consider that this view must expose the workflow variables associated with the workflow elements. This enhances the capabilities to monitor and analyze workflow applications, and to provide higher quality measurements (*e.g.,* in a loan process, the estimation of rejected requests performed by a specific user). It must also provide support for the interception of fine-grained state changes in the activities and workflow variables.

2.2.3 Workflow Monitoring and Analysis Dimensions

The most common requirement for workflow analysis is measuring performance related to processes and activities in terms of generic measurements. *Generic measurements* are predefined and can be computed for multiple workflow applications independently of their application domain (*e.g.,* the time a workflow instance is running, the number of instances of a workflow application, the utilization of workflow resources, the current workflow execution state). A generic dimension for monitoring and analysis comprises measurements within a process view and an activity view. The process view evaluates all performance measurements related to optimize the workflow application.

Although such generic measurements define useful analysis information, they do not provide quality measurements specific to the application domain. Therefore we emphasize another analysis dimension named *application-specific measurement*, which is needed to create custom quality measurements required to analyze the workflow applications. Application-specific measurements capture information that is specific to the particular domain of the workflow application (*e.g.,* loans rejected of a specific client in a banking context). An application-specific monitoring and analysis dimension comprises measurements within a data view combining generic measurements with workflow data entities (*e.g.,* loan, client).

Both dimensions for monitoring and analysis need to be considered to support the strategic goals required for workflow improvement.

2.2.4 Workflow Monitoring and Analysis Technologies

Muehlen et al. [zMR00] also discuss evaluation methods and information provided by workflow technologies to support monitoring and analysis.

The first consideration is about the *available information* for M&A. This information is provided by the selected workflow management system in different ways. For example, some workflow systems record system events associated with timestamps, whereas others also include the object processed within the activity. The available information defines the quality and scope of the analysis that can be performed.

Typically, the information that is recorded by a workflow management system corresponds to a) state-changes in processes and activities, b) resources involved in these state-changes, and c) timestamps of state-changes. Evaluating this information from the audit trail logs is useful for diverse types of analyses (*e.g.,* activity processing times) and predictions (*e.g.,* potentially overdue activities). All this measurement information is very useful to analyze the operative effects of workflow executions, however, other advanced measurements related to process quality (*e.g.,* effects of workflow in time to market) are required to analyze strategic effects in workflow application execution.

The second consideration is about the *evaluation techniques* for M&A. Statistical techniques are used to evaluate a measurement (*e.g.,* processing time) or a set of values against defined upper and lower levels of tolerance. Another technique is the use of a model to identify who is actually performing what type of work. Nevertheless, the latter kind of analysis is not considered in our workflow monitoring and analysis approach.

The monitoring of workflow applications can be performed by multiple solutions:

- *Instrumenting the workflow implementations.* The specification of the workflow application is enriched with code to capture particular events and with new elements (*e.g.,* activities) to send audit information to external services.

- *Tracking control flow events (state changes) managed by the WFMS.* Multiple WFMSs use observer plug-ins to log state changes to access directly the audit trail data. Then the observer plug-in is automatically invoked by the WFMS when a workflow event (*e.g.,* activity started) is performed.

- *Tracking state changes not managed by the WFMS.* These state changes have to be captured in the systems supporting the underlying workflow implementation (*e.g.,* application code, database tables). Then these systems can generate audit trail data when a state change (*e.g.,* a record is inserted) occurs.

- *Querying audit trail data.* Multiple workflow analysis approaches provide mechanisms to gather information about all status changes of a workflow

application execution, which is contained in an audit trail. Typically, this information is gathered according to the time-sequenced record.

- *Interception of web service requests.* When using a service-oriented architecture, a web service gateway can be used to intercept web service requests and to extract the required auditing data.

The Workflow Management Coalition (WfMC) specifies through the *Interface 5* (administration and monitoring tools) the essential information about workflow instances that a workflow management system must record [WfM98]. This historical information correspond to state changes of a workflow instance from start to completion or termination. For example, historical information such as date, time, and type of work performed per state changes may be collected from log records. The log records must provide a minimum of information to support workflow monitoring and analysis: unique object identifier that is processed by a workflow instance, workflow instance that is being enacted, workflow application that is being enacted, resource that performed a transition, source and target of the transition, and timestamp and time zone of the transition. Additional information stored in the log records (*e.g.,* customer, product) are dependent of the domain the workflow application is modeling. The WfMC standard provides the data format and guidelines to record workflow events, however, it gives no advice on how to evaluate this measurement information.

2.3 Summary

This chapter has explained the main concepts involved in the domain of workflow monitoring and analysis. We presented the different levels of abstraction and stakeholders involved in the development process of workflow applications. We identified how the lack of data modeling in BPMN process models is problematic for specifying monitoring and analysis concerns to be analyzed at runtime. The workflow monitoring and analysis phase was motivated as part of the life-cycle of the workflow management systems. Multiple definitions for workflow monitoring and analysis were presented to position our approach with respect to the monitoring and analysis of custom measurements at runtime. We have discussed the necessity to introduce application-specific measurement as an analysis dimension for workflow applications.

The following chapter presents the decisions we made to tackle the problems identified for monitoring and analyzing workflow applications.

Part II

Specifying Monitoring and Analysis Concerns in Workflow Applications

Chapter 3

Rationale and Background

We dedicate this chapter to present the rationale and background behind the specification strategy of our workflow monitoring and analysis approach (see section 1.4). As detailed in the introduction, the main problems to specify monitoring and analysis (M&A) concerns in workflow applications are related to a) high development costs, b) tight coupling with specific workflow platforms, c) limited expressiveness, and d) complex maintainability. This is because workflow developers need to build ad hoc infrastructures into a particular workflow platform to specify and implement M&A concerns. The main decision we made to tackle these problems is to separate the solution in two main parts: 1) the specification, and 2) the implementation of monitoring and analysis concerns.

First, the main contribution of our approach is the creation of a domain-specific language (DSL) named MonitA to ensure that M&A concerns can be specified in a uniform and technology-independent way. Moreover, the rationale to build a DSL was influenced by the needs to trade generality for expressiveness in the monitoring and analysis domain, to ease the specification in this particular application domain, to reduce the software maintenance costs, and to open up this application domain to a larger group of application developers. The MonitA DSL was conceived for raising the level of abstraction to workflow developers for specifying M&A concerns (MonitA specification).

Second, the other contribution of our approach is the definition of a generative strategy to create the infrastructure to enact the MonitA specifications on a specific workflow platform. This generative infrastructure is required to integrate automatically the MonitA specifications with the implementation of workflow applications. The decision to define this strategy is influenced by the need to reuse MonitA specifications across different and existing workflow languages and engines. The rationale to define the generative implementation strategy is presented in chapter 6.

The investment in the development MonitA and of a particular generative infrastructure for it is recovered by lower development and maintenance costs.

The main elements involved in the MonitA execution platform are categorized in three parts: specification, implementation, and enactment. Figure 3.1 illustrates the main parts and stakeholders of the MonitA execution platform.

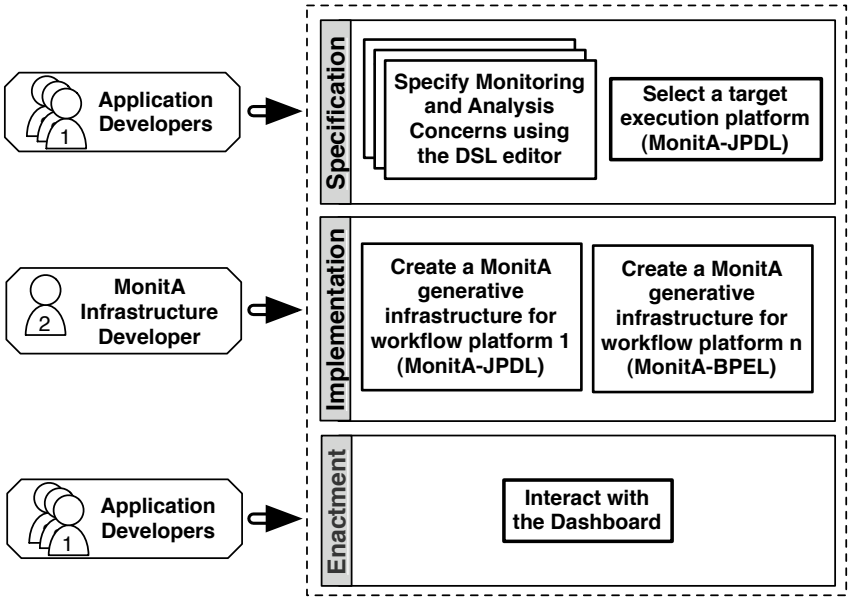


Figure 3.1: Overall View for the MonitA Execution Platform.

There are two different types of developers required to take abstract MonitA specifications to concrete implementations: application developers and MonitA infrastructure developers.

The *MonitA infrastructure developers* are involved in the implementation part by creating a generative infrastructure to automate the implementation of MonitA specifications for a particular workflow platform. The elements of the architecture defined for the implementation part of the MonitA execution platform are presented in chapter 7.

The *MonitA application developers* use the MonitA DSL to specify M&A concerns for a workflow application, and select a target workflow platform to integrate automatically these concerns with the workflow application. The different elements required for MonitA specifications are presented in chapter 4. The application developers also interact with the enactment part in the MonitA execution platform to analyze the monitoring information at runtime. The different elements related to the execution part of the MonitA execution platform are presented in section 8.4.

The following sections present the rationale to create the MonitA DSL.

Section 3.1 presents the importance of a DSL tailored to an application domain, the phases for developing a DSL, and the criteria considered to evaluate

the design of a DSL. The design criteria are retaken in the validation chapter to evaluate the design of our DSL (see Section 5.1).

Section 3.2 describes the main requirements considered to develop the MonitA DSL and the core concepts for a MonitA specification.

Section 3.3 presents the design decisions adopted to develop our DSL.

3.1 Domain-Specific Languages

A domain-specific language (DSL) is a language designed to provide a specific notation tailored to an application domain. These domain-specific notations and constructs offer end-users gains in expressiveness, ease of use, productivity, and maintenance costs.

DSLs can have multiple degrees of executability such as a) a DSL with well-defined execution semantics (*e.g.,* Excel spread-sheets), b) a DSL acting as input language for an application generator (*e.g.,* MonitA), c) a non-executable DSL useful for application generation (*e.g.,* BNF), and d) a non-executable DSL (*e.g.,* data structure representations).

3.1.1 Development Process

Mernik et al. [MHS05] presents a survey with the phases of a DSL development and some patterns that can be identified in each phase. These development phases of the DSL correspond to DSL decision, domain analysis, DSL design, and DSL implementation. This study gives DSL developers a methodology regarding when and how to develop a DSL from a qualitative validation perspective.

The development of a DSL is a hard and time consuming task since the DSL developer requires domain knowledge and language development expertise.This development effort can be reduced by using a language development system (*e.g.,* oAW, Spint, Stratego). These systems typically generate tools from languages specifications such as consistency checkers and interpreters, syntax editors, prettyprinters, analyzers, interpreter and compiler generators, and a debugger. Although these systems provide support for the design and implementation phases, typically there is no support in the analysis phase. The development phases are described below.

DSL Decision

The main concerns to decide on a DSL are a) to obtain improved software development and maintenance tasks, and b) to enable users with less domain and programming expertise to enable these tasks [Nar93] [FGY+04]. Comparing the advantages and disadvantages of creating a DSL aid in the decision process.

An advantage of using a DSL is that the solutions are expressed at the same level of abstraction as the problem domain. Thus stakeholders can achieve the understanding, validation and modification of these solutions [MHS05]. A special DSL can be obtained by combining a GPL with an application library. However, the benefits of DSLs are associated with their domain-specificity, which is represented by domain-specific notations from the beginning of a specification. Typically these domain-specific notations cannot be directly mapped in artifacts allocated in a library. Therefore, the domain-specific constructs allow performing specialized actions such as verification, optimization, and transformation over a DSL.

Kieburtz et al. [KMB+96] consider as an advantage of DSLs that they can increase flexibility, productivity and reliability in software systems. A DSL can lead to automatic code generation, which reduces the time for programming repetitive tasks [JB88] and the complexity required to program new applications by using a general programming language (GPL). Spinellis [Spi01] also describes reliability as an advantage since the correctness of generators can be easily verified due to the small domain and limited possibilities of a DSL. Other works [LR94] [Kru92] consider reusability as an advantage of DSLs.

There are multiple factors that can complicate the decision to develop a DSL. A disadvantage of a DSL is the cost of their development, which requires domain and language development expertise [KLBM08]. Typically, the effort to consolidate a DSL in a large community is expensive in terms of time and become complex since it is necessary to consider important issues such as training material, language support, standardization, and maintenance. Moreover, it is necessary to consider existing software developments using a GPL to be sure that concepts developing a DSL might be useful. The authors in [vDK98] mention as a disadvantage the considerable cost for extending a DSL to support unanticipated changes. The authors in [BBH+94] and [SG97] consider as a disadvantage the lack of knowledge of how to fit the DSL into a regular development process.

Domain Analysis

In this phase, the problem domain is identified and the domain knowledge is gathered. The inputs for domain analysis can be, among others, technical documents, knowledge provided by domain experts, and existing code. The output of the domain analysis phase is the domain-specific terminology, semantics, and scope in an abstract way. This development phase contains a set of associated patterns to aid in the domain analysis:

- *Informal* pattern. The domain analysis is performed in an informal way, thereby without a specific methodology.

- *Formal* pattern. The domain analysis can be performed by using a method-

ology such as feature-oriented domain analysis (FODA) [KCH+90], domain analysis and reuse environment (DARE) [FDF98], family-oriented abstractions, specification, and translation (FAST) [WL99], and ontology-based domain engineering (ODE) [dAFGD02].

- *Extract from code* pattern. In this scenario, the domain concepts are extracted from legacy general-programming language code by inspection or by using mining tools.

The variabilities on terminology and concepts obtained in the domain analysis are used to guide the development of the actual DSL constructs. The commonalities of domain concepts are used to define the set of common operations required in the execution model and the primitives of the language.

DSL Design

The approaches to design a DSL can be grouped in two dimensions: the relationship between the DSL and existing languages, and the nature of the design description. The approaches for both are framed in the following patterns:

- *Language exploitation* pattern. The design of a DSL can be based on an existing language to ease its implementation and to maintain familiarity for users. The DSL design can follow a *piggyback* approach to use parts of an existing language, a *specialization* approach to restrict the existing language into a particular domain, or an *extension* approach to add new features to an existing language to address the required domain concepts. The first two approaches are typically used when the notation is widely known.

- *Language invention* pattern. In this pattern, a DSL is designed from scratch with no commonalities with an existing language. Consequently, the design can be difficult to characterize.

- *Informal design* pattern. In this design, the specification of the DSL is done typically in a natural and illustrative language. The development of an informal DSL specification cannot be validated before the DSL is actually implemented.

- *Formal design* pattern. A formal design is specified using an existing semantics definition method. These methods include regular expressions and grammars for syntax specifications, and rewrite systems, abstract state machines, and attribute grammars for semantic specifications [SK95]. The development of formal DSL specifications can be implemented automatically by language development systems, which reduces the implementation effort.

DSL Implementation Approaches

There are multiple approaches to implement a DSL such as interpretation, compilation (application generator), preprocessing (*e.g.,* macros, template metaprogramming in C++), embedding (*e.g.,* subroutines), commercial Off-The-Shelf (COTS) and extensible compilation/interpretation [KLBM08]:

- *Preprocessing.* In this approach, DSL constructs are translated into constructs in the base language. The preprocessing approach performs compile-time generation of language-specific code achieving good performance. Nevertheless, the generated code is limited to a static analysis provided by the base language processor. Thus it is not possible to make optimizations at the domain level and the error reporting is done in terms of base language concepts.

- *Embedding.* This approach consists of defining domain-specific abstract data types and operators by extending an existing general-purpose language (GPL). The new language reuses all the host language power and the compiler or interpreter of the host language is reused as is. However, the DSL user has to become a programmer, it provides low expressiveness, and domain-specific optimizations and transformations are hard to achieve. Error messages are presented in terms of the host language concepts and not in terms of domain-specific concepts.

- *Compiler/Interpreter.* Building a compiler/interpreter from scratch is costly. However, this approach offers a domain-specific analysis on the DSL programs, a closer syntax to the notation used by domain experts, and good error reporting. The compiler can be implemented by using compiler writing tools to minimize the implementation effort.

- *Extensible compiler/interpreter.* A GPL compiler/interpreter can be extended to provide domain-specific optimization rules or code generation. Nevertheless, extending a compiler is hard and requires to be extremely careful to avoid any interference of domain-specific notations with existing ones.

- *Commercial Off-The-Shelf (COTS).* A COTS approach builds a DSL based on existing notations (*e.g.,* XML), thereby this approach depends on domain rules to apply (extract) the existing functionality.

A detailed study about multiple approaches that can be used to implement DSLs is presented in [KLBM08] [MHS05]. Mernik et al. [MHS05] presents an analysis from a qualitative point of view. Kosar et al. [KLBM08] presents some partial quantitative results evaluating some DSL implementation approaches. This study supports the claim that an embedding approach is the most appropriate in terms of implementation effort. Nevertheless, the authors claim that

the end-user effort required to write rapidly correct programs can be, depending of the DSL, more important than the effort required to implement the DSL. Therefore, when big groups of users are going to use a DSL and the notation must be strictly adopted, then the recommended solution is to implement the DSL using compiler generators.

Consel et al. [CLRC05] presents a methodology to develop DSL compilers to translate the logic of a program into a GPL representation. This methodology relies on generative programming tools such as AOP, annotations, and program specialization to define a modular compilation of DSL programs. A compiler developer chooses the most appropriate generative programming approach in the program generation process.

3.1.2 Design Principles

The life expectancy of a DSL can be damaged by two threats for DSL degradation: domain absorption degradation and general-purpose absorption degradation [Cle10]. Domain absorption degradation is present when the DSL breaks the borders of its initial domain resulting in inconsistent syntax and semantics, and in restrictive and complex specifications. General-purpose absorption degradation is present when the DSL abstractions grow towards general-purpose language concepts that are unfamiliar to DSL users and that can be already offered by programming languages (*e.g.,* libraries). In this case, the creation and maintenance of a DSL introduces an overhead in the software development process.

A set of design principles is described in [GC10]. The evaluation of these design principles for a DSL facilitates the identification of its distinguishable properties and avoids to get an ambiguous and useless DSL. We summarize the main principles which we will use in chapter 5 to evaluate our DSL:

Representation

Representation deals with the syntax for specifying the DSL concepts. A DSL with familiar syntax finds broad adoption and improves the rate at which the language evolves [Cro08]. A DSL representation involves a concrete syntax that enables DSL users to write unambiguous sentences and a code structure to arrange the different grammatical sentences. The DSL syntactic representation must adopt the most optimal syntax for DSL users. In order to validate this principle, the following properties have to be evaluated:

- *Domain syntax.* The DSL is designed based on syntactic representations common to the domain.
- *Distinguishable syntax.* The parts or blocks of a DSL specification can be easily identified.

- *Familiar syntax.* The DSL adopts an existing, related, and well established syntax.

An optimal syntax representation improves *readability* by facilitating the recognition of relevant parts of a DSL specification. An optimal representation also improves the *conciseness* of a DSL specification making it more readable. This is important since most of the time spent in the development process goes to looking at the specification rather than actually writing it. An adequate syntactic representation contributes to the *consistency* ("uniformity") depending on whether a set of constructs is indivisible or divisible into different smaller constructs [Far85].

Absorption

DSLs must absorb the common practices within the specific domain and also absorb these commonalities implicitly into the language to avoid generalization in the specifications. Absorption allows DSL users to assume safely and reliably how the DSL will behave and facilitates them to omit obvious details for focusing on the actual problem. With absorption a DSL must be able to deduce additional information concerning the required behaviour and structure of a specification. In order to validate this principle, the following properties have to be evaluated:

- *Absorption by exploiting context.* The implicit use of assumed structure and behaviour is based on the structure of the surrounding code of a particular construct.

- *Absorption by automation.* The DSL specification contains enough information to allow a DSL processor to automate certain tasks.

Absorption improves the *writeability* and *readability* of DSL specifications by omitting the necessity to write and read several times the obvious. Absorption also contributes to the *reliability* of a DSL rendering the specifications less error prone by unexpected interactions between their elements.

Standardization

Standardization restricts the grammar and semantics of a DSL to assist users in how to write a program that will solve their problem [McK76]. The DSL must be restricted to the minimal set of possible alternatives that the DSL expert requires to consider during a specification. This releases DSL users to figure out how to solve a problem. In order to validate this principle, the following properties have to be evaluated:

- *Restrictions on the grammar.* The DSL grammar constraints the specification to an explicit structure. This guides the DSL users to specify a solution for their problem using different notations (*e.g.,* conventions, idioms, patterns).
- *Restrictions via semantics.* The DSL implementation performs semantic and type checks on the specification declared by the DSL user.

Standardization improves *reliability* guiding DSL users during a specification with a restricted set of elements but with mechanisms to detect semantic errors. Restricted grammars contribute to improve *readability* and *writability* facilitating users to write and read relevant DSL specifications. Standardization contributes to *locality* within the specifications easing the localization of their relevant details. Standardization restricting the grammar improves the *lexical coherence* of the specifications by keeping related code physically adjacent to ease its understanding.

Abstraction

Abstraction formalizes and structures new concepts on top of existing ones to reduce the information available at a particular abstraction level [Gra01]. The concepts created by abstraction do not require the information that has been left out to understand the DSL specifications. The DSLs have meta-level abstractions that define the semantics of the specifications by interpretation or by translation. In order to validate this principle, the following distinguishable properties have to be evaluated:

- *Abstractions from technical complexity.* DSL specifications are released of skills that transcend the boundaries of the tailored domain.
- *Abstractions from irrelevant details.* DSL specifications avoid information that is not an essential part of the problem domain.
- *Abstractions from redundant information.* DSL specifications avoid possibly confusing information (*e.g.,* synonyms).

Abstraction reduces redundancy which contributes to a *non-ambiguous* DSL design. This allows DSL users to focus on what is essential in a solution and to avoid information that causes confusion. Reducing redundancy also contributes to *parsimony* by getting rid of two concepts with the same meaning.

Compression

Compression (*cf.* brevity [Gra01], compactness [Wei98]) reduces the number of lines of code in an specification but retains the amount of semantic details.

Different design characteristics can guide compression such as: visualize a unit of functionality within the programming editor, maintain the size of the problems proportional to specifications, and give the frequently used constructs a short name and more convenient syntax. Compression in DSLs must be fine-tuned for all the abstractions. In order to validate this principle, the following properties have to be evaluated:

- *Compression by factorization* to avoid reoccurring patterns in the code.
- *Compression by increasing conciseness* to remove elaborate statements, to change the syntactic sugar, and to omit default statements.
- *Compression by overloading notations* to provide a specification analyzable according to the execution context.

Compression improves *understandability* since a short specification tends to be more easily comprehended than a long one. However, compression increases the semantic density, thus this density must be kept within the capabilities of DSL users.

Generalization

Generalization is intended to minimize code duplication and to exploit qualities such as maintainability, separation of concerns, and code compression. The goal of generalization for DSLs is to reduce the amount of concepts by replacing a group of more specific cases with a common case. In order to validate this principle, the following properties have to be evaluated:

- *Generalization by inducing* forms a new concept to supplant a set of concrete concepts that cover fine-grained variations. The domain coverage of a generalized concept is explicitly formalized and turned into a computable form since the combination of variation points is made explicit.
- *Generalization by collapsing* searches for concepts which are more specific than another one and collapses them into a new enriched concept (*e.g.,* hypernyms/hyponyms).

Generalization contributes to the *longevity* of a DSL by easing the incorporation and evolution of specifications that were not explicitly defined. Generalization has an impact on *simplicity* allowing the solution of a general problem instead of solving specific ones. Generalization improves *writability* since the writer only needs to recall a single generalized concept and can apply it in many different ways. Nevertheless, generalization decreases the *readability* since the reader of a generalized specification has to interpret each time the general case to the specific one.

Optimization

Optimization is intended to decrease the time to construct executable software programs. Another dimension in optimization is intended to increase the execution performance of DSL specifications. The parts of a software program that are very frequently executed and consume a considerable proportion of the total execution time must be optimized. In order to validate this principle, the following properties have to be evaluated:

- *Optimization by enhancing the language runtime.* This is done by adding new algorithms and functionalities.

- *Optimization by tuning the semantics of the language.* This is done by changing the execution order (*e.g.,* permutations, caches, lazyness) or exploiting performance peculiarities of the language runtime.

- *Optimization by providing special-purpose constructs.* This is giving the DSL user the ability to use optimized constructs when appropriate.

Optimization contributes to the DSL *portability* since the abstract and small specifications can more easily be automated to target other platforms.

3.2 Requirements for the MonitA DSL

The analysis of requirements for the MonitA DSL was done by extracting domain knowledge from code and by defining formally the scope, terminology and concepts description of the workflow monitoring and analysis domain.

Part of the domain knowledge was extracted by inspection of code from BPEL, Java and JPDL workflow implementations. We defined a set of requirements to monitor and analyze the execution of two workflow applications: trouble ticket and loan approval. The trouble ticket workflow applications is implemented in a JPDL platform, whereas the loan approval application was implemented in a BPEL platform. The monitoring and analysis requirements for each workflow application were implemented directly in these platforms to analyze the abstractions provided by these workflow platforms and the level of specification in the implementation.

According to these implementations, we identified the necessity to provide abstractions for monitoring and analysis concerns to evaluate custom measurements in workflow applications during their execution. The domain-specific knowledge (terminology and semantics) was gathered from existing GPL code, related tools and techniques, technical documents, and the analysis of domain-specific necessities.

3.2.1 Monitoring and Analysis Desiderata

The following are the general requirements identified to specify M&A concerns in workflow applications:

- Multiple application developers must be able to specify M&A concerns for an existing workflow application. Workflow analysts have different needs to monitor and analyze a workflow application. Thus, all the MonitA specifications done by application developers to satisfy these needs must be incorporated into the target workflow application.

  For example, in the trouble ticket workflow application introduced in section 1.2, a workflow analyst can be interested in analyzing the workflow application by using measurements such as the *number of problems reported by a specific area of expertise*. At the same time, another workflow analyst can be interested in analyzing the workflow application by using time-related measurements such as *average processing time*.

- All the workflow variables used by a workflow application must be described in a shared model. In this way, this workflow data model can be shared and used by multiple MonitA application developers.

- An interface to access the actual value of the workflow variables and workflow engine information must be available to be used by MonitA specifications. This interface must provide access to the workflow instances under execution.

- The association between workflow variables and flow entities must be explicit in order to identify who (flow entity) is actually doing what type of work with the data. The operations that flow entities perform on workflow variables have to be identified to create custom quality measurements to assess strategic effects on the execution of workflow applications.

The following are the specific requirements identified to specify M&A concerns in workflow applications. These requirements are grouped into: monitoring concerns specification, measurement concerns specification, and control concerns specification.

Measurement Concerns Specification

- Application developers must be able to access predefined measurements and to define new custom measurements. Predefined measurements contain information about the workflow execution state (*e.g.,* execution time, workflow instances, state of flow entities, workflow variables) and are provided by the workflow engine. Custom measurements are specified by combining information provided by the workflow engine, by existing measurements, and by

workflow information. Custom measurements can be related to performance measurements and application-specific measurements.

Performance measurements such as average, maximum, and minimum workflow execution time, number of failures and loops must be evaluated on flow entities [zMR00]. Application-specific measurements (*i.e.,* specific to domain the workflow is modeling) must be evaluated on workflow data entities (*e.g.,* ticket, loan, invoice) and on who (flow entities) affects the workflow data.

- Application developers must specify how to create, update, delete and retrieve values of the custom measurements. All the measurement information that is captured and built has to be persisted for further analysis.

- The persistent mechanism must associate each measurement value with the workflow element that the measure is assigned with. A measurement value can be assigned to a particular flow entity, to a workflow instance, or to a set of workflow instances.

- Application developers must be able to navigate through all the measurement information that has been captured and built on a workflow application. The access to this information must support the navigation through the use of temporal constraints.

- The measurement values have to be evaluated against predefined values. When these values are reached, the measurements must be stored to keep trace of these values and their associated execution information (*i.e.,* temporal information).

Monitoring Concerns Specification

- Application developers must be able to specify the workflow event types they want to intercept. The monitoring specification must capture predefined and custom workflow event types.

  Predefined workflow events are related with flow entities and are typically provided by the workflow engine (*e.g.,* a transition is taken). Typically, these predefined workflow events are captured by tracking log state changes managed by the WFMS through observers to access directly the audit trail data.

  Custom workflow events are observed in terms of CRUD (create-read-update-delete) operations on the data entities (*e.g.,* a workflow variable is updated) and in terms of who is affecting the data (*e.g.,* a particular activity modifies the workflow data).

- Application developers must be able to specify the monitoring data that must be gathered when the workflow events are captured.

- The specification of monitoring concerns must support the combination of workflow events types to create new complex event types. For example, the workflow events captured when an activity starts and when it finishes can be combined into a single complex event type (*e.g.,* start/finish) [MSzM06].

- Application developers must be able to specify the complementary measurement and analysis code to be executed in response to a workflow event.

- Application developers must be able to define selectively the context (monitoring subjects) on which a workflow event has to match to apply complementary M&A concerns. The observation context has to be defined on a specific flow element (*e.g.,* by activity name), on all flow entities of a specific type (*e.g.,* for entities that are activities), and on all flow entities that use a variable (*e.g.,* for all activities that modify a specific workflow variable).

Control Concerns Specification

- Application developers must be able to specify a set of conditions to evaluate the measurement information in order to take complementary actions required to identify problematic aspects on workflow applications. These conditions must be evaluated during the execution of the workflow application to reduce the time required evaluate critical properties and to take decisions for workflow improvements.

- A set of notification mechanisms must be available to provide feedback to workflow analysts as soon as a measurement reaches certain conditions.

3.2.2 MonitA DSL Properties

The following are the main properties considered to design and develop the MonitA DSL:

- *Domain specific.* The design and development of MonitA is based on the analysis of the domain of workflow monitoring and analysis at runtime. MonitA defines a set of abstractions to support the specification of M&A concerns described previously (see Section 3.2).

- *Less developer effort.* The abstractions provided by the DSL have to increase the productivity of application developers for rapidly writing correct MonitA programs for workflow applications.

- *Workflow platform independence.* Another characteristic in the design of MonitA is to avoid concrete implementation details of the workflow application during the specification of M&A concerns. The DSL has to provide the mechanisms to specify the M&A concerns in terms of process model elements. This higher-level specification allows the M&A concerns to be declared in a uniform and technology independent way and to be reused across different workflow platforms and different workflow applications.

- *Data-centric specification.* A MonitA specification has to involve the data managed by the workflow application to provide an application-specific monitoring and analysis approach. A mechanism to expose a subset of the internal workflow data to the outside world has to be provided to access and involve this information in M&A activities that are external to the regular workflow execution.

- *Modularized specification.* M&A concerns have to be modularized with the workflow application. A modular specification offers application developers the opportunity to evolve M&A concerns independently of the workflow implementation and to reuse them. Application developers need to navigate over process and data models to validate the MonitA specifications.

- *Declarative.* The MonitA DSL must offer a declarative specification in which application developers describe what the specifications do instead of describing how it is implemented. This facilitates to specify M&A concerns in terms of domain concepts. Application developers need to manage the historic measurement information to improve the analysis that can be done at runtime.

- *Typed.* The DSL constructs must be typed enabling consistency checking of the data types during the composition of MonitA specifications with the workflow implementation. The consistency of these specifications can be checked in terms of the language syntax rules. The type of the variables must be always clear.

- *Executable implementation.* The M&A concerns have to be translated into executable code in the workflow platform in which the workflow application is automated. MonitA specifications must be implemented into existing workflow languages and engines to offer an approach highly adopted and integrated with WFMS. The MonitA specifications must be processed during the execution of the workflow applications. Application developers need to map conceptual process model events with the actual workflow events provided by the workflow engine. Application developers also need to map conceptual workflow variables with the actual workflow variables used by the workflow application and to customize its instrumentation to support MonitA specifications done in terms of data entities.

- *Compositional*. The MonitA specifications have to be composed automatically with the workflow implementation. This facilitates the evolution of the workflow application independently of the M&A concerns evolution. In this way the existing workflow generation process can be used. Application developers need to trace process model changes to automate the generation and composition of M&A concerns.

3.3 Design Rationale for the MonitA DSL

The approach we use to design our MonitA DSL is based on a language invention technique (see section 3.1.1) with no commonality with existing languages since there is no notation widely known in this application domain. We also use a language exploitation technique to exploit concepts of existing languages (piggyback) tailored towards different domains such as constraints modeling and separation of concerns.

We specified a formal design of our DSL using grammars for syntax specifications. A formal design has the advantages of a) detecting certain problems before the DSL is implemented, and b) implementing the DSL automatically by application generators, which reduce the implementation cost.

The following are the workflow domain entities to be used by the MonitA specifications:

- **Flow entities**. Every flow element in a BPMN model is a flow entity in MonitA. Every monitoring specification is done in terms of a flow entity. The properties to be observed by MonitA on flow entities are identity, instance information and state. The identity and instance information facilitate the unique identification of a workflow element, whereas state refers to the execution point of the entity.

- **Data Entities**. A data entity in MonitA represents a workflow variable that is used by a flow entity during the workflow execution. The properties to be observed by MonitA on data entities are scope information, state and performed interactions. The scope information facilitates the accessibility from a flow entity, the state represents the current value during the workflow execution, and the interaction refers to the operations performed by a flow entity.

The following approaches and technologies were used and served as inspiration for implementing MonitA. We describe the concepts and notations from OCL and AOP that we have reused into our DSL in what follows.

Object Constraint Language (OCL)

The Object Constraint Language (OCL) is used to specify constraints and to navigate over objects in UML models [OMG06b] [WK03]. The evaluation of an OCL expression returns a value and it cannot change the state of a model to avoid side effects. In OCL it is not possible to write program logic or flow control. OCL is suitable to navigate through models, but OCL expressions are not directly executable. Since OCL cannot alter the state of a model, a transformation language should do it.

Similarly to OCL, MonitA is conceived as a navigational language, a query language and a typed language. We have reused concepts and notations from OCL regarding the navigation through elements in a model and the operations on collections. The dot notation is used in our DSL to express navigations on the properties of domain entities. We also use the arrow-syntax to specify operations on collections.

In addition, MonitA acts as a transformation language to add M&A concerns to workflow applications and to change the state of measurement information that is external to the workflow application. Additionally, the MonitA DSL requires to: a) navigate through multiple models, b) transform process models to add M&A concerns, c) specify invariants on the additional M&A elements, d) allow pattern-based model queries on domain entities, e) create and manage new measurement data, and f) query instances of the workflow data and measurement data.

Aspect-oriented Programming (AOP)

Aspect-oriented Programming (AOP) technology facilitates introducing crosscutting behavior to existing applications in a modularized way [KLM+97]. This technology is good to transform a program by adding additional behavior and is highly expressive. However, it is still a low-level mechanism which is not adjusted to monitoring, measurement and control. Thus the implementation of M&A concerns is a complex task since it requires high technical expertise in the aspect-oriented language. Moreover, not all workflow languages support an aspect-oriented workflow language to add crosscutting behavior. In these cases the additional behavior has to be defined in terms of the underlying implementation.

We conceive MonitA as an aspect-oriented language with capabilities for a modular specification of M&A concerns. We offer modular constructs to encapsulate where to intercept the workflow application and how to instrument it with additional behavior. Breaking a program into functions offers multiple advantages such as the reduction of duplication of code, the reuse of code across multiple programs, and the distribution of MonitA specifications among various application developers.

We combine concepts from OCL and AOP since they can complement each other. The constraints that are expressed at the modeling level by means of an OCL-like language need to be satisfied at the implementation level. The M&A concerns specified at the design level can be materialized at the implementation level by using aspects technology. The modular implementation of constraints can enable the automatic composition of M&A concerns at the implementation level.

3.4 Summary

This chapter has explained the rationale and background behind the specification strategy of our approach. We have presented the importance of a domain-specific language (DSL) tailored to the workflow monitoring and analysis domain, the phases during its development, and the criteria considered to evaluate the design of our DSL. We have discussed how the MonitA DSL raises the level of abstraction to workflow developers for specifying M&A concerns by increasing their expressiveness, by easing their specification and maintenance, and by involving a larger group of application developers.

The following chapter presents a detailed description of the elements and syntax of the MonitA DSL.

Chapter 4

MonitA: The Monitoring and Analysis Language

This chapter introduces MonitA (**Monit**oring and **A**nalysis), a domain-specific language we have developed for monitoring and analyzing workflow applications. MonitA was designed to specify monitoring and analysis (M&A) concerns involving the data used by the workflow application in these specifications. The specification of M&A concerns involves different models such as: a process model, a workflow data types model, a data association model, a measurement data types model, and a MonitA model.

4.1 Monitoring and Analysis Specification

The M&A concerns must be specified in terms of process model elements to avoid concrete workflow implementation details and to be reused across different workflow platforms. In addition, the MonitA specifications must involve the workflow data to provide an application-specific monitoring and analysis approach (*cf.* design rationale in section 3.2.2). Nevertheless, to support the specification of M&A concerns at the conceptual level, the BPMN process model must provide a mechanism to describe the data types structure and the workflow variables used by the activities. Because it is not always the case, we offer an approach for setting the context for supporting MonitA specifications in terms of workflow relevant data.

The elements of our specification strategy are presented in three main parts: a) the specification of the data entities structure (complex data types) related to workflow and measurement information (Section 4.1.1), b) the specification of workflow variables used by a workflow application (Section 4.1.2), and c) the specification of M&A concerns required to monitor, measure, and analyze a workflow application during its execution (Section 4.1.3). The latter part is the focus of this chapter, thereby we present their main segments in sepa-

rate sections. The specification of M&A concerns comprises three segments: 1) measurement data (Section 4.2), 2) monitoring events (Section 4.3), and 3) analysis functions (Section 4.4). We present a discussion about the main features offered by the MonitA DSL (Section 4.6).

The workflow data types specification and workflow variables specification are not automatically synchronized with the workflow implementation, thereby they are a representation created at a given moment. A workflow developer has to configure the access and link to the actual workflow data (see Section 7.4).

4.1.1 Data Types Specification

We use XML schemas [BM04] to represent the data types of the workflow relevant data (workflow data types model) and of the measurement data that is external to the workflow application (measurement data types model). A data types model can contain complex data definitions and their associated primitive data types. Complex data types are a composite of other data types, which represent the structure of a data entity (entity data type) or a collection of data types. An entity data type is an ordered collection of one or more fields, which have a name and a data type. A collection data type is made up of one or more elements with the same data type.

The purpose of the workflow data types model is to model the data types of the variables that are manipulated and used by the workflow application. This data types model corresponds to a subset of the actual workflow data types that are made available to become involved in the MonitA specifications.

Figure 4.1 illustrates data entities such as Problem, Originator, Resolution used in the trouble ticket scenario. Our data modeling approach is based on the use of attributes as a mechanism to represent explicitly the structure of the data entities used in a workflow application and the relations between them.

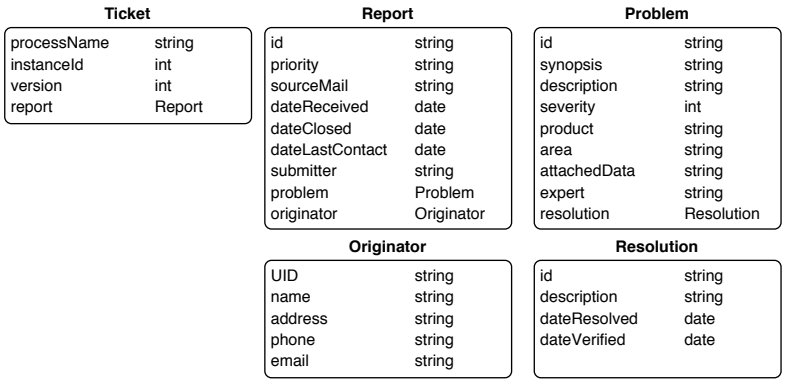


Figure 4.1: Workflow Data Types for the Trouble Ticket Scenario.

These data entities define the structure of the information that can be queried and interchanged from the workflow application. For example, in the

trouble ticket scenario (see Section 1.2) the schema representing the workflow data types contains a complex data type *Problem*, which contains a sequence of elements such as id, description, and area of expertise with their associated primitive data types (*e.g.,* string, double, boolean, integer).

The purpose of the measurement data types model is to model the data types required to define new measurement variables used for the analysis of a workflow application. For example, the measure associated to the measurement requirement defined in the trouble ticket scenario (see Section 1.2) requires the definition of a new measure data type. Thus the measurement data types model contains a complex data type named ProblemsByArea, which contains a sequence of attributes such as problems and area with their associated primitive data types (*i.e.,* int, string).

Measurement data types specified at the conceptual level in our approach are mapped to implementation entities. This mapping is a simple one-to-one link to an implementation entity (see Chapter 8).

A data types model is specified externally to the MonitA model and to the BPMN process model to be reused in multiple MonitA specifications. A data types model facilitates exchanging data between the workflow domain and the monitoring and analysis domain. Note that it is not our goal to provide a full modeling notation for data (*e.g.,* ORM, EER, UML) and that we do not want to discuss the usefulness of one modeling approach over another (*e.g.,* EER versus UML). We made a selection of existing approaches based on multiple criteria such as the usage of standardized specifications widely accepted.

4.1.2 Workflow Data Specification

Flow and data entities are the core workflow elements used in the specification of M&A concerns and in their composition with the existing workflow application.

Flow entities correspond to the elements contained in a process model specified according to the BPMN metamodel such as activities, events, and gateways. Each of these flow entities correspond to a *FlowEntity* type element that contains a set of attributes such as identifier, name, and activityType (see Section 2.1.2). Data entities correspond to workflow variables used by the flow entities of a workflow application. Because, there is not a means to formally define the workflow variables and their data types in BPMN, a process model has to be complemented with the description of the data used by its activities. Note that in a workflow generation process, the relation between flow entities and data entities is implicit since this is determined by the way data entities are implemented in the workflow.

We created a data association DSL to specify the workflow variables used by a workflow application, and their association with the BPMN flow entities. A data association specification (data association model) facilitates MonitA

specifications in terms of workflow relevant data and in terms of the operations performed on workflow variables. A data association specification is performed explicitly to facilitate the navigation throughout the elements of the workflow application. The data association model is specified externally to the monitoring and analysis concerns model, thus, it can be reused in multiple specifications. The grammar of the data association DSL can be found in Appendix C.

Listing 4.1 illustrates the data association model created to complement the trouble ticket process model (see Section 1.2) with the variables used by the flow entities in the workflow implementation.

```
process TroubleTicket import TroubleTicketBpmnModel, ProcessDataTypesModel

//Specify process variables
instanceScope Originator vOriginator
instanceScope Problem vProblem
instanceScope Report vReport
instanceScope Resolution vResol

//Associate data entities with flow entities
SubmitForm creates (vReport, vProblem, vOriginator)
IdentifyProblemandResolution creates(vResol) writes (vProblem) reads (vReport)
```

Listing 4.1: Data Association Specification for the Trouble Ticket Scenario.

In this example, all variable data types correspond to elements of the workflow data types model (`ProcessDataTypesModel`), whereas the activities correspond to flow entities contained in the process model (`TroubleTicketBpmnModel`). This illustrates, among others, a variable named `vReport` with Report data type that is visible for all flow elements within a workflow instance. This example also illustrates that the vReport variable is created by the `SubmitForm` activity and read by the `IdentifyProblemandResolution` activity.

A data association model describes a) the workflow variables manipulated by a workflow application, and b) the operations (*i.e.,* create, read, update, delete) that the flow entities perform on these variables.

First, to declare the workflow variables we assume the workflow data pattern named global data store [RtHEvdA05] in which the activities share the same data entities. Thus, when the value of a workflow variable is changed by an activity, this new value is visible for all flow elements in the workflow instance. We made this assumption since most of the workflow platforms do not pass data from one activity to another (*cf.* streaming pipe and filter). We assume that there is no direct access from external systems to change the value of workflow variables, and that encapsulation and data integrity are preserved by the workflow application. A complete discussion of the data modeling characteristics in MonitA with respect to workflow data patterns is presented in section 5.3.

Second, to associate the workflow variables with flow entities we do not describe the transfer of data entities between flow entities. We describe the operations that flow entities perform on data entities.

A data association specification is created with the *process* keyword followed by the name of the workflow application. The BPMN process model and the workflow data types model (Figure 4.1 in section 4.1.1) can be referenced in the data association model through the import keyword and the name of the model to reference. The models referenced in the data association model have to be separated by a comma.

The following sections describe the syntax of the two main elements that are required to create a data association model: workflow variables declaration, and workflow elements association.

Workflow Variables Declaration

A workflow variable is declared with the following properties: scope, data type, and name. The data type has to be defined in the data types model. The scope modifier defines the visibility that flow entities have on a workflow variable.

An *instanceScope* modifier means that the variable can be accessed during the execution of a workflow instance (*cf.* instance variables in object-oriented programming). For example, the following expression declares that there is a variable named vProblem with a Problem data type (defined in the data types model) for each workflow instance in the trouble ticket scenario (see Section 1.2):

```
instanceScope Problem vProblem
```

A *processScope* modifier means that the variable can be accessed by elements of all workflow instances (*cf.* class variables in object-oriented programming). For example, the following expression declares that there is a variable named processCreation with a dateTime data type that can be accessed by any flow entity in all workflow instances in the trouble ticket scenario (see Section 1.2):

```
processScope dateTime processCreation
```

Workflow Elements Association

We require to model the operations that flow entities perform on the workflow variables to allow the specification of monitoring and analysis concerns in terms of the data (*e.g.,* evaluate all activities that modify a specific variable). Data operations of interest are: create, change, delete and read. These data operations are described by using the name of a flow entity specified in the process model, followed by an operation keyword (*i.e.,* creates, writes, re-

moves, reads), and between parentheses separated by a comma, the name of the variables manipulated by the flow entity.

For example, the following specification declares that in the trouble ticket scenario (see Section 1.2) the activity named SubmitForm creates the vReport, vProblem, and vOriginator variables. This specification also declares that the activity named IdentifyProblemandResolution modifies the vResol and vProblem variables, and reads the vReport variable.

```
SubmitForm creates (vReport, vProblem, vOriginator)
IdentifyProblemandResolution writes (vResol,vProblem) reads (vReport)
```

4.1.3 Monitoring and Analysis Concerns Specification

This section presents the main concepts to specify monitoring and analysis (M&A) concerns related to monitoring, measurement and control (MMC). The grammar of the MonitA language can be found in Appendix A, whereas the semantics of the most relevant constructs of MonitA. A cross-is summarized in appendix B.

A MonitA specification is created with the *concern* keyword followed by a name assigned to the specification. The name of the concern defines the goal of the MonitA specification. For example, a MonitA application developer can create a specification named *timeAnalysis* to analyze time-related measures, whereas another MonitA application developer can create an specification named *qualityAnalysis* to analyze application-specific measures. The following illustrates the structure of a monitoring and analysis specification:

```
concern <specName>

//Inter-Model references
import <ExternalModelName>,...

//Measurement variables
<PersistenceStateModifier> <PersistenceModifier> <DataType> <VariableName>

//Monitoring events
on <WorkflowEventType> [<MonitoringSubject>]
      trigger <FunctionName(<ParameterName>=<ParameterValue>, ...)>, ...

//Analysis functions
mmcfunction <FunctionName> (<DataType> <ParameterName>, ...)
   <MeasurementAction> | <EvaluationRule> | <NotificationAction>
endfunction
```

Each monitoring and analysis specification comprises a reference to the data entities specification, a set of measurement variables, a set of monitoring events, and a set of analysis functions, each one comprising a set of measurement and control actions.

Different external models such as the process model, the data association model, and the measurement data types model can be referenced into a MonitA model through the *import* keyword followed by the model name. If there are

multiple external models, they have to be separated by commas. This inter-model references define the elements that can be used within the monitoring and analysis concerns specification. The BPMN process models are created conform to the the BPMN metamodel (see section 2.1.2) by using a process modeling tool (*e.g.,* BPMN modeler project). A data association model is created conform to the data association metamodel (see section 4.1.2). The measurement data types model is created conform to the data types metamodel (see section 4.1.1).

Listing 4.2 illustrates the specification in MonitA of the monitoring and analysis concerns described in the trouble ticket scenario (see Section 1.2).

```
1  concern EfficiencyAnalysis  import AssociationModel, MeasureDataTypesModel
2
3  persistent multiinstance Collection<ProblemsByArea> pba
4
5  on finish [root.SubmitForm]
6    trigger updateProblemsByArea(newArea=root.SubmitForm:vTicket.area)
7  on change [root.IdentifyProblemandResolution:vTicket.area]
8    trigger updateProblemsByArea(
9                newArea=root.IdentifyProblemandResolution:vTicket.area)
10
11 mmcfunction updateProblemsByArea (string newArea)
12   ProblemsByArea pro = pba->select(ProblemsByArea p|p.area==newArea)->first();
13   pro.problems = pro.problems+1;
14   if pro.problems  > 10
15   then alert(variable=pro, message='ProblemsByArea>10'); endif
16 endfunction
```

Listing 4.2: MonitA Specification for the Trouble Ticket Scenario.

This MonitA specification references the external models defined for the trouble ticket scenario such as the measurement data types model (`Measurement DataTypesModel`), the process model (`TroubleTicketBpmnModel`), and the data association model (`AssociationModel`).

Listing 4.2 line 3 specifies a collection of custom measures (`ProblemsByArea`) to capture the number of problems reported by a specific area of expertise for the trouble ticket scenario. The `multiinstance` modifier indicates that each measure of the collection can be associated in persistence with multiple workflow instances. Each measurement instance is indexed by the area of expertise attribute (defined in the measurement data types model) of the `vTicket` variable associated with the workflow instances. These measures are defined in terms of the workflow relevant data (area attribute of a ticket).

Listing 4.2 lines 5-9 illustrate two monitoring events which intercept the workflow execution. The first one (line 5) intercepts when the `SubmitForm` activity finalizes, whereas the second interaction (line 7) intercepts when the workflow variable `vTicket` is updated in the `IdentifyProblemandResolution` activity (requirement 2 in section 1.2). Both monitoring events trigger the `updateProblemsByArea` analysis function by capturing the area associated

with the vTicket variable in both activities (lines 6 and 9). Thus an analysis activity has to be added after the `SubmitForm` activity and another one after the `IdentifyProblemandResolution` activity.

For example, Listing 4.2 lines 11-16 illustrate an analysis function encapsulating the measurement and control requirement for the trouble ticket scenario (requirements 1 and 3 in section 1.2). In this specification, the `updateProblemsByArea` function requires a string type parameter to be executed (line 11). The measurement information is accessed through the `pba` measurement variable. Lines 12 and 13 illustrate how to retrieve a specific measure of the ProblemsByArea data type and how to increase the number of problems reported by a specific area of expertise (requirement 1 in section 1.2). Lines 14 and 15 illustrate an evaluation rule on the measurement variable to generate an alarm to be visualized in the dashboard when the number of problems of a specific area of expertise is higher than ten (requirement 3 in section 1.2). Other examples can be considered for analyzing a workflow application such as the classification of generic Service Level Agreement (SLA) metrics [CCDS03].

The following three sections describe the syntax of the main elements that are involved in a MonitA specification: a) measurement variables, b) monitoring events, and c) analysis functions.

4.2 Measurement Data Segment

A measurement variable is declared in MonitA to specify custom measures required to analyze the execution of a workflow application. All measurement variables have an implicit global scope modifier (*cf.* process scope modifier in process variables), thus, they have meaning in the context of the entire workflow application. This means that the measurement variables are visible to and can be accessed by all workflow instances.

The following sections describe measurement variable declaration, persistence management, initialization, and navigation.

4.2.1 Measurement Variable Declaration

All measurement information persisted in the measurement data store system (see Section 8.4.2) has to be declared as a measurement variable. All stored measurement variables are indexed by a unique persistence context. The persistence context corresponds to the workflow instances on which the M&A concerns are specified and in certain scenarios to the name of the activity associated with the measurement variable.

The following illustrates the structure of a measurement variable declaration:

```
<PersistenceStateModifier> <PersistenceModifier> <DataType> <VariableName>
```

A measurement variable is declared with a persistence state modifier, a persistence logic modifier, a data type, and the name of the measurement variable. We illustrate the semantics of a set of measurement variables in terms of the persistence mechanism, and then we detail the elements required to declare a measurement variable.

Assume that a workflow application with 3 activities has 4 workflow instances in execution. For 3 of these workflow instances a problem was reported in the area of expertise named *area1*, whereas for the last workflow instance the problem was reported in the area of expertise named *area2*. Figure 4.2 illustrates the MonitA specification of 4 measurement variables with different persistence logic modifier and their corresponding representation in the measurement data store.

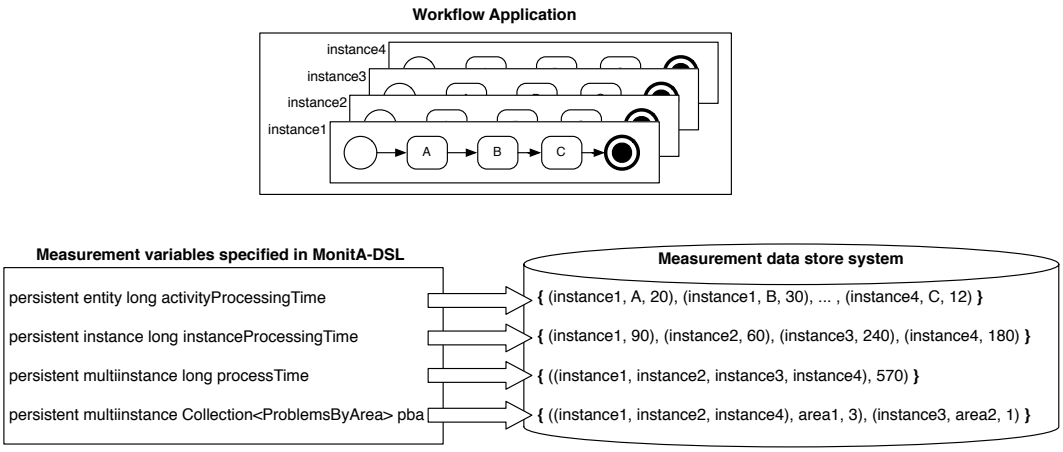


Figure 4.2: Persistence Logic for Measurement Variables.

The measurement variable named *activityProcessingTime* in Figure 4.2 corresponds to a collection of measure values of type long, which are indexed by the workflow instance and by the activity name. The measurement variable named *instanceProcessingTime* in Figure 4.2 corresponds to a collection of measure values of type long, which are indexed only by the workflow instance. The measurement variable named *processTime* in Figure 4.2 corresponds to a collection with only one measure value of type long, which is indexed by the set of all workflow instances. The measurement variable named *pba* in Figure 4.2 corresponds to a collection of measure values of type ProblemsByArea, which are indexed by a subset of workflow instances.

First, a *persistence state modifier* (*i.e.,* persistent, transient) defines whether the measure is temporal or it has to be made persistent in our external measurement data store system.

Persistent variables are defined for measures that are built to be persisted in our measurement data store system. For example, the following expression illustrates the specification of a collection of custom measures of a `ProblemsByArea` data type, which implement the measurement requirement defined in the trouble ticket scenario (see Section 1.2). Each of these measures capture the number of problems reported by a specific area of expertise. These measures are defined in terms of the workflow relevant data such as the "area" attribute within a Problem data entity. The `persistent` modifier means that each measurement variable that is changed within the collection is made persistent in our measurement data store system.

```
persistent multiinstance Collection<ProblemsByArea> problemsByArea
```

Transient variables are intermediate measures that are used for the computation of another measure. For example, the following expression illustrates a transient measure, which is alive during the execution of each activity:

```
transient entity dateTime startTime
```

Second, the *persistence modifier* defines the correspondence of a measure with a workflow entity, a workflow instance, or a set of instances. An *entity* modifier means that the value of the measurement variable is assigned to a specific flow entity within a workflow instance. An *instance* modifier means that the value of the measurement variable is assigned to a workflow instance. A *multiinstance* modifier means that the value of the measurement variable is assigned to multiple or to all workflow instances of a workflow application.

The persistence logic modifier for transient variables indicates when the variables are created and destroyed during the workflow execution. However, MonitA does not allow the specification of transient variables with a multiinstance modifier since these variables would remain alive indefinitely until the workflow engine stops running.

For example, the following expressions illustrate a set of measurement variables related with the trouble ticket scenario:

```
persistent multiinstance long processTime
persistent instance long instanceProcessingTime
persistent entity long activityProcessingTime
```

In this example, the processTime measurement variable stores the time spent executing a workflow application, thereby this measure is related to all instances of the workflow application. An instanceProcessingTime measurement variable is created for each workflow instance to store the value of the time spent executing the set of activities within a workflow instance. Finally, an activityProcessingTime measurement variable is created for each and every activity within a workflow instance to store its execution time.

Third, a *measurement data type* can be a simple data type, a complex data type, or a collection data type. Simple data types corresponds to the built-in primitive and derived types provided by an XML schema such as string,

boolean, double, duration, dateTime, and long. Complex data types for the measurement variables must correspond to one of the data structures defined in the measurement data types model (see section 4.1.1). A collection data type is made up of one or more elements with the same data type and it is specified with the *Collection<DataTypeName>* notation. Although certain measurement data types are similar to a dictionary, we only consider the collection data type to define a set of elements.

For example, the following expression illustrates a measurement variable with a collection data type which is parameterized with a ProblemsByArea complex data type:

```
persistent multiinstance Collection<ProblemsByArea> pba
```

Finally, the *variable name* is a unique identifier (persistence root) used to access the values of the measurement variables. For example, an application developer can use the variable named *instanceProcessingTime*, which is declared above, to access the set of measure values that correspond to the execution time of each workflow instance.

The measurement variables can be used to declare generic measures as well as application-specific measures. A generic measure applies to any workflow application since it does not refer to the data of a particular workflow application but to generic workflow engine information. For example, a generic measure to capture the processing time of each workflow instance. A domain-specific measure applies to a particular workflow application since it refers to specific data associated to it. The declaration of application-specific measures can also contain a set of properties. Each property consists of a name and a data type. For example, a application-specific measure of type ProblemsByArea to capture the number of problems reported by a specific area of expertise.

The automation of the entire business of a company typically involves a complex interaction of many interconnected (self-contained) workflow applications. In our work we limit ourselves to the construction of measurements that are not enterprise-wide. In essence we focus on the definition of measurements that do not cross the boundaries of a single workflow application specification.

Note that measures are only declared, thereby the definition of how measurements are computed/calculated is done in the analysis functions specification (see section 4.4).

4.2.2 Measurement Variables Initialization

All measurement variables are initialized with a default value when the workflow application is started and the workflow variables are created. The default value depends on the data type of the measurement variable either simple data type, complex data type, collection data type.

Table 4.1 summarizes the different data types associated to a measurement variable and their corresponding initial default values.

Data type	*Default value*
Complex	null
Collection	null
int	0
long	0
double	0
boolean	false

Table 4.1: Default variables Initialization Values

4.2.3 Navigation of Measurement Information

The MonitA language uses specific operators to navigate through the measurement and workflow information. We distinguish between intra and inter model navigation, navigation of collections, and navigation of measurement variables.

Inter and Intra Models Navigation

We use a specific operator depending on the type of navigation that has to be performed: intra model and inter model navigation. Intra model navigations stay in the same model, whereas inter model navigations combine elements from two different models.

The "." operator is used for intra model navigations. In the case of flow entities, the "." operator localizes by name the flow entities contained in the root element of the process model. For example, the following expression localizes the flow entity named SubmitForm in the process model:

```
root.SubmitForm
```

In the case of workflow variables, the "." operator access the attributes contained in the data type associated to a workflow variable that is defined in the association model. For example, the following expression retrieves the value of the attribute area for the variable vProblem:

```
vProblem.area
```

The ":" operator is used for inter model navigation. The navigation starts from an element of the process model towards a workflow variable (linked through the data association model, see section 4.1.2). This operator is used to retrieve the value of the workflow variables that are accessed by a flow entity in the workflow application. For example, the following expressions access a) the value of the data entity vProblem in the activity SubmitForm, and b) the value of the area field of a Problem data type named vProblem:

```
root.SubmitForm:vProblem
root:vProblem.area
```

Navigation of Data Collections

The "->" operator can be applied on a measurement variable of a collection data type to access the value of each element in the collection and to retrieve the value of certain related properties. The value or set of values that can be accessed from a collection are retrieved by using the following operations:

- The *size()* operation returns the number of elements in the collection.
- The *notEmpty()* operation returns true if the collection is not empty, otherwise false.
- The *isEmpty()* operation returns true if the collection is empty, otherwise false.
- The *select(DataType iteratorVariable— <expression>)* operation returns a subset of a collection. The collection is filtered by the attributes of the elements contained in the collection.
- The *first()* operation returns the first element of a collection.

For example, in the trouble ticket scenario (see Section 1.2), the following expression retrieves a measurement entity with ProblemsByArea data type from a measurement variable named setProblemsByArea that was defined of a collection data type. The expression selects a subset of elements in the collection whose area attribute has the value Area1, and then the first element is selected:

```
setProblemsByArea->select(ProblemsByArea pba| pba.area=='Area1')->first()
```

Navigation of Measurement Variables

The *allInstances()* operator returns a collection with all measure value instances that are related with a measurement variable in persistence. Measurement variables of a collection data type returns directly a collection of all measure value instances from persistence.

For example, the following MonitA expression returns all measure value instances of the measurement variable named activityProcessingTime, which was declared with an entity persistence logic modifier and with a long data type:

```
activityProcessingTime.allInstances()
```

MonitA offers a built-in *Measure* data type that can be used as data type of the iterator variable in the select operation when the values of the collection correspond to simple data types. In this scenario, the iterator variable does not contain additional attributes to navigate it.

Figure 4.3 illustrates persistence attributes that can be retrieved from measurement information. These persistence attributes such as instanceId, activityName, timestamp, and processVersion are read-only.

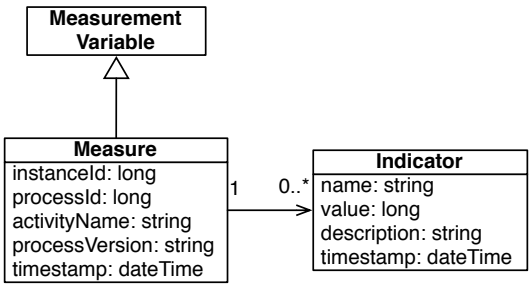


Figure 4.3: Persistence Information within Measurement Variables.

For example, the following expression retrieves a subset of the measure value instances related to measurement variable named activityProcessingTime, which are created in the last month:

```
activityProcessingTime.allInstances()
            ->select(Measure m| m.timestamp > dateTime.now().month(1-))
```

The navigation on measurement variables applies for the same workflow application (persistence space) where the monitoring and analysis concerns are specified. If the expression for navigating measurement variables cannot be evaluated, the MonitA execution platform returns a null value. The measure value instance for the current persistence context of a measurement variable (*e.g.,* workflow instance) can be retrieved by using the *current()* function (*e.g.,* activityProcessingTime.current()).

4.3 Monitoring Events Segment

A monitoring event is a point in time during the execution of the workflow application when something interesting for monitoring happens and where additional analysis functions are added to the workflow application. Monitoring events also specify the information that has to be captured and passed to the analysis function. The monitoring events allow quantification for multiple monitoring points in a workflow application.

The following illustrates the structure of a monitoring event specification:

```
on <WorkflowEventType> [<MonitoringSubject>]
      trigger <FunctionName(<ParameterName>=<ParameterValue>, ...)>, ...
```

The following three sections describe the main elements required to specify monitoring events: workflow events monitoring, analysis functions invocation, and execution context passing.

4.3.1 Workflow Events Monitoring

The first element to specify a monitoring event is the definition of the *workflow events* (state changes) to be intercepted in a workflow application. Workflow developers define the desired workflow event types associated to a particular flow entity that they want to be aware of. The workflow event types represent the execution state of a particular domain entity (*i.e.,* activity, workflow data variable) and are defined at a higher level without any specific implementation dependency. The workflow events correspond to basic abstractions such as starting or finishing a flow element, reading or writing properties of a data entity, and the creation of new data entities.

The workflow events are specified by using the *on* keyword, a *workflow event type* identifier, and between brackets, the *monitoring subject* which refers to a particular workflow entity or to a set of them. Each workflow event type is qualified with a workflow application context (workflow subject) such as the workflow application itself, the name of a flow entity, and the names of the flow entity and data entity context. These subjects identify the status of the domain entity or set of domain entities according to the workflow event types.

The workflow events fall into two dimensions:

1. *Flow events.* A flow event is the representation of the execution state of a particular flow entity (*i.e.,* activity, gateway). Workflow event types such as *start, finish* are defined in terms of flow entities. Flow events can be mapped to predefined events provided by the workflow engine (see Chapter 8).

   For example, the following expression illustrates a fragment of a monitoring event containing a workflow event that intercepts the start of an activity:

   ```
   on start [root.ActivityName]
   ```

2. *Data events.* A data event represents a workflow variable that was produced, changed, read, or deleted during the execution of a particular workflow instance. Workflow event types such as *change, delete, create, read* are defined in terms of workflow variables. Data events have to be mapped to customized mechanisms that allow the interception of these data events in the workflow implementation (see Section 7.4.4).

   For example, the following expression illustrates a fragment of a monitoring event containing a workflow event that intercepts when a workflow variable is updated by a particular activity:

   ```
   on change [root.ActivityName:processVariableName]
   ```

Table 4.2 summarizes the different workflow event types that can be intercepted by MonitA specifications.

Event type	*MonitA*	*Event description*
Activity started	on start	Start of the workflow root or a flow entity when it is reached by a transition
Activity finished	on finish	Finalization of the workflow root or a flow entity when it triggers a transition
Variable read	on read	Read of a workflow variable directly in the workflow root or through a flow entity
Variable write	on change	Value assignment to a workflow variable in the workflow root or through a flow entity
Variable create	on create	Creation of a workflow variable directly in the workflow root or through a flow entity
Variable delete	on delete	Elimination of a workflow variable directly in the workflow root or through a flow entity

Table 4.2: Workflow Event Types

We have defined patterns to combine these workflow events (*e.g.,* start and finish) to gather and process information from multiple state changes of the same entity. New event patterns can be constructed by using the "/" operator that acts as an "or" logical operator between two event types. The combination of event types results in a new event pattern that can be used to define a monitoring event that must match both workflow event types.

For example, computing the processing time of an activity requires the combination of two workflow events *activity started* and *activity finished*:

```
on start/finish [root.ActivityName]
```

We have also defined a set of patterns to define how to intercept a workflow event type in a particular monitoring subject and in multiple monitoring subjects. The following patterns are considered in terms of flow events and data events:

i. *Intercept the workflow application on a specific domain entity.* This pattern corresponds to the identification of a unique workflow element on a specific workflow event type.

   For example, in the trouble ticket scenario (see Section 1.2), the following expressions illustrate different workflow events whose workflow event type matches a particular monitoring subject:

```
on start [root.SubmitForm]
on start [root]
on change [root.SubmitForm:vProblem]
```

In the first scenario a flow event type matches a flow entity by its name. The second scenario shows a flow event type that matches the root element in the workflow. Finally, the third scenario shows a data event type that matches a workflow variable by its name.

ii. *Intercept the workflow application on all flow elements and on multiple flow elements regarding their type.* This pattern corresponds to the identification of a set of flow entities on a particular workflow event type. The "*" operator denotes a navigation through all the flow elements in the process model, whereas the "!" operator denotes a process model navigation through the flow elements of certain type (*i.e.,* Activity, Gateway). MonitA provides a built-in process data type named *FlowEntity*, which is used to capture properties of the flow elements intercepted in a monitoring event. A FlowEntity data type contains a set of attributes such as identifier, name, and activityType (*e.g.,* Process, Activity, Gateway)).

For example, in the trouble ticket scenario (see Section 1.2), the following expressions illustrate different workflow events whose workflow event type matches a set of flow entities of a specific type:

```
on start [FlowEntity ite| root.*]
on start [FlowEntity ite| root.!Activity]
on change [FlowEntity ite| root.*:vProblem]
on change [FlowEntity ite| root.!Activity:vProblem]
```

In the first scenario, the flow event type (start) matches all the flow entities in the process model. The second scenario shows a flow event type (start) that matches a set of flow entities of type Activity. In the third scenario, a data event type (change) matches all flow entities that write the vProblem variable. The last scenario shows a data event type (change) that matches a set of flow entities of type Activity that write the vProblem variable.

iii. *Intercept the workflow application on all flow elements that use a variable.* This pattern corresponds to the identification of a particular flow event type of a set of flow entities that operate workflow variables. The "/" operator denotes the combination of data operations that must be satisfied in order to match a flow entity.

For example, in the trouble ticket scenario (see Section 1.2), the following expressions illustrate different workflow events whose workflow event type matches a set of flow entities that operate on workflow variables:

```
on start [FlowEntity ite| root.* writes vProblem]
on start [FlowEntity ite| root.!Activity writes/reads vProblem]
```

The first scenario shows a flow event type that matches all the flow entities that write the vProblem variable. In the second scenario, the flow event type matches a set of flow entities of type Activity that write and read the vProblem variable.

Table 4.3 summarizes the patterns for specifying (quantifying) the context (monitoring subjects) that matches with the workflow event types.

MonitA pattern	**Pattern description**
	Flow patterns (applicable on flow events)
[root]	Expression to match against the workflow root
[root.EntityName]	Expression to match against a specific flow entity
[root.*]	Expression to match against any flow entity in the workflow root
[root.!FlowEntityType]	Expression to match against any flow entity of a specific type in the workflow root
[root.*<operation> VariableName]	Expression to match against any flow entity that performs an operation (*e.g.,* reads) or a set of them (*e.g.,* writes/deletes/reads/creates) on a workflow variable
[root.!FlowEntityType <operation> VariableName]	Expression to match against any flow entity of a specific type that performs an operation (*e.g.,* writes) or a set of them (*e.g.,* writes/deletes/reads/creates) on a workflow variable.
	Data patterns (applicable on data events)
[root:VariableName]	Expression to match against the workflow root when it operates on a workflow variable
[root.EntityName: VariableName]	Expression to match against a flow entity that operates on a workflow variable
[root.*:VariableName]	Expression to match against any flow entity in the workflow root that operates on a workflow variable
[root.!FlowType: VariableName]	Expression to match against any flow entity of a specific type in the workflow root that operates on a workflow variable

Table 4.3: Patterns for Events Context Definition

4.3.2 Analysis Functions Invocation

The second element in the specification of a monitoring event is the invocation of *analysis functions*, which can be incorporated in the workflow application at the points specified by a workflow event. Each analysis function is invoked before, after or in the associated monitoring subject depending on the workflow event specification.

The invocation of an analysis function or a set of them is done through the *trigger* keyword, followed by the name of the analysis function(s) separated by commas.

For example, the following monitoring event illustrates how the analysis function named updateProblemsByArea is applied when the SubmitForm activity finalizes:

```
on finish [root.SubmitForm] trigger updateProblemsByArea()
```

An analysis function invocation defines the instrumentation point in the workflow application depending on the workflow event types specified in the workflow events:

- A *start* workflow event type means that the analysis functions are incorporated into the workflow application just before each flow entity specified in the monitoring subject (*e.g.,* on start [monitoringSubject] trigger analysisFunctionName()).
- A *finish* workflow event type means that the analysis functions are incorporated into the workflow application just after each flow entity specified in the monitoring subject (*e.g.,* on finish [monitoringSubject] trigger analysisFunctionName()).
- A data event type (*i.e.,* create, change, delete, read) means that the analysis functions are incorporated into the workflow application just after the flow entity that uses the workflow variable is finished (*e.g.,* on change [dataMonitoringSubject] trigger analysisFunctionname()). In some cases it is possible to incorporate the analysis functions into the workflow application just after the workflow variable is affected and before the flow entity affecting this variable is finalized. However, this depends on the target language where MonitA specifications are automatically generated.

Multiple monitoring events can refer to the same monitoring subject, thus multiple analysis functions can be incorporated at the same point in the workflow application. The order to incorporate these analysis functions at the same workflow application point is not defined in MonitA. Although this can be problematic, we have evaluated the possible inconsistencies and interferences that can arise (see Section 11.2.3).

4.3.3 Execution Context Passing

The third element in a monitoring event specification is passing the execution context required by the analysis functions. An analysis function can require a set of properties to be executed. The value of these properties can be passed by gathering predefined measurements (*i.e.,* provided by the workflow engine) and custom measurements (*i.e.,* stored in our measurement data system). MonitA instruments the workflow applications to capture information from the monitoring subject to invoke the analysis function with this information. This information is passed to the analysis function as a set of values that are assigned

to the properties required by the function, which are separated by a comma and between parenthesis. The captured information is passed to the analysis function through named parameters. If the parameters are not named, the number and types of properties are passed by order according to the analysis function specification.

The value of workflow and measurement variables is accessed through their name. The "." operator can be applied on a workflow variable to invoke its attributes (*e.g.,* vProblem.expert), whereas the "->" operator can be applied on a collection data type to access each element of the collection (*e.g.,* problemsByArea->size()). The ":" operator is used to navigate through workflow variables associated to a flow entities that are specified in a different model (*e.g.,* root.SubmitFormF:vProblem.area).

Predefined measurements correspond to the information available during the workflow execution context such as flow entities and workflow variables information. Flow entities comprise information such as process name, process identifier, workflow instance identifier, and user. Workflow variables comprise the information associated with the flow entities. The navigation is always done through a flow element.

For example, the following expressions illustrate different scenarios to gather and pass data to analysis functions:

```
trigger functionName(property1=root.ActivityName)
trigger functionName(property1=root.ActivityName:variableName)
trigger functionName(property1=root.ActivityName:variableName.attributeName)
```

The first scenario shows how to localize a flow entity by its name for passing an Activity entity type to the analysis function. In the second scenario, the value of a workflow variable used by an activity is passed to the analysis function. The third scenario shows how to gather the value of a workflow variable property for passing it to the analysis function.

Custom measurements correspond to the information that can be captured from the measurement variables, which is stored in an external persistent system. This information corresponds to elements that are not dependent of the workflow execution. This information can be associated with measurement information previously captured.

For example, for the problemsByArea measurement variable that is of collection data type, the value of its collection size can be retrieved and assigned to a property of the analysis function:

```
trigger functionName(property1=problemsByArea->size())
```

The information captured is mapped or stored as a monitoring event that contains the information required by the analysis functions. A monitoring event captures the information associated to general properties such as process, version, instance.

4.4 Analysis Functions Segment

Measurement and control actions are encapsulated into an analysis function. Each analysis function has a name and a set of properties to distinguish the set of actions to be applied as a consequence of a particular workflow event. Each monitoring event is associated with one or more analysis functions. If multiple analysis functions are associated to the same monitoring event the order in which they get invoked is not defined.

We decouple the generation of monitoring events from analysis functions. Thus analysis functions are defined once and reused by multiples analysis concern specifications. Conceptually this is supported since a new MonitA specification can refer to existing analysis functions. However both models are integrated in only one specification. Separating analysis activities from their connections allows reusing both parts independently.

An analysis function is specified through the keyword *mmcfunction* followed by the name of the analysis function, and optionally a set of properties. These properties are specified through the set of variables (data type and name) that parametrize the analysis function, which are specified between parenthesis an separated by commas.

The following illustrates the structure of an analysis function specification:

```
mmcfunction <FunctionName> (<DataType> <ParameterName>, ...)
   <MeasurementAction> | <ControlAction>
endfunction
```

The properties of an analysis function facilitate reusing the measurement and control logic for multiple monitoring events that only vary their values. This avoids analysis functions to evaluate measurement and control actions against hard-coded values or values captured directly from the execution context.

When multiple analysis functions require the same properties to be executed, a *measurement data type* specified in the measurement data types model can be used to describe the structure of the data used to parametrize an analysis activity. This avoids the repetition of code by parametrizing the analysis activities with the set of properties. The creation of a variable of this type is specified through the name of the measurement data type and, between parenthesis and separated by commas, the assignment of values to its properties.

For example, the following expression represents the definition of an analysis function named updateProblemsByArea which is parameterized with a property name newArea of string data type:

```
mmcfuntion updateProblemsByArea(string newArea)
```

and the following illustrates the invocation of the analysis function parameterized with the value of creating an entity of type MeasurementDataType

that is parameterized with two strings:

```
trigger analysisFunctionName(prop=MeasurementDataType(p1='string',p2='string'))
```

Analysis functions represent the desirable measurement and control actions required a) to measure the workflow application, and b) to evaluate workflow application-specific measurements, and c) to perform notification actions according to the analyzed information. The following two sections describe the MonitA constructs defined to specify measurement an control actions.

4.4.1 Measurement Actions

A *measurement action* computes the previously declared custom measures, and defines the mechanisms to query and to manage the measurement information in our persistent measurement system.

MonitA offers a set of measurement expressions that can be evaluated in the measurement and control actions:

- *Access data values.* The workflow information and measurement information can be navigated through the name of the variable and, separated by "." or "->", the name of the property or operation to be accessed (*e.g.,* variableName->operationOnCollections). This depends on the data type associated to the variable (complex type, primitive, or collection). The value accessed through a workflow variable name corresponds to the value assigned in the current workflow instance.

- *Date time invocation.* The operations now() and today() on the dateTime data type can be used to assign a date and time value to a variable. The values of a variable with dateTime data type can be accessed through a set of operations such as year(), month(), day(), hour(), min(), and sec() (*e.g.,* dateTime.now().min()).

- *Literals.* A measurement expression can be a literal such as a string (*e.g.,* 'this is an string'), an number (*e.g.,* 23), or a name (*e.g.,* variableName).

- *Engine invocation.* There are predefined measurements that can be gathered from the execution context and that correspond to the information provided by the workflow engine. This information is accessed through the *engine* keyword and the name of the property (*i.e.,* instances, instance, wfEvent) to be retrieved. The instance property retrieves the identifier of the current workflow instance (*i.e.,* engine.instanceId), the instances property retrieves the number of workflow instances (*i.e.,* engine.instances), and the wfEvent property retrieves the name of the last workflow event that was triggered (*i.e.,* engine.wfEvent).

- *Measurement entity creation.* A measurement entity is created with the name of a complex data type and, between parenthesis and separated by comma, the assignment of the values to its attributes through named parameters. A complex data type corresponds to an element defined in the data types model (see section 4.1.1).

Table 4.4 summarizes the operations supported on the variables according to their data types.

Action	**Operations**
Operate Integers	+, -, *, /, +=
Operate Booleans	and, or, not
Operate Strings	+
Access DateTime values	year(), month(), day(), hour(), min(), sec()
Assign DateTime object	now(), today()
Compare values	<, <=, >, >=, ==, < >

Table 4.4: Operations Supported on Simple Data Types

The operations to access the date and time values of a dateTime data type can be parameterized with a negative or positive number to indicate a past or future value respectively. For example, the following expressions illustrate a) the navigation on a collection of problems to retrieve the problems created in the last 2 days, and b) the assignment of the date and time values within a month to a temporary variable named deadline:

```
Collection<Problem> recent;
recent = problems->select(Problem p, p.timestamp > dateTime.now().day(-2));
dateTime deadline = dateTime.now().month(+1);
```

The following subsections describe the measurement actions supported by MonitA: measurement data creation and persistence, temporary variables declaration, and indicators creation.

Measurement Variables Creation

A measurement variable is given a default value in its initialization (see section 4.2) and it is created/computed explicitly with an assignment through the "=" operator and a measurement expression. The name of a measurement variable (persistence root) must be used to retrieve its value and to add a new value from/to the persistence space. A measurement expression is evaluated in the current workflow execution.

For example, the following expressions illustrate how to evaluate measurement expressions to assign their values to measurement entities :

```
activityStartTime = dateTime.now();
proByExpert = ProblemByExpert(problems=1, expert='Expert1');
```

```
ProblemsByArea pro = problemsByArea->select(ProblemsByArea p| p.area=='Area1')
                                    ->first();
```

In this example, the measurement variables activityStartTime and proByExpert are declared externally to the analysis function (in the measurement variables segment). The first expression assigns the current date and time to a measurement variable named `activityStartTime`. The second expression creates a measurement entity with a ProblemByExpert data type and assigns its value to a measurement variable named `proByExpert`. The `ProblemByExpert` data type is specified in the measurement data types model (see Section 4.1.1) and contains a sequence of elements such as problems, and expert with their associated primitive data types (*i.e.,* integer, string). The Third expression specifies how to retrieve persistence measurement information through the persistence root named problemsByArea. The measurement information is filtered to assign the value to a temporary variable named pro.

If an expression specified to navigate and retrieve the value of the measurement data cannot be evaluated, the MonitA execution platform returns a `null` value. An evaluation rule (see section 4.4.2) must be defined to validate if the measurement data does not exist to proceed to create it. For example, in the trouble ticket scenario (see section 1.2), the information about the areas of expertise to compute the number of problems by area is not known before execution. Thus, the MonitA application developer must specify when and how to create this measure. The following expression illustrates this scenario:

```
ProblemsByArea pro = pba->select(ProblemsByArea p| p.area=='Area7')->first();
if pro==null then
  pro = ProblemsByArea(problems=1, area='Area7');
  pba->add(ProblemsByArea pi| pi=pro);
else
  pro.problems = pro.problems+1;
endif
```

A new measure value instance can be added to the collection of measure value instances in persistence by using the *add* collection operation. The *add(DataType iteratorVariable | <assignment>)* operation is parameterized with an assignment of the new measure value to the iterator variable of the collection.

The information referenced by a persistent measurement variable is retrieved and persisted in each transactional instance (*i.e.,* measurement data read, created, and updated). In MonitA there is no need to call a particular function (*e.g.,* save(), load()) to make the modifications on measurement data persistent.

Temporary Variables Declaration

MonitA allows the creation of temporary variables to ease the manipulation of domain entities in the monitoring and analysis concerns specification. A

temporary variable is defined with a measurement or process data type (see section 4.1.1) and with a variable name. A temporary variable must be explicitly given a value before it is used. The variables defined within an analysis function are only alive during its execution.

For example, the following expression denotes a temporary variable named *tempProblem* that is created with the value of the workflow variable named vProblem.

```
Problem tempProblem = root:vProblem;
```

Temporary variables enable MonitA application developers to refer to them by their name several times within an analysis function where that domain entity is required.

Indicators Creation

The values of the measurement variables are not reset. They are computed during all the execution of a workflow application. An indicator stores a copy of a measurement variable value when the corresponding measurement variable reaches critic values defined to analyze a workflow application. The values of the indicators are used to trace and evaluate the behaviour of the measurements in time. For example, an indicator can be created each time a measurement variable (*e.g.,* m) increases in 10 its value (*e.g.,* m - m.indicators->last() >= 10). The created set indicators (*e.g.,* {(i1,m,10), (i2,m,20), ..., (i20,m,200)}) facilitate the evaluation of a measurement variable against its behaviour in time (*e.g.,* frequency to reach a critic value).

An indicator is declared by using an *Indicator* data type, and the name of the indicator. An *Indicator* data type is a built-in data structure provided by MonitA, which has attributes such as measure, value, and description. These attributes have data types such as Measure, long, and string respectively.

An indicator must be explicitly given a value when it is declared. The indicators are typically created when the condition of an evaluation rule is satisfied. A measurement expression can be used to create an indicator by using the *Indicator* data type and, between parenthesis and separated by comma, the assignment of the values required by data type attributes through named parameters. An indicator cannot be associated directly to measurement variables with collection data type since an indicator must correspond to a particular measure value instance in persistence.

When an indicator is created, it is persisted with the information such as the persistence context (*e.g.,* workflow instance) associated to the measurement variable, the name of the indicator, and a timestamp. Figure 4.3 illustrates the information that can be used to navigate on an indicator associated to a measurement variable. An indicator becomes persistent only if the corresponding measurement variable has persistent state, otherwise the indicator is only alive

during the analysis function execution.

For example, the following expression creates an indicator named i1 when the processing time of the current workflow instance is higher than 240 minutes. The measurement variable named instancePTime was declared as persistent and its value had to be computed before the indicator is created:

```
long temp1 = instancePTime.current();
if  temp1 > 240
then Indicator i1 = Indicator(measure=instancePTime, value=temp1,
                              description='Processing time > 240 minutes');
endif
```

A collection of indicator value instances with Indicator data type is accessed through the *indicators* attribute of a measurement variable. For example, the following expression retrieves the number of times that a workflow instance has been executed in more than 240 minutes (*i.e.,* indicator named i1) in the last 5 days:

```
int temp = instancePTime.current().indicators->select(Indicator i| i.name=='i1'
                             and i.timestamp > dateTime.now().day(5-))->size();
```

4.4.2 Control Actions

A *control* action comprises the definition of evaluation rules and the corresponding notification actions to be taken in response to these evaluations. The following two subsections describe the control actions supported by MonitA: notification actions and evaluation rules.

Notification Actions

The notification actions help identifying potential improvements in a workflow application by calling an external function such as:

- *Sending an email* to communicate relevant information:

```
notify(destination='string', subject='string', content='string')
```

- *Generating an alarm* to be visualized in a dashboard:

```
alert(variable=measureObject, message='string')
```

- *Creating event logs* with the analysis information for further evaluation:

```
trace(path='string', message='string')
```

Note that the control actions are not intended to be used to alter the control flow of the workflow application at runtime (*e.g.,* to dynamically adapt the workflow application at runtime).

Evaluation Rules

An evaluation rule represents prescribed states established on measurement information based on the occurrence of a specific condition. When a prescribed state is reached, the evaluation rule triggers the measurement and notification actions associated to it. The evaluation rules can be specified over 1) the measurement variables, 2) the indicators related to a measure variable, 3) the workflow execution information, and 4) the workflow variables.

An evaluation rule is specified with an if-then conditional expression:

```
if <conditionExpression>
then <actionExpression>
else <actionExpresion> endif
```

The condition block denotes a boolean expression that can involve multiple elements such as: invocations from workflow events, gather information, reference domain entities, comparison expressions. These elements can be combined in logical expressions and in nested expressions. The action block is defined according to measurement actions (see section 4.4.1) and notification actions (see section 4.4.2).

For example, the following expression illustrates an evaluation rule with a condition expression that denotes the result of two comparisons that are related with an *and* expression. Both comparisons involve the result of navigating through a variable named *pro* with ProblemsByArea data type. The attributes *problems* and *area* correspond to the properties of the measurement data type named *ProblemsByArea*:

```
if pro.problems > 10 and pro.area=='Testing'
then alert(variable= pro, message='ProblemsByArea>10');
endif
```

4.5 Discussion

The MonitA language is targeted to workflow developers and not to business experts. This is because there are still concepts that require to be moved to a higher level of abstraction. For example, data management concepts for defining, creating, persisting, and navigating collections of measurement data have to be abstracted to hide technical complexity. MonitA provides abstractions to define and persist measurement variables releasing application developers from learning a complete model to manage the persistence. Although the management of measurement information do not require a deep technical background or programming skills, there is always a possibility to create tools that facilitate it specification and navigation.

The evaluation rules in MonitA are basic if-then-else statements which just perform actions (measurement and control) without returning a value as a result. We use the evaluation rules only for the evaluation of measurement

information, thereby MonitA does not require all capabilities (*e.g.,* nested expressions) of a programming language. In terms of business rules, our evaluation rules correspond to action enabler rules which check conditions at a certain event and upon finding them true apply an action. MonitA does not have an engine, thereby the execution of MonitA specifications depends on the workflow execution.

4.6 Summary

This chapter has explained MonitA, a domain-specific language we have developed for monitoring and analyzing workflow applications in a uniform and technology-independent way. MonitA provides an expressive medium to specify M&A concerns in terms of workflow relevant data by describing the workflow variables associated to flow elements in a BPMN process model. This allows an analysis of the workflow application in terms of the specific business domain it is modeling (*e.g.,* Trouble Ticket). Specifically, the monitoring is based on the declaration of data entities interactions, the measurement is driven by the declaration of custom measurements related to workflow relevant data, and the control specification defined in terms of the measurements performed.

An advantage of MonitA is the use of domain-specific notations from the beginning of the specification, thereby raising the level of abstraction for specifying M&A concerns. The MonitA specifications are not tightly coupled to the workflow implementation since they are specified in terms of the process model. The modularized nature of MonitA specifications improves maintainability since the M&A concerns are easy to locate and no longer crosscut the workflow implementation. MonitA specifications can be shared between workflow developers to improve reusability. Although the dependency now lies with MonitA, the specification of M&A concerns is simpler since application developers use a single domain-specific language staying away from implementation technologies (*i.e.,* workflow language, workflow engine, instrumentation language).

The following chapter presents the studies we performed to evaluate different elements of the design and expressiveness of our domain-specific language.

Chapter 5

Evaluation of the MonitA Language

This chapter presents the studies we performed to evaluate different elements of the design and expressiveness of our domain-specific language. We also evaluate our data modeling approach in relation with workflow data patterns.

Section 5.1 presents the evaluation of our domain-specific language against a set of design principles typically associated with DSLs.

Section 5.2 presents a workshop and questionnaire developed to evaluate the expressiveness and learnability of the MonitA DSL. These properties are evaluated by considering criteria such as suitability for users, strengths and limitations to specify monitoring and analysis (M&A) concerns, and complexity in the specification. The workshop purpose was the identification of potential improvements to the expressiveness of MonitA, of which some were introduced directly in MonitA and other ones are considered for future work. This is a preliminary study which needs to be repeated with a larger user base.

Section 5.3 summarizes the key properties of our approach for modeling data in high-level workflow models and in the associated MonitA specifications on that workflow. We also analyze workflow data patterns to describe the relation between data management in workflow applications and our data modeling approach.

5.1 Evaluation of Design Principles

We evaluated our domain-specific language based on the following 7 design principles [Cle10] [GC10] described in section 3.1.2. These principles capture the essence of a good DSL design. This is why we evaluated them on MonitA to identify its distinguishable properties and to avoid to get an ambiguous and useless DSL. We present a short explanation of these principles and the distinguishable properties observed in MonitA.

Representation

This principle is related to the use of the most optimal syntax to specify monitoring and analysis concerns. The main idea is to express things using domain-syntax. However, there is no a standardized and common syntax in the workflow monitoring and analysis community. The following are the distinguishable properties observed in MonitA according to this design principle:

- An example of the *domain syntax* property is the use of conceptual monitoring event types (*i.e.,* on start, on finish) for representing workflow event types (*e.g.,* node-enter, node-leave, task invocation). This facilitates to specify M&A concerns at a higher level of abstraction.

- An example of the *distinguishable syntax* property is the use of named arguments in MonitA (*cf.* Python programming language). They ease the specification of M&A concerns since the information required to execute a set of analysis functions is distinguishable.

- An example of the *familiar syntax* property is the reuse of certain OCL elements for models navigation in MonitA. MonitA looks more familiar to certain users by using the arrow-syntax to navigate over collections, and the dot notation to access and retrieve values of the properties of a model.

Absorption

This principle is related with the implicit use of assumed structure and behavior in the MonitA language to avoid tedious specifications. Absorption by exploiting context means to hide and derive information, whereas absorption by automation means to avoid the execution of certain manual tasks. The following are the distinguishable properties observed in MonitA according to this design principle:

- An example of *absorption by exploiting context* in MonitA specifications is the support for implicit references to the data association model and to the data types model. The import keyword is used to reference these models.

- Multiple examples of *absorption by automation* are observed in MonitA. MonitA automates the implementation of observers in the workflow application reducing the specification *effort*. MonitA persists automatically the measurement information contained in a workflow variable of collection data type when it is modified. This allows the *consistency* of the measurement information. MonitA also provides implicit accessor methods for the workflow and measurement variables. Finally, the management of workflow instances during the specification of M&A concerns is transparent for workflow developers.

Standardization

This principle is related to offering a structured and standardized way of solving a problem to reduce erroneous specifications. Standardization by restrictions on the grammar enforces the way M&A concerns are specified. Standardization by restrictions on semantic checks for domain-specific errors (*e.g.,* type checking, correct programs, variable referencing) at compilation time. The following are the distinguishable properties observed in MonitA according to this design principle:

- Standardization by *restrictions on the grammar* is identified in different parts of MonitA. Before specifying M&A concerns, workflow developers are required to set up the process specification by defining a data association model. A new measurement variable must be declared with a set of ordered modifiers and attributes (*i.e.,* persistent modifier, persistence association modifier, data type, variable name). MonitA restricts the data types that can be used in the specifications to those defined in the data types model decreasing the errors that can be introduced by DSL users. The grammar of MonitA enforces the structure (standard) to specify monitoring concerns: all the workflow subjects to be observed are selected, then the analysis functions required to instrument the workflow application are indicated, followed by passing data to the analysis functions.

- Standardization by *restrictions on semantics* can be easily observed when the MonitA specifications are implemented and integrated within a workflow application. Different semantic checks can be performed to identify when: data types mismatch, invoked measures do not exist, undeclared variables are referenced, data entities not defined in the association model are referenced, analysis functions invoked by a monitoring event are not declared, and when imported models do not exist. MonitA application developers must declare the measurement variables before they can be used. MonitA can exploit information beyond what is written in the specification by navigating through the data association model and the data types model.

Abstraction

This principle is related to constructing new abstractions on top of existing ones to reduce the information available at a particular abstraction level. Abstractions from technical complexity hide complexity, abstract information in the domain, and remove repetition. Abstractions from irrelevant details remove details which are not important to know about. Abstractions from redundant information get rid of similar information that causes confusion. The following are the distinguishable properties observed in MonitA according to this design principle:

- *Abstractions from technical complexity.* MonitA abstracts the information involved in the observation of workflow events within specific monitoring subjects into a monitoring event. MonitA removes the data types' definition from its specifications to reduce their verbosity. MonitA provides persistence modifiers to define measurement variables in order to hide the complexity of the persistency management. The impact if MonitA does not provide these persistence modifiers is that application developers need to learn a complete model to manage the persistence.

- *Abstractions from irrelevant details.* MonitA removes irrelevant details from the specifications such as: the use of a "new" keyword for the creation of new objects, and an explicit "of" keyword for defining the workflow monitoring context (*i.e.,* monitoring subjects).

- *Abstractions from redundant information.* MonitA encapsulates the measurement and control actions into an analysis function. This avoid repetition.

Compression

This principle is related to reducing the code footprint while retaining the same semantic details. *Compression by restructuring* defines patterns and compresses from them, whereas *compression by increasing conciseness* combines concepts into one. The following are the distinguishable properties observed in MonitA according to this design principle:

- *Compression by restructuring.* MonitA defines different patterns to compress the specifications such as: quantification of the workflow subjects that are instrumented with analysis concerns, and composition of event types to specify monitoring events. This reduces the number of monitoring events that have to be specified by application developers.

- *Compression by increasing conciseness.* MonitA provides the "on" and "triggers" keywords to combine three concepts (workflow events observation, analysis function invocation, gathering data) into a single concept named monitoring event. The monitoring events increase conciseness instead of using if-then statements. MonitA also provides operations on collections to increase conciseness.

Generalization

Generalization is related to inferring a common case from specific cases, and to deal with 2 similar concepts. In contrast to abstraction, in generalization the new concepts can be decomposed into the former ones. The following are the generalization properties observed in MonitA:

- *Generalization by inducing.* MonitA induces the analysis function concept to group different kinds of measurement and control actions together. The domain coverage of an analysis function is explicitly formalized and turned into a computable form since the combination of measurement and control actions are made explicit.

Optimization

This principle is related to performance improvements to write and execute MonitA specifications faster. The following are the optimization properties observed in MonitA:

- Optimization by tuning the semantics of the language. The execution semantics of the MonitA language, defined in the application generator, can be easily changed to improve the performance of the DSL specifications at runtime.

- Optimization by providing special-purpose constructs. MonitA provides filtering constructs and abstractions to customize the points in the workflow application where additional M&A concerns must be added (*i.e.,* monitoring subjects). The quantification mechanism associated to monitoring events facilitates the identification of a set of particular monitoring subjects (*i.e.,* flow entity, data entity) that are interesting for monitoring. Thus, there is no necessity to add intercept every workflow event generated by the workflow application.

The evaluation of these principles in MonitA give us an indication of a good design. We observed a consistent syntax and semantics in the MonitA specifications, which are scoped within the borders defined in the DSL analysis. MonitA is nonrestrictive to specify M&A concerns and allows simpler specifications. We identified the main distinguishable properties of MonitA to get an unambiguous and useful DSL.

5.2 Evaluation of Expressiveness and Learnability

We conducted a study by means of a workshop and questionnaire among potential users of MonitA to evaluate its expressiveness and learnability. We based this study on the questionnaire presented by Hermans et al. [HPvD09]. We took this questionnaire as a reference since it presents the identification of a number of DSL success factors and how they can be assessed. We complemented this questionnaire with questions specific to the interaction with MonitA.

The goal of this study was to determine the main advantages and weaknesses of MonitA to identify potential improvements for its expressiveness. Several outcomes and suggestions obtained in this study were incorporated in the MonitA language, whereas other ones were considered for future work.

5.2.1 Basic Study

We conducted the study in an academic environment by asking a group of 8 master students in computer science to participate in the workshop and questionnaire[1]. Even though the number of users is too low, it gave us the opportunity to get early feedback. So this study should not been seen as a final assessment, but as a starting point to improve MonitA. The workshop contains 1) a BPMN model with a general description of the trouble ticket workflow application, 2) the data types involved in this application, 3) a data association model with the description of the operations performed by the activities on the data, 4) an overview of MonitA, 5) a number of examples of MonitA specifications, and 6) a set of exercises to specify M&A concerns. The questionnaire allowed us to evaluate expressiveness issues in MonitA such as what is/is-not possible to specify, what is difficult to specify, and what is not clear.

The 8 students work as developers for software companies and they are experts in MDE, however, they have limited experience with workflows. Two of these students have used the MonitA language in two different projects, whereas the other 6 students have no experience with MonitA. These students received 1 hour training on MonitA and the purpose of the study. We estimated that the time needed to answer the workshop and questionnaire was 180 minutes since there is no experience with MonitA.

Different factors can decrease the success of this study such as the fact that developers did not use the DSL editor to specify the M&A concerns. This would facilitate the specification, however, it was important to validate the expressiveness of MonitA without a tool kit. Another factor that complicates this study is the short time spent for training.

Table 5.1 illustrates the questions adopted and defined to evaluate the expressiveness and learnability in MonitA.

The DSL success factors under consideration in this study are expressiveness and learnability:

- When using a DSL, *Expressiveness* is the ability to implement domain-specific features compactly [MHS05]. This is the briefness for specifying M&A concerns uniformly in different levels of abstraction. A DSL limits the scenarios that can be expressed since the language is specific to a domain.

[1]The complete workshop and questionnaire for MonitA application developers can be found at https://soft.vub.ac.be/soft/members/oscargonzalez/research

Id	*Question*	*DSL Success Factor*
Q1	How much time did you spent specifying the exercises?	Learnability
Q2	How much time did you spent developing the workshop?	Learnability
Q3	How many years have you worked as a professional software developer?	Learnability
Q4	How much experience do you have with MonitA?	Learnability
Q5	How much time did it take you to get to know MonitA?	Learnability
Q6c	Is MonitA powerful?	Expressiveness
Q7	Did you deny a requirement because you knew you would not be able to implement it using MonitA?	Expressiveness
Q8	What was not possible to specify in MonitA?	Expressiveness
Q9a	Is MonitA difficult to use?	Expressiveness
Q9b	Does MonitA restrict your freedom as programmer?	Expressiveness
Q9c	Does MonitA have all features you need?	Expressiveness
Q10	What kind of monitoring and analysis requirements would you like to specify with MonitA?	Expressiveness
Q11	What changes or additions to MonitA do you propose?	Expressiveness
Q12	Do you have any other remarks regarding MonitA?	Expressiveness

Table 5.1: Questions Used to Evaluate Expressiveness and Learnability in MonitA

The expressiveness of the MonitA language is evaluated according to the analysis of requirements presented in section 3.2.

To measure the expressiveness of MonitA we asked the level of agreement / disagreement, by using a five-point scale, with a set of statements to investigate whether MonitA: is a powerful DSL (Q6c), is difficult to use (Q9a), restricts their freedom as programmers (Q9b), and provides all constructs to specify monitoring and analysis concerns (Q9c). We also asked the frequency, by using a five-point scale, with which the student had to deny a requirement since it could not have be specified in MonitA (Q7).

- *Learnability* refers to the time and effort that developers invest in learning an extra language [Spi01]. The learnability of MonitA is evaluated by the time invested in actually learning our DSL.

  To measure learnability we asked different questions to identify the effort (number of days of 8 working hours) invested in learning MonitA.

The measurement of various success factors is done through two five-point Likert [Lik32] scales used by Hermans et al. [HPvD09], which have an additional neutral option [PK01]. The frequency scale ranges from strongly disagree, disagree, neutral, agree, to strongly agree. The level of agreement/disagreement ranges from very often, often, sometimes, seldom, and never.

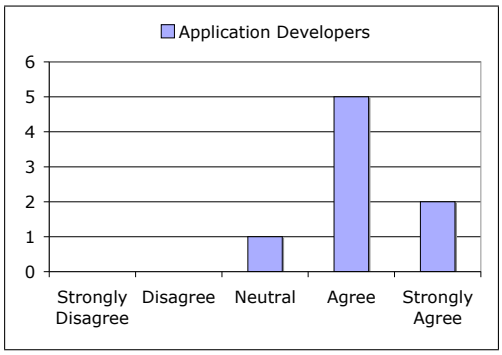
5.2.2 Results

The results show that using our DSL, the M&A concerns can be implemented compactly. The following illustrates the results after measuring the expressiveness and learnability in MonitA.

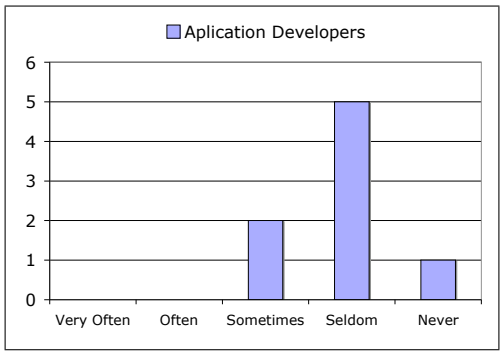
Expressiveness

The responses of the students to question 6c show that they can specify M&A through a powerful expressiveness mode (Figure 5.1a). 62,5% of them agree that MonitA is powerful to write expressive M&A concerns, 25% are strongly convinced of that, whereas 12,5% have no opinion.

The students indicated that they had to deny a requirement since it could not be specified in MonitA (Figure 5.1b). They also expressed that they did not review the documentation but only the training material.



a) Q6c. Do you consider MonitA powerful?



b) Q7. Did you deny a monitoring requirement?

Figure 5.1: Questions 6 and 7 for Measuring the Expressiveness of MonitA.

The responses to question 9 show that MonitA is not difficult to use since none of the students indicate that (Figure 5.2 question 9a). The students do not consider MonitA restrictive for their programming freedom. Thus the limited scope imposed by the DSL is not a problem for MonitA specifications (Figure 5.2 question 9b).

Different application developers indicated that MonitA requires more features (Figure 5.2 question 9c):

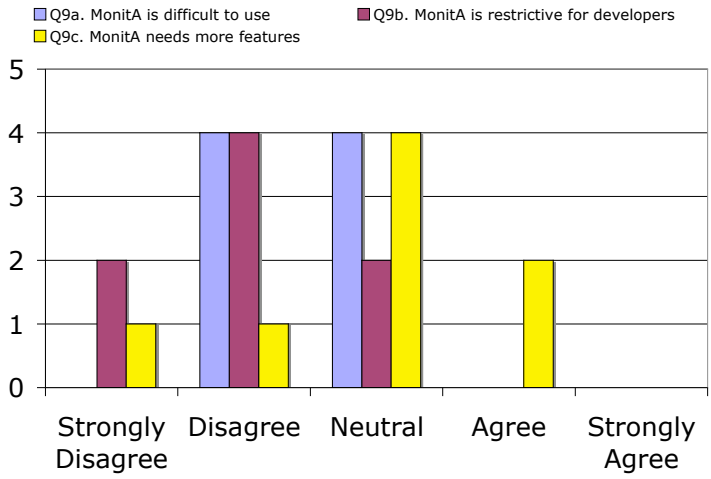


Figure 5.2: Question 9 for Measuring the Expressiveness of MonitA.

- Initialize measurement data collections. This feature was incorporated to the DSL.
- Analyze measurement information considering future dates (*e.g.*, deadline). This feature was incorporated to the DSL.
- Validate that the measurement information has not been created. This feature was incorporated to the DSL.
- Navigate on indicators. This feature was incorporated to the DSL.
- Specify monitoring events in terms of workflow transitions to support more interception points in the workflow application. This feature is considered as future work.
- Define the execution order of the M&A concerns. This feature is considered as future work.
- Compare the previous value of a measurement with its current value (*e.g.*, old and new functions). This feature is considered as future work.

We can observe that many students do not have an opinion about the difficulty and restrictiveness of MonitA. This is probably due to not take the time to review the documentation of the MonitA language.

The main threat to expressiveness validity is that a wider range of enterprise workflow scenarios and workflow developers has to be considered to evaluate the expressiveness of MonitA.

Learnability

The responses indicate that the students did not spend enough time to learn about MonitA (Table 5.2).

Learning Time (hours)	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Application developers	0	3	3	0	0	0	0	1	0	0	0	0	0	1

Table 5.2: Measuring the Learnability of MonitA.

Most students were able to develop the workshop with the training material plus a couple of hours extra. However, the users that spent more time learning MonitA specified the exercises adequately.

5.3 Data Modeling Characteristics

We have defined a set of management interactions to create, retrieve, and update measurement data entities. In addition, we have defined a set of workflow events to request notification when data changes. The latter interactions exchange data representations to support monitoring specifications. We monitor interactions on data to capture relevant information to these events in order to assure workflow execution integrity. We focus on supporting events and relations in terms of data interactions that are important for a particular MonitA specification.

The management interactions are simplified for the workflow variables since we do not want to alter the state of the workflow application. Nevertheless, we provide the complete set of interactions on the measurement variables to support data access according to the requirements defined for monitoring and analysis.

Table 5.3 summarizes the key properties of our approach to modeling data on high-level workflow models and on the associated monitoring specifications. We present if the property is supported and the corresponding implementation technology.

5.3.1 Relation to Workflow Data Patterns

The well-known workflow data patterns of Russell et al. [RtHEvdA04] aim at capturing the various ways in which data is represented and utilized in workflows. They also present the evaluation of a set of workflow products according to the evaluation criteria of each pattern. We analyzed these patterns to describe the relation between data management in workflow applications and our measurement data.

Property	*Workflow Variables*	*Measurement Variables*
Variable declaration	DSL	MonitA
Data types	XSD	XSD
Access variables	WSDL	WSDL
Variable representation	XML, Object-based	XML
Intercept data interactions	AOP	N/A
Create variable	No	Yes
Retrieve variable	Yes	Yes
Update variable	No	Yes
Delete variable	No	Yes

Table 5.3: Properties to modeling data in MonitA

Modeling data in workflow management systems comprises the following characterization: data visibility, data interaction, data transfer, and data-based routing. We consider this data characterization to establish a relation with our data association model and measurement variables model.

Data Visibility

These patterns define how data elements can be viewed by various components of a workflow application. Data elements can be accessed a) by the context of individual execution instances of a task (*Task Data* pattern), b) by each of the components of the corresponding subprocess (*Block Data* pattern), c) by a subset of the tasks in a workflow instance (*Scope Data* pattern), d) by multiple task instances within a single workflow instance (*Multiple Instance Data* pattern), e) by a workflow instance (*Case Data* pattern), f) by multiple workflow instances on a selective basis (*Folder Data* pattern), g) by all the components in each and every workflow instance (*Workflow Data* pattern), and h) by components of the workflow application when they exist in the external operating environment.

Most of these patterns are supported by the scope modifier for the workflow variables and by the persistence modifier for the measurement variables. Table 5.4 summarizes the data visibility patterns supported in MonitA.

Typically workflow management systems do not support multiple instance tasks. In the structural model, data elements are scoped at the workflow level and instance level. We do not include the notion of multiple instance tasks, thereby multiple instances data is not supported. However, a measure can have multiple instances for a workflow instance. The workflow data can be stored in an external system as is the case with the measurement information.

Visibility Patterns	*Workflow Variables*	*Measurement Variables*
Task Data	No	Yes
Block Data	No	No
Scope Data	No	No
Multiple Instance Data	No	Yes
Case Data	Yes	Yes
Folder Data	No	No
Workflow Data	Yes	Yes
Environment Data	Yes	Yes

Table 5.4: Data Visibility Patterns in MonitA

Data Interaction

These patterns define how data is communicated between elements within a workflow application and between a workflow application element with the external environment. Table 5.5 summarizes the data interaction patterns in terms of the interactions between workflow data and measurement data.

Interaction Patterns	*Workflow and Measurement Variables Relation*
Task to Monitoring Task	Global data store
Block Task to Sub-Workflow Decomposition	No
Sub-Workflow Decomposition to Block Task	No
to Multiple Instance Task	No
from Multiple Instance Task	No
Case to Case	Yes
External Data Interaction	Yes

Table 5.5: Data Interaction Patterns in MonitA

The following are the set of internal data interaction patterns:

- *Task to Monitoring Task*: data elements can be communicated between one task instance and another within the same workflow instance.

  In the data association model, we do not consider the specification of the data flow that is integrated with the control flow using the same channels (ports), neither the data flow that is passed through tasks with dedicated channels different than the control flow channels. Instead, the data association model specifies how tasks share the same data elements, which can be accessed in a global data store. An explicit data passing between activities in not required in our approach. The data association model is an approach to data sharing by providing knowledge of the naming and location of workflow variables. It corresponds to the *Global Data Store Pattern*

presented in [RtHEvdA05]. We assume that the workflow implementation is able to deal with potential concurrency issues that may arise when several task instances access the same data element.

Although we model the operations performed by flow entities on data, this specification does not affect the workflow implementation neither the execution semantics of MonitA. In the specification, these operations increase the type of analysis that can be performed (*e.g.,* evaluate all activities that modify a particular variable).

- *Block Task to Sub-Workflow Decomposition*: data elements from a block task instance can be passed to the corresponding subprocess that defines its implementation. This pattern is not supported since the monitoring is defined in terms in terms of atomic tasks.

- *Sub-Workflow Decomposition to Block Task*: data elements can be passed from the underlying subprocess back to the corresponding block task. This pattern is not supported since our level of granularity is monitoring atomic flow entities such as tasks, events, and gateways.

- *Data Interaction from/to Multiple Instance Task*: data elements can be passed from a preceding task instance to a subsequent task which is able to support multiple execution instances. This pattern is not supported since there is no support for multiple instance tasks.

- *Case to Case*: data elements can be passed from one workflow instance during the workflow application execution to another workflow instance that is executing concurrently. This pattern is supported since the data can be captured during a workflow state change or transition and passed indirectly through an external repository (measurement data system).

External data interaction patterns depend of the workflow management system. These patterns define how data elements can be passed between a workflow element with the external environment. We evaluate data interactions when they occur internally in the workflow, this is if they are described in the data association model.

Data Transfer

These patterns consider the mechanisms by which the actual transfer of data elements occurs between workflow elements and describe the multiple mechanisms by which data elements can be passed across the interface of a workflow component (*e.g.,* by value, by reference). Table 5.6 summarizes the data transfer patterns considered for workflow variables and measurement variables.

The following are the set of data transfer patterns:

Transfer Patterns	*Workflow Variables*	*Measurement Variables*
by Value - Incoming	No	Yes
by Value - Outgoing	No	Yes
Copy In/Copy Out	No	Yes
by Reference - Unlocked	No	No
by Reference - With Lock	No	No
Data Transformation - Input	No	Yes
Data Transformation - Output	No	Yes

Table 5.6: Data Transfer Patterns in MonitA

- *Data Transfer by Value*: a workflow application element receives/passes incoming/outgoing data elements by value. This avoids the need to have shared names or a common address space with the component(s) from which it receives them. This pattern is not supported for workflow variables since we do not consider explicit data passing between monitoring activities to workflow activities. In contrast, it is supported by the measurement variables since the monitoring data receives data from external sources. The measurement data is passed explicitly by value between the process activities and the monitoring activities through monitoring events.

- *Data Transfer - Copy In/Copy Out*: a workflow component copies the values of a set of data elements from an external source into its workspace and at the end of the execution the activity copies their final values back. The measurement data is stored temporarily in the analysis functions (monitoring activities) and as soon as they are computed the information is stored in the measurement data store.

- *Data Transfer by Reference*: data elements can be communicated between process components by utilizing/passing a reference to the location of the data element in some mutually accessible location. Nevertheless, MonitA does not transfer data by reference.

- *Data Transformation*: a transformation function is applied to a data element prior to it being passed/passed-out to a workflow component. The process data can be transformed before is passed to the monitoring component. The process data can be transformed after the completion of a task to be passed to the monitoring component.

Data-based Routing

These patterns characterize the manner in which data elements can influence the operation of the control flow of the workflow application. Table 5.7 summa-

rizes the data routing patterns considered to evaluate and specify constraints on the workflow and measurement data.

	Workflow Application	*Measurement Actions*
Task Precondition-Data Existence	No	Yes
Task Precondition-Data Value	No	Yes
Task Postcondition-Data Existence	No	Yes
Task Postcondition-Data Value	No	Yes
Event-based Task Trigger	No	Yes
Data-based Task Trigger	No	No
Data-based Routing	No	No

Table 5.7: Data-based Routing Patterns in MonitA

These patterns are not supported for the workflow application since we do not alter its control flow. The following are the set of data routing patterns:

- *Task Precondition*: data-based preconditions are specified for tasks based on the presence/value of data elements at the time of execution. MonitA can verify if the data has not being created yet. MonitA also supports task preconditions on data value by defining evaluation rules (if-then expressions) for evaluating the value of a field or a variable.

- *Task Postcondition*: data-based postconditions are specified for tasks based on the presence/value of data elements at the time of execution. MonitA verifies postconditions when a task finalizes.

- *Event-based Task Trigger*: a task can be initiated from an external event by passing data elements to the task. MonitA does not influence the control flow of the workflow application. However, a monitoring task (analysis function) is applied statically when a workflow event occurs.

- *Data-based*: a specific task can be triggered when an expression based on data elements in the workflow instance evaluates to true. We do not alter the state of a workflow instance based on measurements evaluation.

5.4 Summary

This chapter has explained the studies we performed to evaluate different elements of our domain-specific language. We evaluated our domain-specific language against a set of design principles typically related with DSLs to present the distinguishable properties of MonitA. A workshop and questionnaire was developed to evaluate the expressiveness and learnability of MonitA

DSL. We identified a set of improvements for the MonitA expressiveness. We incorporated some of these improvements in MonitA, whereas the others were presented as possibilities for future work. We have analyzed workflow data patterns in MonitA to describe the relation between data management in workflow applications and our measurement data.

The following chapter presents the decisions we made to generate the implementation of MonitA specifications. This corresponds to the second part of our solution to tackle the problems for workflow monitoring and analysis.

Part III

Implementing Monitoring and Analysis Concerns Using Generative Approaches

Chapter 6

Rationale and Background

We dedicate this chapter to present the rationale and background behind the implementation strategy of our approach (see section 1.4). As detailed in the introduction, the workflow developers need to build ad hoc infrastructures and abstractions for specific workflow platforms to implement monitoring and analysis (M&A) concerns. This low level implementation creates multiple problems such as a) high development costs, b) tight coupling with specific workflow platforms, and c) complex maintainability due to the amount of scattered code. The second part of our solution to tackle these problems is to generate the implementation of MonitA specifications.

We defined an implementation strategy to create the infrastructure to enact the MonitA specifications on a specific workflow platform. The decision to define this strategy is influenced by the need to reuse MonitA specifications across different and existing workflow languages and engines. This decision is also supported by considering that the implementation of workflow applications typically uses a transformation approach, and that this implementation is not easy to extend with the required monitoring and analysis characteristics.

A MonitA generative infrastructure is required to integrate automatically the MonitA specifications with the implementation of workflow applications. This increases the productivity of application developers and decreases the complexity and time required to implement M&A concerns. This has a favourable impact for workflow analysts since they need less time to materialize their requirements and get feedback to take improvement decisions. This is also beneficial for evolvability since MonitA specifications can be re-generated and re-composed with a new workflow implementation.

Section 6.1 describes the main requirements and challenges considered to develop a MonitA generative infrastructure.

Section 6.2 present the main decisions to create a generative implementation infrastructure.

Section 6.3 introduces a background on Model-Driven Engineering for gen-

eration of code and for separation of concerns at a higher level of abstraction.

Section 6.4 introduces a set of definitions of Aspect-Oriented Programming as compositional mechanisms at the implementation level. We focus on describing the concepts of aspect-oriented workflow languages.

6.1 Requirements for the MonitA Implementation Strategy

This section describes the main requirements and challenges considered to develop the MonitA generative infrastructure. The following challenges are taken into account in our strategy to compose automatically the implementation of M&A concerns with a workflow application:

IC1 *Target diverse execution platforms.* If the workflow developers use the JPDL workflow language and the jBPM workflow engine (or *e.g.,* the BPEL workflow language and the Apache ODE workflow engine) to implement and execute a workflow application, we have to provide a solution to translate the MonitA specifications into *executable code* for that workflow platform. This is important to support the different workflow platforms adopted by companies to automate their business processes, and to support the migration to new workflow platforms due to new expected necessities or alliances between companies.

Specifically, the MonitA specifications have to be mapped to elements in the workflow language that implements the workflow application (*e.g.,* JPDL) and to elements in a general-purpose language (GPL) (*e.g.,* Java) that implements the underlying delegated code (*e.g.,* handlers, services). The links between the source and target elements in the transformation from BPMN to a workflow language must be stored automatically in a traceability model. This traceability information determines the target of the model transformation that generates workflow code from MonitA specifications.

IC2 *Composition of MonitA code.* The translational semantics of MonitA specifications, which can be specified with different tools (*e.g.,* model transformation language), requires taking into account diverse target workflow language constructs. This is because the mapping of MonitA specifications is not equivalent between different target workflow platforms. The workflow generation process (*e.g.,* to transform BPMN to BPEL) and the implementation of the workflow application must stay oblivious of the M&A concerns implementation.

A mechanism to support the composition of M&A concerns with the workflow application should be reused to automate their composition in different locations of the workflow application. To this end, the MonitA code can be

translated into a mechanism for separation of concerns (*e.g.,* aspect code) supported by the workflow implementation language.

IC3 *Customization of workflow code.* If the MonitA specifications cannot be translated into workflow aspects code for a particular workflow platform, the transformation rules must define how to add additional elements to the workflow implementation. Specifically, the workflow implementation must be instrumented to support the interception of workflow data events (*e.g.,* when a workflow variable changes) declared in the MonitA specifications. Moreover, multiple flow entities can operate on the same workflow data and we require to intercept only workflow data events caused by a particular flow entity during the workflow execution (*i.e.,* when the area of expertise is updated by the *Provide Problem Solution* activity). Thus the workflow implementation must also be instrumented to capture which flow entity operates the data entity.

The translational semantics of the MonitA DSL must instrument the workflow specification when the implementation of the activities is defined directly in a workflow language (*e.g.,* BPEL). Otherwise, the underlying workflow code must be instrumented when the implementation of the activities is delegated to code specified in a programming language (*e.g.,* Java).

IC4 *Navigation through workflow data.* The generation of MonitA code requires to establish a mechanism to access and navigate the workflow variables used by a workflow application. A connection link must be established between the workflow variables defined in the data association model and the actual workflow data.

6.2 Design Rationale for the MonitA Implementation Strategy

One of the target requirements defined for developing our DSL is its executability, thus, it is necessary to define a suitable implementation approach to minimize the effort required to implement MonitA specifications.

We adopted a *source-to-source transformation* (preprocessing) approach for MonitA to target existing workflow platforms. We translate MonitA specifications into source code of an existing workflow language (*i.e.,* BPEL, JPDL, Java) to increase the spectrum of potential users that are going to use our DSL (*cf.* section 3.1.1).

Our strategy to generate and compose automatically the implementation of MonitA specifications into the implementation of the workflow application involves multiple generative programming approaches. We have adopted Model-Driven Engineering (MDE), and Aspect-Oriented Programming (AOP) for this

purpose. The usage of these paradigms is hidden for workflow developers who are specifying M&A concerns. Only the MonitA infrastructure developer has to know about them.

We use MDE to generate the implementation of MonitA specifications over diverse workflow platforms (*e.g.,* JPDL workflow language and JPDL engine). A traceability mechanism is incorporated in the MonitA generative infrastructure to compose automatically the generated code with an existing workflow implementation. We use a DSL development system (*i.e.,* OpenArchitectureWare) and model transformations to automate the DSL processor construction, thereby minimizing the usual effort required to implement it.

We use AOP as a mechanism to keep the implementation of MonitA specifications modularized and to compose this modular code automatically with the workflow implementation. We decide to transform the MonitA specifications into workflow aspects in a particular workflow language to preserve the existing workflow applications.

A different approach that could be used for implementing M&A concerns is to override the workflow engine to trigger new workflow events. However, since one of the desirable properties is to target multiple workflow platforms, workflow developers may not have control over the engine. Moreover, this has to be done carefully in order to avoid damaging other concerns. Our approach facilitates application developers to create custom events to monitor and measure the workflow application according to the necessities of workflow analysts. We instrument source code by considering the M&A concerns as a development environment and not to develop debugging environments.

The following sections present the background of our generative implementation strategy: model-driven engineering and aspect-oriented programming.

6.3 Model-driven Engineering

Model-driven engineering (MDE) technology uses language definitions to supply the effective expression and creation of complex platforms at a high-level of abstraction [Sch06]. MDE focuses on shifting the software development process from coding to modeling to decrease the complexity, and to increase the productivity and maintainability. This brings new possibilities to define, implement, maintain, analyze, and reason about software systems at a high-level of abstraction. The aim of MDE is to develop and maintain high-quality software systems with the least possible effort.

MDE defines the structure and behavior of applications within a particular domain using *models* and *model transformations*.

6.3.1 Metamodels, Models and Transformations

The domains that define the structure and behavior of applications are analysed and engineered by means of a metamodel, which is a coherent set of interrelated concepts. The constraints between domain concepts are expressed at the meta-level.

A model is expressed by the concepts defined in the metamodel. Models are the primary assets of MDE to capture designs at a higher-level of abstraction. This contrast with technical documentation which has a fragile connection to the implementation of a software system. The models are meant to be automatically transformed to an executable implementation. Thereby, the effort of producing a new software system can decrease and the maintenance can be reduced to model maintenance. Models can be specified at different levels of abstraction and with different modeling languages (*e.g.,* UML, BPMN).

A model transformation is a set of transformation rules that describe how multiple source models can be transformed into multiple target models [KWB03]. The existing techniques and languages for model transformation enable several different automated activities such as translating models expressed in different modeling languages, generating code from models, refining models, extracting models, composing models, restructuring models, and evolving models. This is the reason to consider model transformations the backbone of MDE [SK03]. The taxonomy of model transformations presented by Mens et al. [MCG04] helps in deciding which model transformation approach (*i.e.,* vertical, horizontal, rephrasing, translation) is best suited to deal with a particular problem.

6.3.2 MDE and DSLs

Model-driven engineering is highly related to the field of domain-specific languages and both have complementary strengths. For example, whereas models are typically represented with a graphical notation, DSLs use typically a textual representation. Moreover, the models describe structures that DSLs complement by describing their business logic. MDE incorporates the experience from work on code generation and domain-specific languages to provide a systematic approach to the construction of modeling languages that can be integrated in the software development process.

The increase of productivity can be achieved by code generation. The application code is replaced by DSL programs that capture the variability in a software system and by code generators that produce the application code automatically. The DSL defines the architecture to compose specialized application code with a software system. An advantage of a DSL is the possibility for targeting a generator to a new architecture or platform, without changing the DSL programs. Specifications written in a DSL can automatically generate system families by using generative software development [Cza04] [GS03].

6.3.3 Traceability Models

Model transformations may store the links between their source and target elements for multiple purposes such as a) analyzing how the changes done to one model affect other related models, b) synchronizing models, c) mapping the stepwise execution of an implementation back to its high-level model, and d) determining the target of a transformation [CH03].

The following are the common approaches to incorporate traceability on model transformations:

- The transformation platform provides dedicated support for traceability. In this approach, the traceability links are created automatically thereby a traceability model is generated at a low cost since it does not require additional effort from the developers. Nevertheless, the level of granularity of the traceability information differ among different transformation engines since the traceability metamodel is fixed. Certain transformation engines may offer support to limit the amount of traceability information that is generated with mechanisms to control which traceability links must be created.

- The traceability rules are encoded in the transformation model. The traceability rules are encoded by developers who can use the same mechanisms used for adding transformation rules to generate a regular output model of the transformation. The main advantage is that the developer can customize the traceability metamodel independently of the transformation engine. One disadvantage is that the model transformation gets polluted with the transformation rules, which require additional effort from the developers.

- The traceability rules are added to the original transformation using high order transformations (HOTs). A HOT is a transformation that generates another transformation. The main advantage of this approach is that the traceability rules are automatically generated.

A traceability model can be also produced manually, however, it is beneficial to incorporate the traceability generation into the model transformations to automate the traceability model creation.

6.4 Aspect-Oriented Software Development

In the Aspect-Oriented Software Development (AOSD) [JN04] approach, a modularization way in the software development cycle is applied by means of crosscutting concerns separation. In addition, AOSD allows describing the relation of multiple concerns to the system, and provides the mechanisms to compose them into a coherent product [GBNT01].

Aspect-oriented Programming (AOP) facilitates introducing crosscutting behavior to existing applications in a modularized way [KLM+97]. The crosscutting mechanisms provided by AOP break with the *"tyranny of the dominant decomposition"* [TOHJ99].

AOP introduces a unit of modularity named *aspect*. An aspect module consists of pointcuts, joinpoints, advices, and bindings. Pointcuts define the specific points during the process execution where additional behavior has to be added. For example, different pointcuts in object-oriented programs are method calls, constructor calls, field read/write. A joinpoint indicates where the new behavior is introduced before, after, or around the interception point defined in the pointcut. For example, it is possible to select related method execution points by pattern matching, by the type of the return values of these points, and by the type of their parameters. An advice contains the extra behavior that requires to be inserted to the base program. The advice code is executed when a joinpoint is reached in the set specified by a pointcut. Finally, bindings are used to combine the behavior of the base program with the new behavior defined in the advice.

6.4.1 Aspect-Oriented Programming Languages

AspectJ [Asp] is the most popular aspect-oriented programming language created as extension to the Java programming language. AspectJ has a joinpoint model, a pointcut language, and an advice language.

The joinpoint model of AspectJ defines points in the execution of object-oriented Java programs such as method calls and field reading/writing. The pointcut language provides a predefined set of pointcut designators such as *call* that selects a set of method call joinpoints, *execution* that selects a set of method execution joinpoints, and *get and set* that select read/write field access joinpoints. The advice language of AspectJ is the same as the programming language (*i.e.,* Java) and defines the advice types before, after, and around. AspectJ supports a static weaving approach, in which the weaver transforms the byte code of Java classes to integrate aspects. The AspectJ weaver generates classes that have plain Java byte code, thereby they can be interpreted by any Java interpreter.

6.4.2 Aspect-Oriented Workflow Languages

Although most available aspect-oriented languages are extensions to programming languages, aspect orientation is also applicable for workflow languages. The aspect-oriented workflow languages introduce the concepts of aspect-oriented software development to workflow languages in order to improve the modularity of crosscutting concerns in the workflow specifications [CM06] [Cha07].

The modular specification of crosscutting concerns within a workflow application, also named workflow aspects, involves the different perspectives of workflow applications (*i.e.,* functional, informational, behavioral, operational, and organizational). The workflow logic is encapsulated in a *process module*, whereas the the non-functional concerns are encapsulated in *aspect modules*. Workflow aspects provide a view on how a certain concern is handled within different workflow applications. Thus, the workflow developer has to modify only one aspect module when he needs to modify a crosscutting concern.

The aspect-oriented workflow languages provide concepts for crosscutting modularity such as pointcuts, joinpoints, advices, and aspects. These languages also provide a weaving mechanism to compose workflow aspects with workflow applications.

Aspects. A workflow aspect contains pointcut and advice declarations and defines the activities, the variables, the transitions, the participants declaration, and the application declarations that implement a crosscutting concern. Workflow aspects are complementary to programming aspects. For example, whereas workflow aspects can be specified in scenarios at the workflow level (*e.g.,* workflow-level data persistence), programming aspects can be used for a Java-based implementation of an artifact to persist the data. The workflow aspects developer is responsible to take care on the effects that the aspect can cause to the different workflow perspectives (*e.g.,* control flow, data flow).

Joinpoints. Joinpoints specify a point in the execution of the workflow application when the advices should join. The advices and the workflow application are defined separately. There are two types of joinpoints in aspect-oriented workflow languages: activity joinpoints and internal joinpoints. Activity joinpoints (workflow-level joinpoints) correspond to the execution of activities to capture their start and completion. Internal joinpoints (interpretation-level joinpoints) capture internal points during the execution of an activity to support a crosscutting concern. Typically, the aspect-oriented workflow languages only support activity joinpoints, thus other fine-grained points are not exposed to be capture. For example, internal joinpoints such as as the creation or termination of a workflow instance, the assignment or change of a participant to an activity, and the operating of workflow variables are needed to add a crosscutting concern.

Pointcuts. Pointcuts specify a selection of related joinpoints. For example, to select a joinpoint where an activity is executed. This selection must support quantification [FF00] to span different elements of a workflow application. A limitation of current aspect-oriented workflow languages is that they do not provide support to enable the direct joinpoint selection according to the workflow perspectives such as behavioral, organizational, informational and operational. For example, for allowing a pointcut to select all joinpoints where an activity modifies certain variable.

Advices. An advice is a workflow activity that implements the crosscut-

ting functionality that needs to be executed at the set of joinpoints specified by a pointcut. An advice activity can be executed before, after, or instead of the selected joinpoints. It is also possible to define different orders of execution (*e.g.,* sequence, parallel) between joinpoint activities and the advices according to the workflow control patterns [RAvdAM06]. The advice language is the same as the workflow language to avoid incompatibilities in the workflow specification between workflow developers. The aspect-oriented workflow languages can extend the workflow language with special constructs to collect the context, to access the meta-data, and to avoid incorrect behaviors about the current joinpoint activity. These extensions include constructs to define advice precedence and to access the input and output variables, the participants, the applications, the activity name, the activity type related to the joinpoint activity and its parent workflow specification. The advice may also provide the constructs to specify if a workflow aspect is applied to all workflow instances of the workflow application or only to some of them.

Weaving. The composition of workflow aspects with workflow applications can be performed before deploying the workflow application (statically) or during the execution of the workflow (dynamically).

In the static approach, the composition tool takes the workflow application and the workflow aspects as inputs and generates an instrumented workflow application as output. The generated workflow application can be deployed in any workflow engine that supports the workflow language, however, the composition cannot be performed at runtime to incorporate/remove un-anticipated workflow aspects. The workflow aspects are not first-class runtime entities in the workflow engine but only in the workflow specification.

In the dynamic approach, interpretation of the workflow engine is modified to check for workflow aspects when executing an activity. The composition can be performed at runtime what improves the flexibility of workflow applications, however, the composition is dependent of the workflow engine breaking the portability of the workflow application. The workflow aspects are first-class entities at the workflow specification as well as the workflow execution level.

Whereas the static approach requires to add custom hooks to the workflow specification, the dynamic approach requires to modify the workflow engine to support these custom hooks.

The following aspect-oriented workflow language was used in our case studies to implement crosscutting concerns in BPEL workflow applications.

Padus Language

Padus allows introducing crosscutting behaviour to an existing BPEL process in a modularized way [BVJ+06].

The workflow aspects in Padus are written in separate XML files. The Padus language facilitates defining the specific points during the workflow ex-

ecution where additional behaviour has to be added to the BPEL specification. These points can be selected as needed by using a logic *pointcut* language. The Padus *weaver* can be used to combine statically the behaviour of the core workflow application with the behaviour specified in the workflow aspects. Similar to traditional aspect-oriented systems, the new behaviour can be introduced by inserting it before or after certain *joinpoints* defined by the pointcut or it can replace existing behaviour by using an around *advice*. The Padus language introduces the concept of *in advice* to add new behaviour to existing process elements and provides an explicit deployment construct to specify aspect instantiation to specific workflow specifications. The advice code contains the extra behaviour that should be inserted, which is specified using standard BPEL elements.

Padus contains a construct named *using* to specify the workflow global information required by the advices. This information corresponds to a) the *namespaces* of the web services to be invoked, b) the *partner links* that define the interaction between services, and c) the BPEL *variables* that are global to the workflow application. AO4BPEL is another aspect-oriented workflow language that supports the modularization of crosscutting concerns in BPEL [CM07]. The implementation of AO4BEL is based on a dynamic weaving approach.

Despite that current implementations of aspect-oriented workflow languages are applicable to domain-specific workflow languages (*e.g.,* BPEL), the concepts of aspect-oriented workflow languages can be generalized to general-purpose workflow languages.

6.5 Summary

This chapter has explained the elements considered to develop a MonitA generative infrastructure. We presented the requirements for implementing MonitA specifications such as a) target diverse workflow platforms, b) modularize the generated MonitA code, c) instrument the workflow applications, and d) access and navigate the workflow data. We presented how the adoption of multiple generative programming approaches such as MDE and AOP facilitates generating and composing automatically the implementation of MonitA specifications into the implementation of a workflow application. We have discussed how the usage of these paradigms is hidden for workflow developers who are specifying M&A concerns.

Our goal for the next chapter is to describe in detail the architecture and strategy to create a new MonitA generative infrastructure to target MonitA specifications into a new workflow platform. The implementation requirements presented in this chapter are solved in the next one.

Chapter 7

MonitA: The Generative Implementation Strategy

This chapter presents the generative strategy that we have defined to implement and execute the MonitA specifications. Despite their platform independence, these specifications require to target a particular workflow platform (*e.g.,* BPEL platform, JPDL platform) to be executable. The main goal in our strategy is to generate and compose automatically the implementation of MonitA specifications into the implementation of the workflow applications. Consequently, MonitA specifications can be executed in the same workflow platform (*e.g.,* BPEL-workflow language and BPEL-engine) than the workflow application. In this way, our workflow monitoring and analysis architecture can be adopted and integrated with existing WFMS.

Figure 7.1 illustrates the main elements involved in our MonitA execution platform. This figure presents an example for executing MonitA specifications into two different workflow platforms (*i.e.,* JPDl, BPEL).

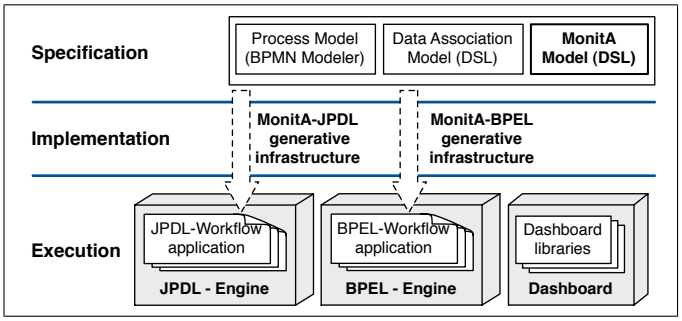


Figure 7.1: MonitA Execution Platform.

The specification part was explained in chapter 4, by considering the specification of different models such as: a process model, a data types model, a data association model, a measurement data types model, and a MonitA

model. In the execution part, a workflow engine is used to execute the workflow application that is instrumented with monitoring and analysis (M&A) concerns. The goal of this chapter is to explain a generic strategy to create a MonitA generative infrastructure (the arrow in Figure 7.1) required to derive executable workflow applications instrumented with M&A concerns.

The automatic generation and composition of MonitA specifications into executable workflow applications is a complex task that requires: a) to implement the workflow generation process (*e.g.,* to transform BPMN into BPEL) taking into account the access to workflow data and the traceability of BPMN to workflow code mappings, b) to generate MonitA code taking into account the modularization of M&A concerns and the management of measurement data, and c) to compose automatically M&A concerns with the workflow application.

This chapter also illustrates the different types of developers required to take abstract MonitA specifications to concrete implementations: application developers and infrastructure developers. Although we briefly describe the role of application developers, the focus of this chapter is on the role of MonitA infrastructure developers. The MonitA infrastructures developers use our generative strategy to create a generative infrastructure required to automate the implementation of M&A concerns.

Section 7.1 describes the process that application developers have to follow to specify and execute M&A concerns for a workflow application.

Section 7.2 presents the architecture and strategy that we have defined to generate and compose automatically MonitA code into an existing workflow application. We present our general strategy that can be applied to target diverse workflow languages and engines. The instantiation that MonitA infrastructure developers perform on our strategy to target concrete workflow platforms is presented in chapter 8.

Sections 7.3, 7.4, and 7.5 detail the main elements within the three blocks defined in the architecture of a MonitA generative infrastructure. The instantiation of these elements creates a new MonitA generative infrastructure to target MonitA specifications into a new workflow platform.

7.1 M&A Analysis Concerns Execution

The monitoring and analysis requirements defined by workflow analysts have to be implemented by MonitA application developers.

Figure 7.2 summarizes the process that an application developer must follow to specify and implement M&A concerns using MonitA.

An application developer starts by selecting a workflow application and defining the workflow variables associated to its activities. This association between activities and workflow variables is done only once. An applica-

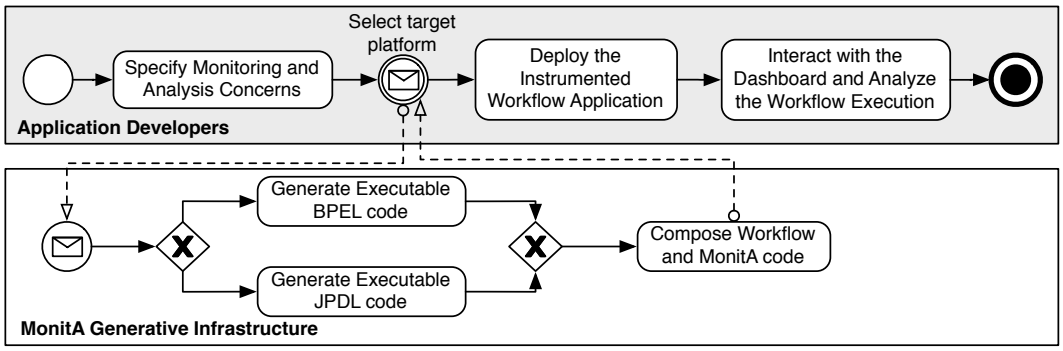


Figure 7.2: Specification and Implementation of M&A Concerns using MonitA.

tion developer specifies the M&A concerns by using the DSL editor, and also specifies the new measurement data types required to support these concerns. The MonitA application developer has to select an existing MonitA platform (*e.g.,* MonitA-BPEL platform) to automatically generate and compose M&A concerns into an existing workflow application (*e.g.,* BPEL workflow application). The MonitA platform generates and composes automatically the MonitA code with the workflow application code that is implemented with a workflow language (*e.g.,* BPEL workflow language). Consequently, the code of the workflow application is instrumented with M&A concerns and can be executed in a corresponding workflow engine (*e.g.,* BPEL engine). Then, the application developer has to deploy the instrumented workflow application into a particular workflow engine to start the workflow execution and the monitoring and analysis of the workflow instances. Once the workflow application instrumented with M&A concerns is under execution, an application developer interacts with the dashboard to visualize the monitoring information, and uses the existing libraries to customize his queries (*e.g.,* queries on demand).

Now, suppose that the workflow application has to be executed in another workflow platform (*e.g.,* JPDL workflow language and JPDL engine) that is not supported by MonitA. Then, the application developers have to contact the MonitA infrastructure developer to create a new MonitA generative infrastructure. This is required to execute M&A concerns with workflow applications into this new workflow platform. The following sections describe the strategy to target M&A concerns into new workflow platforms.

7.2 Architecture for Creating a MonitA Generative Infrastructure

This section presents the architecture defined for creating the generative infrastructure (the arrow in Figure 7.1) required to automate the implementation of

MonitA specifications into a workflow application. It also presents a strategy to target MonitA specifications into diverse workflow platforms.

7.2.1 Functional Decomposition Viewpoint

A MonitA infrastructure developer creates the generative infrastructure required to automate the implementation of MonitA specifications for a new workflow platform. Each MonitA infrastructure developer focuses on the instantiation of the components defined in our architecture for the generative infrastructure. For example, he creates the artifacts (*e.g.,* model transformations) required to transform and compose automatically MonitA specifications into JPDL workflow applications (see Section 8.2).

Figure 7.3 illustrates the main elements involved in the architecture for the MonitA generative infrastructure.

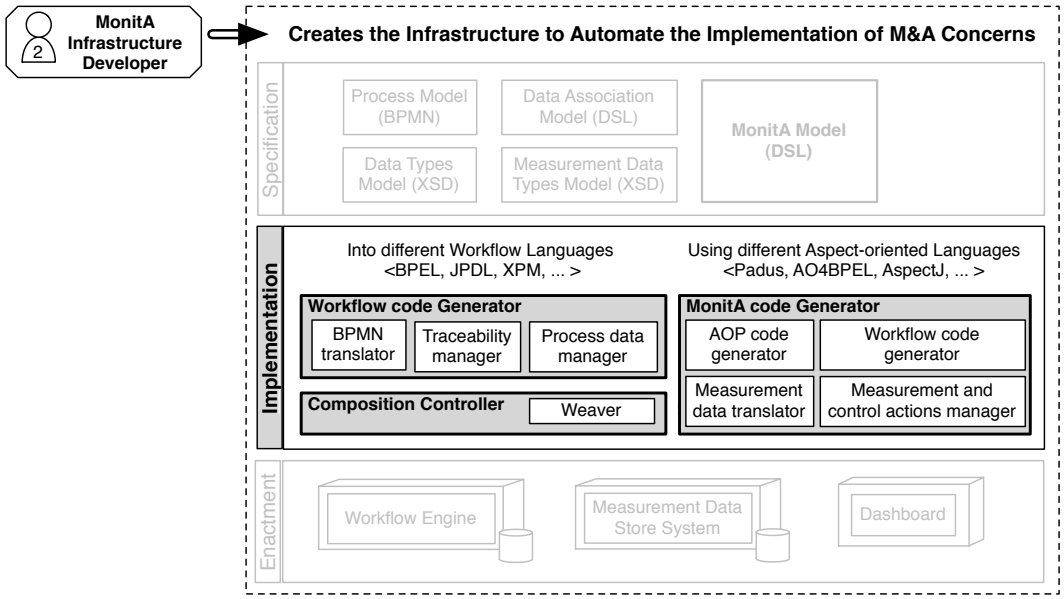


Figure 7.3: Architecture for the MonitA Generative Infrastructure.

The main components to create a MonitA generative infrastructure are: the workflow code generator, the MonitA code generator, and the composition controller. The workflow code generator contains modules that define how to: translate BPMN models into different workflow languages, store the link between source (BPMN) and target (workflow) elements, and access workflow data. The MonitA code generator contains modules that define how to: modularize the implementation of MonitA specifications, translate measurement data, manage measurement and control actions, and instrument the workflow application. The composition controller defines how to compose the generated MonitA code with an existing workflow application implementation.

The modules defined in these components differ for each workflow platform that a MonitA infrastructure developer is targeting.

7.2.2 Generative Strategy

Figure 7.4 illustrates the set of models and model transformations used in our strategy to generate the implementation of M&A concerns into a particular executable workflow implementation.

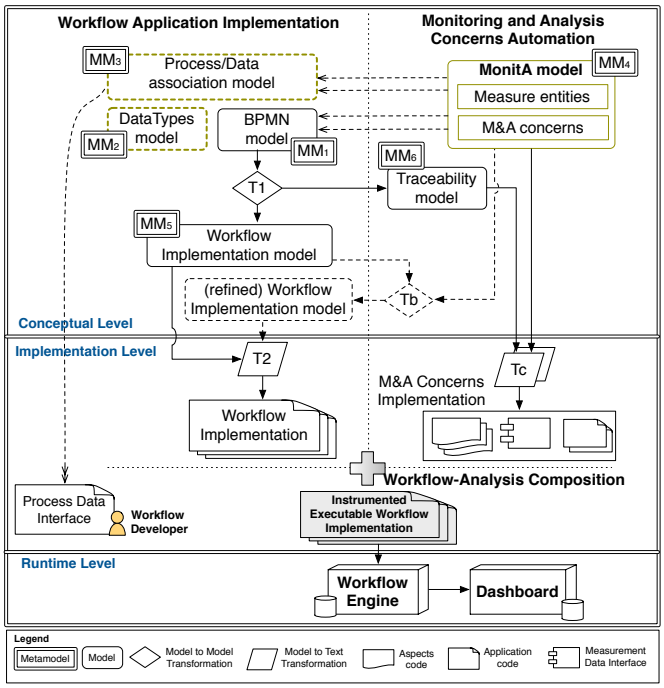


Figure 7.4: MonitA Generative Strategy.

In this figure, the elements in the MonitA generation strategy can be framed in three components according to our MonitA implementation architecture.

The first component manages the workflow generation process to establish the correspondence between elements of a process model and its workflow implementation (see Section 7.3). We complement a `BPMN model` by declaring, in a *data association model*, the data entities used by the workflow application. A partial workflow implementation is generated from a process model. We extended this workflow generation process by defining a set of model transformations (`T1` and `T2`) to generate a) the `workflow implementation` code of the target workflow platform, and b) a `traceability model` to trace the mapping done in the transformation.

The second component generates workflow executable code from the MonitA specifications (see Section 7.4). M&A specifications are rooted in a `MonitA`

`model` referring to *conceptual workflow specification models*, which describe flow entities and data entities in the workflow application. Our generative strategy defines the creation of a model transformation (`Tc`) to generate the M&A code (workflow, aspect, data management, application) for implementing the M&A concerns. This model transformation uses the `traceability model` to automatically obtain the relations between the MonitA specifications and the actual workflow implementation.

Finally, the third component customizes the instrumentation of the workflow implementation by adding custom MonitA code automatically (see Section 7.5). The composition between the generated M&A code and the workflow application code is performed automatically at the implementation level by using the weaver engine of the used aspect language. The resulting code artifacts correspond to an `executable workflow implementation instrumented with M&A concerns`. The resulting code can be executed in a existing workflow engine and interacts with external systems (dashboard, measurement data store) to manage measurement information.

The elements of each of these three phases are characterized according to their role in the infrastructure: specification artifact, transformation artifact, and generation artifact.

7.3 Controlling the Workflow Generation Process

This section details the *workflow code generator* component illustrated in Figure 7.3.

There are existing tools to translate BPMN process models into a workflow language (*e.g.,* BPMN2BPEL [BPM]), however, the MonitA infrastructure developer has to create a new transformation to manage the traceability between BPMN to workflow code mappings. This section presents how to control the workflow generation process to transform BPMN into a workflow language (*e.g.,* BPEL) in order to access the workflow data and to generate a traceability model containing the BPMN to workflow code mappings.

In those cases where a BPMN process model is not provided, the M&A concerns can be specified directly in terms of elements of the workflow application (*e.g.,* BPEL activities). Nevertheless, we consider that the typical life cycle of a workflow applications starts by specifying a process model using a high-level notation.

The following sections detail the three modules contained in the *workflow code generator* component (Figure 7.3): BPMN translator, traceability manager, and process data manager.

7.3.1 Transforming BPMN Models into Executable Workflows

A *process model* is created conform to the BPMN metamodel [The08] (MM1 in Figure 7.4). A description of the elements involved in the BPMN metamodel was presented in section 2.1.2. A model transformation (*T1 in Figure 7.4*) generates a) a specific workflow platform model, and b) a traceability model to trace the mapping done in the transformation.

In some cases the target workflow platform provides a workflow implementation model (MM5 in Figure 7.4), which has a direct mapping with the actual workflow implementation code. When an implementation model is provided, then the process model is transformed into a workflow implementation model. Otherwise, the process model is translated directly to workflow implementation code. This transformation (T2 in Figure 7.4) generates the workflow definition file (*e.g.,* JPDL file) and the underlying application code (*e.g.,* Java code).

A workflow developer uses a DSL to specify the workflow variables used by the workflow application and their association with the flow entities. This data association model (MM3 in Figure 7.4) complements the BPMN process models with a data view required to model a workflow application to be evaluated from a high-level of abstraction. The data types model associated with the workflow variables is specified according to an XML schema (XSD) metamodel created from the XSD specification (MM2 in Figure 7.4). The data association model is specified according to the data association metamodel provided by MonitA (MM3 in Figure 7.4).

7.3.2 Managing Traceability

We store the links between source and target elements in the model transformation for determining the target of other model transformations.

We use the TraceComponent metamodel (MM5 in Figure 7.4) used by the transformation language of the OpenArchitectureWare models framework [Ope] to link process model elements with workflow implementation ones.

The traceability models created with this metamodel store a) a set of relations between source elements (BPMN model) and target elements (workflow implementation), b) the references to these elements, and c) the type of transformation. Thus, information such as names and types of model elements can be queried from both models to establish their correspondence. A traceability model can be generated in a model-to-model transformation as well as in a model-to-text transformation.

To build the traceability model, the MonitA infrastructure developer, who is implementing the transformations, must specify explicitly where to create a trace in the transformation. These traces are specified by using the methods and extensions provided by the traceability mechanism. For example, to create

a trace the workflow developer must invoke the method *createTrace(source, target,"m2m")* in the transformation code. This method invocation adds an *item* element (M2MTraceItem) with the information provided. The extraction of data from the traceability model requires the navigation through these items. The traceability mechanism resolves the references to the source and target models, thereby the properties of the model elements can be queried.

As in any specification, the reference to the metamodels, the source model, and the target model must be well defined to localize the elements referenced in the traceability model. The MonitA infrastructure developer must know all the involved metamodels in order to apply the object conversions (*e.g.,* castings) required to access the elements information in the traceability model.

All the details about the workflow implementation have to be exposed. However, the workflow generation process only contains the basic information (*e.g.,* naming convention) and not the code added manually by workflow developers to implement the activities. Thus the workflow developer has to complement this information with the actual implementation of the activities (*e.g.,* data, invocations, operations). The data view is covered by our approach, which models a projection of workflow data and their association with flow entities.

When the traceability model cannot be generated automatically and when it does not contain all necessary information about the workflow implementation, this traceability has to be described manually by the workflow developer.

7.3.3 Accessing Workflow Data

We use web services and method invocation implementations for exchanging one-way messages to access workflow data and to satisfy our MonitA requirements. All the workflow variables used by the workflow application must be exposed to be shared by the workflow developers who are specifying M&A concerns (process data interface in Figure 7.4).

The only requirement that has to be implemented to access the workflow variables is the *read* operation. The workflow data access is simplified since we do not want to alter the state of the workflow application from the MonitA specifications during execution.

7.4 Generating the M&A Code

This section details the *MonitA code generator* component illustrated in Figure 7.3. This section presents how to generate MonitA code taking into account the modularization of MonitA specifications and the management of measurement data.

The MonitA specifications correspond to a model that conforms to the MonitA metamodel (MM4 in Figure 7.4). The MonitA models are transformed into MonitA code specified in the workflow language used by the workflow application (*e.g.,* JPDL). The MonitA models also generate MonitA code, specified in a programming language (*e.g.,* Java), that is delegated by the workflow application (*e.g.,* notification actions).

A model transformation (*Tc in Figure 7.4*) generates the executable MonitA code in a specific workflow platform. This model transformation uses as input the information provided by the MonitA models and by the traceability model created in the workflow generation process. This traceability model is used to automatically obtain the relations between the elements in the process model and in the actual workflow implementation. The output of this model transformation comprises: a) aspect code to modularize and compose the M&A concerns, b) workflow code with the M&A concerns implementation, c) application code with a representation of measurement data, and d) application code (*e.g.,* web services) required to manage the measurement information in the persistent system. These artifacts are explained in the next subsections.

A model transformation (*Tb in Figure 7.4*) automatically instruments the workflow implementation model by adding the workflow elements (*e.g.,* events) required to support the MonitA specifications. The workflow implementation model is instrumented since this model is directly transformed to code and avoids the necessity to add the workflow elements to the generated workflow implementation (*e.g.,* processdefinition.xml). This model transformation applies only when there is not an aspect-oriented workflow language that manages crosscutting concerns in the target workflow language. Thus the refinements done to the workflow implementation model instrument the workflow application with monitoring events, which delegate its execution to application code. This code can be then used to introduce M&A concerns by using a regular aspect-oriented language. The input for this model transformation consists of the workflow implementation model and the MonitA model, whereas the output is a workflow implementation model instrumented with monitoring concerns.

The following sections detail the four modules contained in the *MonitA code generator* component (Figure 7.3): AOP code generator, workflow code generator, measurement data translator, and measurement and control actions manager.

7.4.1 Transforming MonitA Specifications into AOP Code

We use AOP as a composition mechanism at the implementation level to facilitates the automatic addition of M&A concerns in multiple points of the workflow application. This facilitates to bring the MonitA specifications up to date each time the workflow implementation changes. In addition, the

generated MonitA code remains modularized from an existing workflow implementation. Thus, the existing workflow implementation can be maintained independently of the generated MonitA code. Moreover, an evaluation of the impact of the MonitA code on the workflow implementation can be performed when the M&A concerns have complex interactions and realizations at the implementation level.

Our workflow monitoring and analysis generation process generates an aspect for each MonitA specification, which is composed of monitoring, measurement and control concerns. In general, the *monitoring events* in MonitA are translated into pointcut and joinpoint elements, which describe how the aspect has to interact with a workflow entity (*e.g.,* activity). The *analysis functions* in MonitA are translated into advices containing the implementation of the measurement and control actions. The implementation of these actions is generated into code using the workflow language of the target workflow platform.

A data association model offers a representation of the workflow variables that can be accessed and shared. The interception of custom workflow data events is materialized by using aspects.

There are scenarios where using an aspect-oriented workflow language is not enough to customize the workflow implementation to support the interception of workflow data events. This is, when the activities implementation is done in workflow code, and when the aspect-oriented workflow languages do not offer support to intercept events in terms of data or functions inside the activities.

When the workflow data is represented in the application code, the accessor methods associated with the workflow variables can be intercepted by an aspect-oriented language (*e.g.,* AspectJ). When the monitoring specification is done in terms of workflow data events, the joinpoint in the generated aspect intercepts the invocation of a class that contains the definition of the data (*e.g.,* Problem class in the trouble ticket scenario). As soon as the data entity class is invoked, the aspect captures the information (*e.g.,* performer, instance) required to execute the monitoring and control concerns.

7.4.2 Transforming MonitA Specifications into Workflow Code

Different elements of a MonitA specification are translated into workflow code depending on the target workflow platform. The normal case in our generative strategy is to translate monitoring events into pointcut elements in the aspect-oriented workflow language supported by the workflow language. Nevertheless, not all workflow languages provide an aspect-oriented workflow language (*e.g.,* JPDL). In these cases the workflow specification must be instrumented with additional flow elements that delegate the implementation of M&A concerns to application code.

The MonitA generation process generates an object class (*e.g.,* ActionHandler) to capture execution information when the implementation of MonitA actions is delegated to an object instead of being executed directly by the workflow language. Thus the monitoring events can be translated into pointcut elements of a general-purpose aspect-oriented language, which intercepts the methods associated with these actions.

In this way, the implementation of the MonitA actions can be generated directly into workflow code or into the underlying application code.

7.4.3 Managing Measurement Data and Control Actions

The MonitA code generator generates a class (*i.e.,* ActionManager) to manage the notification actions that can be applied (*i.e.,* email notification, log creation, alarm visualization) and to connect with external systems (*e.g.,* dashboard).

A *DataManager* class is generated to access and manage the measurement data from our measurement data store system.

7.4.4 Transforming Measurement Data

Each complex data type associated with the measurement variables specified in the M&A concerns is translated into a Java class to provide a representation of the data that can be accessed and shared. We use a web services implementation for performing the set of operations (*i.e.,* read, update, create, delete) provided to manage the measurement data. The measurement data is identified through the workflow application identifier, the worflow instances identifier, and in certain cases with a flow entity identifier.

MonitA supports the specification of monitoring events in terms of workflow data events to support a more advanced analysis activity (*i.e.,* in terms of workflow data). These specifications require to modularize monitoring concerns at the level of local variables (workflow variables used by a particular activity) and not only at the workflow instance or activity level. Nevertheless, the workflow data events can be triggered by multiple flow entities and also by external systems. This is problematic since the workflow developer is interested only in the interception of workflow data events when they are triggered by a specific flow element (*e.g.,* activity). Figure 7.5 illustrates this scenario.

In this scenario, the activities A and C update the variable x, whereas the activity B uses the variable y. An external system also modifies the variable x. However, a workflow developer is only interested in intercepting when the variable x is updated by the activity C. The aspect-oriented languages (*e.g.,* AspectJ) and the ones specialized in workflows (*e.g.,* Padus) do not support the modularization of crosscutting concerns at the level of local variables.

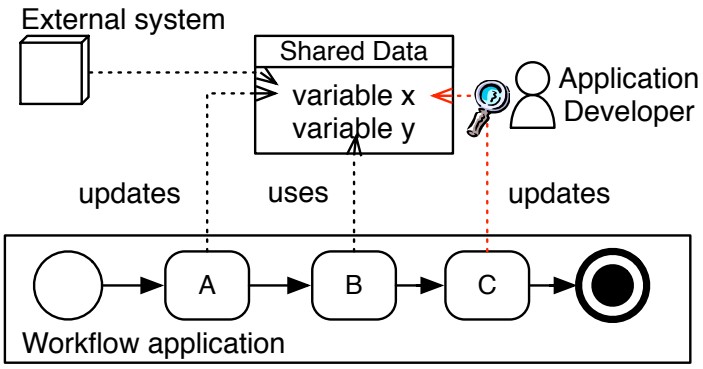


Figure 7.5: Intercepting data interactions.

Thus a monitoring concern defined in terms of data related events cannot be encapsulated and reused.

We defined a set of possible solutions to allow the interception of data related events triggered by specific flow entities. These solutions can be applied depending on the technologies and characteristics of the workflow platform in which the workflow application is automated.

Solution 1: Instrument the workflow implementation and use aspects to intercept the generated application code

This solution can be used in workflow platforms where the implementation of the activities is delegated to application code. In this case, the monitoring and analysis specification generates a handler to capture the execution information and an aspect that intercepts when the execute method or the class attributes are accessed in the handler. This aspect also contains the MonitA code that has to be added when the activity involved finishes its execution. Figure 7.6 illustrates this scenario.

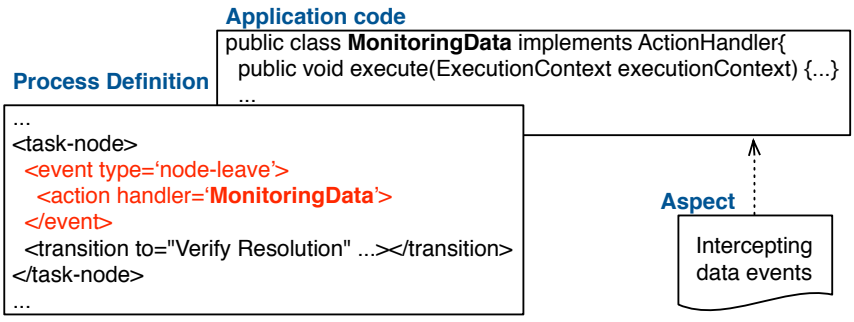


Figure 7.6: Intercepting data related events in the application code.

This solution allows the aspect to recover (from the handler) the information of the activity such as instance, process name, and process identifier. This

information is required to execute the M&A concerns. This solution also allows to add information about the activity, which is required by the measurement and control aspect. We use this solution to manage control flow events in the MonitA platform created for JPDL applications (see Section 8.2).

Using an aspect for intercepting the handler is enough to detect a workflow event in the class, in a method, or in a class attribute, but not in local variables. A drawback of this solution is that it allows to intercept an operation performed on a workflow variable, but not to intercept particular operations performed by different flow entities (*e.g.,* intercepts only when the variable x is updated by the activity C). Another drawback is that the advice contained in the aspect has to be executed when the activity finalizes and not when data is changed. This is problematic since there could be a big gap of time between the data related event and the activity finalization.

Solution 2: Annotate the workflow implementation

Another solution to allow the interception of workflow data events in finer-grained execution points is to annotate the workflow definition or its underlying implementation. Figure 7.7 illustrates this scenario.

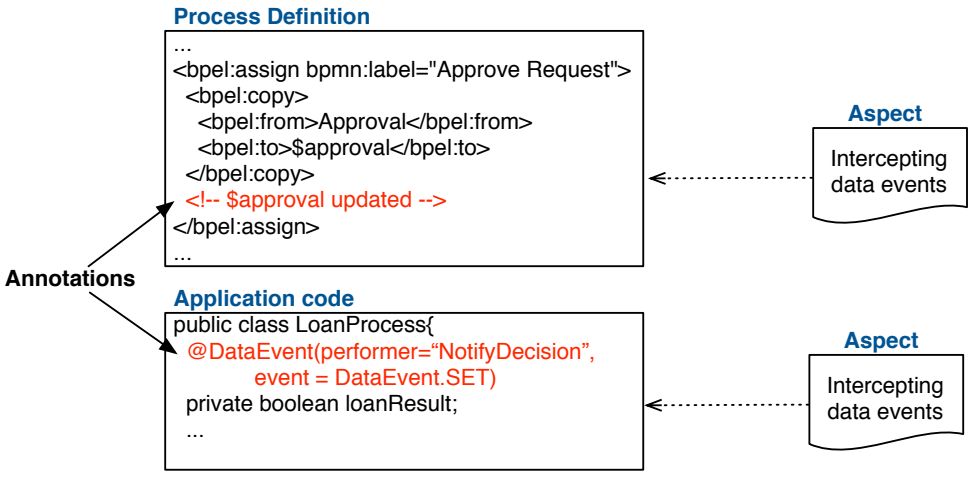


Figure 7.7: Intercepting workflow data events using annotations.

In this case, the MonitA specification has to generate an aspect to intercept these annotations. Although this approach would facilitate the interception of fine-grained execution points, the following drawbacks have to be considered:

- The workflow code is contaminated with annotations complementing the MonitA code. This is not a problem if the annotations are generated and added automatically to the workflow code. However, this requires a very detailed traceability model to know exactly where to instrument workflow code with data interaction annotations.

- If the annotations are added manually, the workflow developers require detailed knowledge of the workflow implementation to add the annotations in the right location. The workflow variables are not always explicit in the control flow specification but in the underlying implementation. Thus, this solution can be used in workflow platforms where there is access to the underlying implementation of the activities.

- An additional formalism has to be defined to add annotations that allow to describe the relevant functionality that is executed. Nevertheless, this formalism is already provided by the MonitA specifications, thus, an additional formalism introduces synchronization and maintainability problems.

- The workflow specification can restrict certain workflow elements to add additional code into it (*e.g.,* into an *assign* in BPEL). If the monitoring code is added outside the workflow element, then it behaves as the workflow definition instrumentation presented in the first approach.

This is why we do not consider this as a viable solution since it would break our base objectives.

Solution 3: Use aspects to intercept the data representation entities in the underlying implementation

This solution can be adopted where there is access to the underlying implementation of the activities. In particular to the classes that contain a representation of the data entities. Figure 7.8 illustrates this scenario.

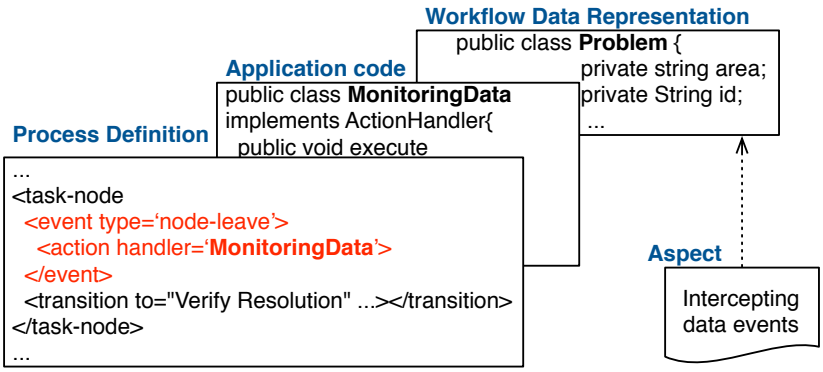


Figure 7.8: Intercepting workflow data events in the data entities representation code.

When there is access to the underlying implementation, the pointcut intercepts when the data is changed in the data representation entity (Java class). Additionally, the aspect requires to recover the information of the activity

(*e.g.,* instance, process identifier) that operated on the variable. However, this information is not contained in the data entity class (*e.g.,* Problem).

The main drawback with this solution is that it is not possible to intercept workflow data events in the underlying implementation when there is no access to it (*e.g.,* when the activities are implemented by external services).

Solution 4: Extend the workflow languages and aspect languages to manage workflow data related events

A workflow language can be extended to provide the generation of events in terms of the workflow variables in such a way that these state changes can be intercepted and processed manually or by external systems (*e.g.,* AOP technology). At the same time, the workflow aspect languages can be extended to support the interception and composition of crosscutting concerns in intermediate points of the activities.

The main drawback with this solution is that the workflow applications have to be re-implemented to incorporate the management of workflow data events in the workflow code. If these extensions are incorporated in the workflow application, they are not supported by existing workflow engines.

None of these solutions satisfy the requirements for implementing custom monitoring events (*e.g.,* data events triggered by specific flow entities) in workflow applications. Thus, a new flexible mechanism must be created to allow the interception of custom monitoring events in a multi-platform workflow engine. In our approach, we used MonitA for the specification of high level monitoring events and an AOP pointcut language for capturing these events in the workflow implementation. However, most AOP languages do not offer the ability to reason about a function's (activities') body. We require to define a different approach (*e.g.,* a new compiler) to support the implementation of monitoring events defined in MonitA.

7.5 Composing the MonitA Code with Workflow Applications

In addition to transforming M&A models (specifications) into code, these specifications must be composed to produce and executable application. This section details the *composition controller* component illustrated in Figure 7.3. The composition controller defines how to compose automatically MonitA specifications with the workflow application.

The composition between the generated MonitA code and the workflow code is performed automatically at the implementation level (*i.e.,* workflow definition and underlying implementation). The code required to compose the

workflow application and M&A concerns is generated according to the weaver engine provided by the aspect language (*e.g.,* Padus, AspectJ).

The resulting code artifacts correspond to an executable workflow implementation instrumented with M&A concerns.

7.5.1 Selecting the Level of Abstraction

The composition of M&A concerns requires the integration of multiple models (*i.e.,* MonitA model, process model, workflow data model) specified in different languages. The composition between these models can be performed at the conceptual level or at the implementation level by defining the link between concepts in the different models.

The composition of heterogeneous models at a conceptual level requires the definition of the composition semantics between concepts of two metamodels. At the conceptual level, the complexity to define the composition semantics increases for each metamodel added to the specification since the new metamodel could not have a well defined semantics. Moreover, each time a new concept is added to a metamodel or a new metamodel is incorporated for the specification of M&A concerns, the composition mechanism has to be adapted affecting the evolution of the resulting applications. Nevertheless, the main motivation to compose MonitA specifications and workflow applications at a high-level of abstraction is to validate their consistency before execution. A possible validation scenario is to identify potential M&A concerns interferences and interactions between MonitA specifications declared by different application developers. This facilitates the detection of conflicts independently of the technology, therefore the validation is performed only once ensuring that the MonitA specifications are going to execute correctly in every execution technology. An evaluation of potential conflict between MonitA specifications is presented in section 11.2.3.

We delay the composition between the workflow application and the MonitA specifications at the implementation (application) level to reuse existing composition mechanisms. At the implementation level, the semantics of the generated M&A code is well defined and the composition semantics is already defined around uniform composition concepts. Thus, each time a new concept is added to a metamodel or a new metamodel is incorporated for the specification of M&A concerns, the composition mechanism is not affected. We performed a composition at the implementation level by considering that the code of the workflow applications already exists. The advantages at this low-level of abstraction are that a) the semantic gap is smaller since both domains converge to one abstraction, b) the models are richer in implementation details, which allows fine grained composition, and c) the existing assets (metamodels, models, and model transformations) can be reused.

The MonitA specifications cannot be fully mapped to elements of the

BPMN process model. This is mainly because the MonitA specifications not only add new activities to workflow specification but they also require to add elements to the underlying implementation of the activities. Consequently, MonitA specifications are not longer executable. Specifically the measurement and control actions such as the declaration, creation, navigation, and persistence of measurement information cannot be expressed in terms of process models. A first effort for composing MonitA specifications at the conceptual level and the required research work is presented in section 11.2.1.

Analyzing the advantages and disadvantages of these composition approaches, the best decision to compose M&A concerns with a workflow application seems to be the combination of both levels of abstraction.

7.6 Summary

The architecture and strategy outlined in section 7.2 is used by MonitA infrastructure developers to automate implementation of MonitA specifications into a new target workflow platform. The target workflow platform requires a mechanism to support separation of concerns in order to generate accurate monitoring and analysis implementations according to the specifications. An aspect-oriented workflow language (*e.g.,* Padus) can be used to modularize M&A concerns when it is supported by the workflow implementation language (*e.g.,* BPEL). Otherwise, an AOP language (*e.g.,* AspectJ) supported by the underlying implementation language (*e.g.,* Java) can be used to modularize the M&A concerns.

A MonitA infrastructure developer of a new MonitA generative infrastructure must define the mappings from the MonitA model to the language abstractions in the target workflow platform.

Model transformations in a Model-Driven Engineering (MDE) context are used to translate the MonitA specifications into executable workflow language code. The generated code is modularized and composed with a workflow application by using aspect-oriented programming (AOP) technology. The generative strategy for MonitA specifications offer some maintainability possibilities. For example, if the workflow specification changes, the M&A concerns implementation can be re-generated and composed with the new workflow implementation without loosing the MonitA specification and reducing the time to implement M&A concerns.

Our generation strategy offers support for transforming different modeling languages, implementation languages, and workflow engines. We used this strategy to automate the implementation of MonitA specifications into two different executable workflow platforms: a) the JPDL workflow language [JPD] with the jBPM engine, and b) the BPEL workflow language with the Apache ODE engine. AspectJ and Padus were used as languages for separation of con-

cerns in these platforms respectively. The next chapter presents the realization of these two MonitA generative infrastructures.

Chapter 8

MonitA: The Implementation and Execution Infrastructure

This chapter presents the MonitA infrastructure that has been developed for deploying MonitA specifications in workflow applications.

Section 8.1 presents the technologies used to implement the MonitA platform.

Sections 8.2 and 8.3 present the validation done with two different MonitA infrastructure developers to implement two different MonitA generative infrastructures (*i.e.,* MonitA-JPDL and MonitA-BPEL). These implementations are an instantiation of the generative strategy described in the previous chapter. For each infrastructure we detail the same components presented in the architecture for the MonitA generative infrastructure (*cf.* Figure 7.3 in section 7.2). The model transformation approach used to transform the MonitA models workflow code is described in detail in appendix D.

Section 8.4 presents the infrastructure for enacting monitoring and analysis (M&A) concerns at runtime. The enactment involves elements to store and manage historic measurement information, and to visualize this information in a dashboard.

8.1 Selected Technology

The following technologies have been used to implement the MonitA platform:

- *Language Implementation.* We use Xtext [Ope] version 4.3.1 as a framework for the development of our DSLs. Xtext provides an EBNF grammar language and a generator that creates automatically the implementation of the parsers. Xtext also generates an AST-meta model (implemented in EMF) as well as an Eclipse text editor. Xtext is an intermediate language, whereas antlr is the language that provides the grammar syntax. The language is compiled into Java.

- *Model transformations.* We use the Xpand generator framework (version 4.3.1) to implement different assets in our model-driven development process. These assets correspond to checks, extensions, code generation, and model transformations. The Xpand generator framework provides textual languages (*e.g.,* Check , Xtend , and Xpand) to specify these assets.

- *Workflow languages.* We use BPEL 1.1 and JPDL 3.2.3 for the implementation of workflow applications.

- *Workflow engines.* We use ApacheODE 1.2 and jBPM 3.2.3 for the enactment of workflow instances.

- *Aspect languages.* We use the Padus weaver (build 2009.09.15) and AspectJ 1.6.5 for the implementation of monitoring and analysis aspects.

- *Programming languages.* We use Java 1.5 for the core implementation of the monitoring and analysis framework.

- *Graphical user interface.* We use Google Web Toolkit (GWT) 1.6.4 to create the graphical interface to provide feedback to workflow analysts about the online workflow monitoring and analysis. GWT allows developers to create JavaScript front-end applications in the Java programming language.

8.2 MonitA-JPDL Generative Infrastructure

This section presents the instantiation of our generative strategy (chapter 7) to implement and compose MonitA specifications with JPDL workflow applications (*cf.* goal G1 and assessment goal AG1 in section 1.3).

The following sections detail the three components we defined in the architecture for creating a MonitA generative infrastructure: the workflow code generator, the MonitA code generator, and the composition controller.

8.2.1 JPDL Workflow Code Generator

We control the workflow generation process in order to store the traceability information between the source (*i.e.,* process model) and target (*i.e.,* workflow implementation) elements (*cf.* generative strategy described in section 7.3). In order to do so, we need a) to create a transformation to generate JPDL workflow applications from BPMN process models, b) to create a traceability model storing the corresponding mappings, and c) to access workflow data.

The following three sections detail the instantiation of the modules contained in the *workflow code generator* component (Figure 7.3 in section 7.2).

The modules within the workflow code generator correspond to: BPMN translator, traceability manager, and process data manager (*cf.* implementation challenges IC1 and IC4 in section 6.1).

BPMN Translator to JPDL

We created a model-to-model transformation (*T1 in Figure 7.4*) for mapping a subset of BPMN elements into JPDL elements. The BPMN models are created conform to the the BPMN metamodel provided by the BPMN modeler project [The08]. In the particular case of JPDL we generated an ecore JPDL implementation metamodel from the XSD specification provided in the JPDL project[1]. A description of the elements involved in these metamodels are presented in sections 2.1.2 and 2.1.3.

This transformation generates a traceability model to capture the mapping between the BPMN and JPDL models. We do not provide a complete transformation between BPMN to JPDL since there are tools (*e.g.,* JBoss BPMN Convert module[2]) focussed on that. However, it is important to incorporate the traceability model into the existing workflow generation tools in order to trace the naming convention used by each one of them. A MonitA infrastructure developer created this transformation to extract a traceability model by controling the workflow generation process.

This model-to-model transformation translates automatically the BPMN specification into flow elements in the workflow definition file of JPDL and into the structure of two handlers (Java classes). An action handler is created to implement the workflow application actions triggered by workflow events. A decision handler is created to manage the control flow of the workflow application in the decision nodes elements depending on the information used by the workflow. The actual implementation code of the activities is implemented manually in Java by the workflow developers.

All the mappings between the BPMN model and the JPDL implementation model are one to one mappings since the considered BPMN elements have a corresponding JPDL representation. The transformation between BPMN and JPDL covers only the basic elements of BPMN (*i.e.,* activity, task, split, start node, end node, decision nodes, transitions) that facilitates the process representation. Complex structures such as subprocesses and other element types were not taken into account in this model-to-model transformation. Nevertheless, these mappings are not difficult to add.

[1] http://docs.jboss.org/jbpm/xsd/jpdl-3.1.xsd
[2] http://docs.jboss.org/tools/3.1.0.M2/en/jboss_bpmn_convert_ref_guide/html_single/

Traceability Model Generator

We encoded a set of traceability rules in the aforementioned model-to-model transformation in order to generate a traceability model. The traceability model contains a reference to source and target elements in the BPMN to JPDL transformation. These references facilitate the localization of elements in both models to extract their name and type.

Figure 8.1 illustrates the traceability model generated in the transformation from the trouble ticket BPMN model (see section 1.2) to a JPDL model. The traceability model contains the type and name of the source and target elements of the model transformation. For example, the `Activity` named *Submit Form* in the BPMN model was translated into a `Task Node` with the same name in the JPDL model.

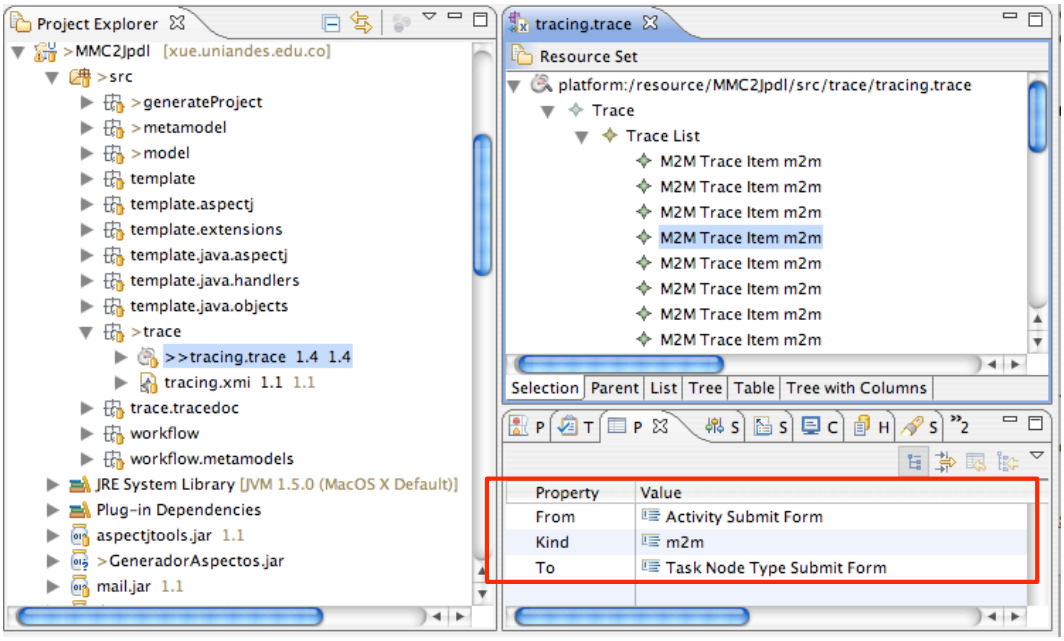


Figure 8.1: Traceability Model Generated in a Workflow Implementation.

This information can be used by another model transformation to automatically extract the correspondence between elements in the BPMN specification and elements in the JPDL implementation. M&A concerns are specified in terms of BPMN elements but they have to be translated into JPDL elements to be executed. For example, if a monitoring event is specified to monitor all activities in the BPMN process model (*on start [root.!Activity]*), a node-event workflow event must be added for all flow entities of type *TaskNode* in JPDL.

Process Data Manager in JPDL

In JPDL all the workflow information is accessed through the ExecutionContext class provided by the API of the jBPM engine. We developed an interface (Java class) to encapsulate the available methods that can be used by the MonitA specification to access the workflow data (*Process Data Interface in Figure 7.4*). The methods provided by the interface invoke a subset of the ones provided by the *ExecutionContext*. This is done to get read-only access to the workflow variables since we do not want to alter the state of the workflow application. The methods provided by the interface facilitates to retrieve a) a workflow variable, b) the identifier of a workflow instance, c) the identifier of the workflow application, and d) the name of the workflow application.

The workflow variables specified in the data association model are accessed through this interface from the workflow engine. Each time a MonitA concern references a workflow variable, defined in the data association model, the MonitA platform uses the interface to extract its data type, to define the required casting, and to return the value of the workflow variable. The value of a workflow variable is retrieved for the workflow instance where the MonitA specifications is being executed. In the current implementation, we assume that the data association model reflects the actual workflow variables, however, if both specification and implementation are not synchronized then there can be inconsistency problems (see Section 11.2.3).

8.2.2 MonitA Code Generator into JPDL

According to our generative strategy described in section 7.4, we generate MonitA code taking into account the modularization of MonitA specifications and the management of measurement data (*cf.* implementation challenges IC2 and IC3 in section 6.1). This section validates our generative strategy by implementing a set of model transformations to generate executable monitoring and analysis JPDL code from the MonitA specifications.

We created a model-to-model transformation that takes as input the MonitA model, the JPDL model and the traceability model to generate a new JPDL model instrumented with the elements required for monitoring and analysis.

The following four sections detail the instantiation of the modules contained in the *MonitA code generator* component defined in our architecture for creating a MonitA generative infrastructure (Figure 7.3 in section 7.2). The modules within the MonitA code generator component correspond to: workflow code generator, AOP code generator, measurement data translator, and measurement and control actions manager.

JPDL Code Generator

This module defines how to translate monitoring concerns into control flow entities in the JPDL workflow language. We created a model transformation (*Tb in Figure 7.4*) to instrument the original JPDL model with JPDL events required to implement the monitoring events specified in the MonitA-DSL. The JPDL model is instrumented only with the control flow related events types (*i.e.,* node-enter, node-leave) since data related events are not supported by the workflow language. The MonitA monitoring events that require to intercept data related event types are implemented as AspectJ aspects, which intercept the Java classes that represent the workflow variables when they are accessed by a workflow activity. This is detailed later in this section.

The main input in this transformation are the monitoring events specified in a MonitA model, which represent the interception of a workflow event to execute complementary measurement and analysis code. Monitoring events in MonitA describe the moment to generate the event (workflow event type), the element or set of BPMN elements (monitoring subject) that must match the event type, and the additional actions (function invocation) that must be executed. A monitoring event in MonitA generates an event element in JPDL, which is added to each JPDL node that corresponds to the monitoring subject specified in the monitoring event. The generated JPDL event contains information about the moment in which the workflow subject is intercepted (*e.g.,* node-leave) and the action (Java class) that must be executed at this moment. The link between the workflow subject defined in terms of BPMN activities and the corresponding JPDL node automatically established through the traceability model.

Table 8.1 summarizes the mappings between conceptual MonitA workflow event types and implementation JPDL event types.

MonitA model		JPDL model	
Event type	*Workflow subject type*	*Event type*	*Node type*
start	Task	node-enter	TaskNodeType
start	EventEndEmpty	node-enter	EndStateType
start	EventStartEmpty	node-enter	StartStateType
start	GatewayParallel	node-enter	ForkType
start	GatewayDataBaseExclusive	node-enter	DecisionType
finish	Task	node-leave	TaskNodeType
finish	EventEndEmpty	node-leave	EndStateType
finish	EventStartEmpty	node-leave	StartStateType
finish	GatewayParallel	node-leave	ForkType
finish	GatewayDataBaseExclusive	node-leave	DecisionType

Table 8.1: Mapping conceptual events to JPDL workflow events

We created a model-to-text transformation (*T2 in Figure 7.4*) to generate the workflow specification (xml file) and the handlers (Java code) required to manage the workflow application. The input of this transformation is the JDL implementation model that is instrumented with monitoring events as explained previously. This transformation generates the workflow specification in the JPDL workflow language (processdefinition.xml) and three action handlers. An action handler is created to implement the workflow application actions triggered by workflow events. A decision handler is created to manage the control flow of the workflow application in the decision nodes elements depending on the information used by the workflow. The actual implementation code of the activities is implemented manually in Java by the workflow developers.

Our MonitA generative strategy (see section 7.4) defines the creation of a set of to transformations required to generate MonitA executable code. Consequently, we created a model-to-text transformation (*Tc in Figure 7.4*) to generate the delegated application code required to manage the monitoring events contained in the workflow specification. The input of this transformation are the MonitA model (MonitA-DSL specifications) and the traceability model. The output of this transformation are the handlers (Java classes) required to implement the M&A concerns that must be executed when a monitoring event is triggered in the workflow application.

A handler is generated to implement the actions associated to the JPDL events regarding monitoring activities. The methods associated with the generated handler can be intercepted by the aspect language to execute the measurement and control actions defined in each analysis function. Figure 8.2 illustrates a fragment of the handler generated from the MonitA model.

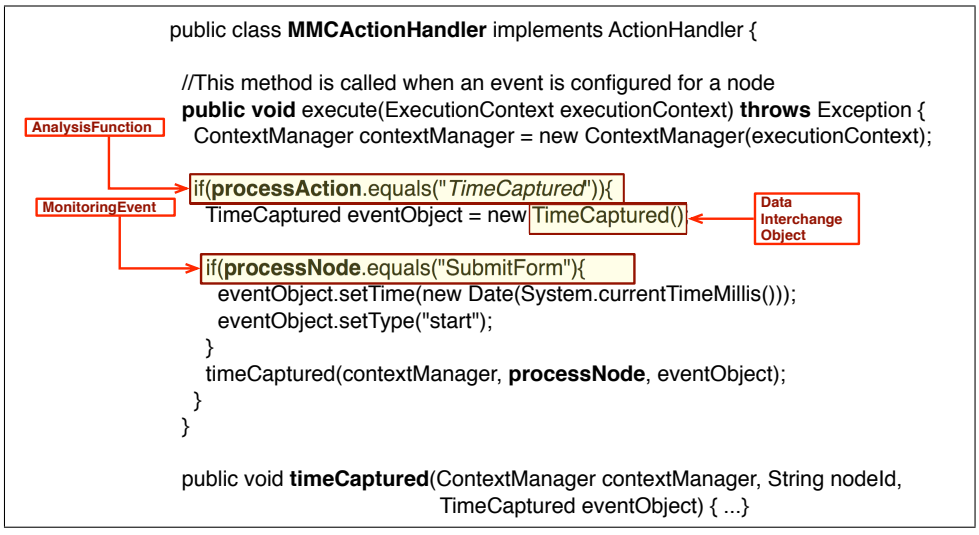


Figure 8.2: Generated handler to capture monitoring information.

Each monitoring event in MonitA generates a conditional in the action handler to process the action (processAction in Figure 8.2) when the generated event matches the monitoring event. Each conditional contains nested conditionals to validate the activity that generated the event (processNode in Figure 8.2) and to capture the information required to create the objects (timeCaptured(...) in Figure 8.2) that allow the interchange of information. The object type is generated with the name of the analysis function specified in MonitA. This handler processes all events generated in the workflow application. Consequently, this is the central point that AspectJ aspects use to intercept and to associate the M&A concerns.

Another model-to-text transformation (*Tc in Figure 7.4*) was created to generate a set of Java classes that represent the information that has to be interchanged between the handlers and aspects. This model-to-text transformation generates a Java class for each analysis function defined in a MonitA specification. For each parameter of an analysis function, a class attribute is generated with its corresponding name, data type, and accesor methods.

Measurement Data Translator

This module defines how to map the MonitA measurement variables into workflow data entities.

A measurement variable is mapped to a variable in Java, which is declared by using the name and type of the measurement variable. Measurement variables are declared in XML, but their JPDL implementation are not; therefore, we need a mapping between XML and Java. The measurement data types are declared according to the XML schemas data typing, which include support for Java primitive datatypes. A XML schema specification admits a processor in which particular schemas are compiled into executable Java code. This processor *can be said to be minimally conforming but not necessarily in conformance to the XML Representation of Schemas* [BM04].

The MonitA-JPDL infrastructure provide support to map primitive data types (*e.g.,* string) into built-in Java types and to generate Java classes for measurement complex types (*e.g.,* ProblemsByArea).

Table 8.2 summarizes the automatic mapping between XML schema and Java primitive data types.

The MonitA-JPDL infrastructure maps the primitive measurement data types to either primitive or object types in Java. The mapping between primitive measurement data types to JPDL data types can be adapted if required.

The complex measurement data types, which are defined in an external XML Schema (XSD) file (measurement data types model in section 4.1.1), are mapped into a Java class. The Java class is generated with the set of Java variables (names and data types) corresponding to the elements of the measurement variable data type, and with the set of getter and setter methods

XML schema data type	*Java data type*
xs:string	Java.lang.String
xs:int	int or Java.lang.Integer
xs:long	long or Java.lang.Long
xs:short	short or Java.lang.Short
xs:float	float or Java.lang.Float
xs:double	double or Java.lang.Double
xs:Boolean	boolean or Java.lang.Boolean
xs:dateTime	Java.util.Calendar

Table 8.2: Primitive Data Types Mapping between XML Schema and Java

to access and modify these elements. For example, the measurement data type *ProblemsByArea*, defined for the trouble ticket scenario (see Section 1.2), contains a set of elements such as problems and area with their associated primitive data types (*i.e.,* int, string). For this complex measurement data type, the MonitA-JPDL infrastructure generates an interface and a Java class to access and modify the elements (problems, name) of a ProblemsByArea measurement variable.

Measurement Data and Actions Manager

This module defines how to access and manage the measurement information, and how to generate notification actions.

We generate a set of Java artifacts named *Managers* to access and manage the measurement information. These managers communicate with the persistence systems through a set of interfaces. The workflow developer has to provide these interfaces and to connect them with the managers. The following describes the role of each one of these managers:

- The *DataManager* artifact is a singleton class that manage the measurement information such as measures, indicators and workflow records. The persistence management of measurement information within the workflow engine is delegated to Hibernate [KEvS+05].

- The *ContextManager* is a singleton class that encapsulates the JPDL execution context to restrict the services to manage workflow variables to only read-only access.

- The *ControlManager* is a singleton class that offers the methods to execute the notification actions such as sending an email, creating an event log trace, and sending events to external systems (*i.e.,* dashboard).

The measurement data manager, the measurement actions manager, and the measurement data translator components can be reused by a MonitA in-

frastructure developer that requires to create a new MonitA generative infrastructure (*e.g.,* MonitA-BPEL).

MonitA Translator to AspectJ Aspects

This module defines how to modularize the implementation of MonitA specifications into the workflow application implementation.

We created a model-to-text transformation (*Tc in Figure 7.4*) to generate the aspect code required to implement the MonitA specifications. The input of this transformation is the MonitA model, which contains references to the measurement data model, and to the data association model.

Each monitoring event that triggers an *analysis function* in MonitA generates an AspectJ aspect. The following scenarios present the transformation of different MonitA elements to AspectJ elements:

- *Flow-based monitoring events.* These monitoring events correspond to the ones defined in terms of a flow event (*i.e.,* start, finish) related to a flow entity (*e.g.,* activity, gateway). For example, the *on start [root.SubmitForm] triggers functionName()* monitoring event triggers an analysis function when the activity named SubmitForm starts.

  The main artifact to be intercepted is the generated monitoring handler (Java class) that contains the methods delegated by the workflow specification (JPDL file) to implement the analysis functions. The name of the analysis function defines the name of the method invoked by the generated AspectJ *pointcut*. The workflow event type and the monitoring subject in a monitoring event defines the AspectJ *joinpoint* to instrument the workflow application before or after the execution of the method.

- *Data-based monitoring events.* These monitoring events correspond to the ones defined in terms of a data event (*i.e.,* change, create, read, delete) related to a data entity (*e.g.,* workflow variable, variable attribute). For example, the *on change [root:vProblem] triggers functionName()* monitoring event triggers an analysis function when the workflow variable vProblem is modified in the workflow application. The main artifact to be intercepted is the class that contains the representation of the workflow variable referenced in the monitoring subject.

  If the monitoring event is defined in terms of an attribute of a data entity (*e.g.,* on read [root.SubmitForm:vProblem.area]), the attribute name defines the name of the method invoked by the generated AspectJ pointcut. The data type of the monitoring subject (data entity) and the attribute name in a monitoring event defines the joinpoint to instrument the workflow application after the execution of the method associated to the pointcut.

When the monitoring event is defined in terms of a workflow variable (*e.g.,* on change [root.SubmitForm:vProblem]), all the methods associated with the workflow event type (setter methods) are going to be invoked by the generated pointcut. The data type of the monitoring subject (data entity) in a monitoring event defines the joinpoint to instrument the workflow application after the execution of each method associated to the pointcut.

This mechanism to intercept data related events can be used when the workflow data is managed by the execution context or by an external system. This is because the interception is done on the data object directly.

The information about the activity that perform an operation on a data entity, the type of operation, and the activity user are not captured directly in the aspect. This is because of the aspect is observing a data object directly and this object does not know about this information. Thus, the advice uses the ContextManager interface to retrieve this information if required by the MonitA specification.

- *Analysis Functions.* Each analysis function generates an AspectJ advice that contains the code to implement the measurement and control actions and to evaluate the monitoring and measurement information. The code of the advice corresponds to Java code.

  First, a MonitA measurement action is used to manage the monitoring and measurement data. The measurement variables are computed through a set of operations between invocations. The elements in the invocation are analyzed to determine if it is necessary to query or compute other data previously. The operations (addition, substraction, multiplication, division) between these invocations have a direct mapping to Java. Each invocation has a particular transformation to Java code that uses the managers generated to access and manage the monitoring and workflow information. Once the value of a measurement is computed through the assignment function, this value is stored in the measurement data store system.

  Second, a MonitA control action is used to notify specific events after evaluating the value of a measurement. These notifications are used for sending an email, creating an event log trace, and sending events to a dashboard. These elements correspond to the invocation of a method of the generated ControlManager interface.

  Finally, an evaluation rule is used to evaluate the monitoring and measurement information in order to take some notification actions. An evaluation rule is represented with the if-else statement in Java. The set of invocations involved in a condition are evaluated through logic and boolean operators, which have a direct representation in Java. The set of actions correspond to the measurement and control actions describe previously.

8.2.3 Composing MonitA Code with JPDL Applications

We compose automatically MonitA specifications with the workflow application (*cf.* generative strategy described in section 7.5). This section validates our generative strategy by generating the code required to compose M&A concerns with JPDL workflow applications (*cf.* implementation challenge IC2 in section 6.1). This section also describes the instantiation of the weaver module contained in the *composition controller* component defined in our architecture for creating a MonitA generative infrastructure (Figure 7.3 in section 7.2).

We use the generated traceability model to identify the flow elements in the JPDL workflow implementation that must be instrumented according to the monitoring events defined in MonitA. These elements are instrumented with additional workflow events to capture state changes during its execution. The AspectJ weaver integrates the generated aspects and classes (*i.e.,* handlers and data representation) associated to these events to generate a woven class. The weaver integrates the aspects into the locations specified in the joinpoints.

In JPDL applications, the analysis functions are added inside an activity when using workflow events. Analysis functions are not added before or after the monitored activity. It would require to modify the transitions of the original workflow application. Analysis functions are also added inside an activity when using aspects in terms of the data.

8.3 MonitA-BPEL Generative Infrastructure

This section presents the instantiation of our generative strategy (chapter 7) to implement and compose MonitA specifications with BPEL workflow applications (*cf.* goal G1 and assessment goal AG1 in section 1.3).

We created a set of model transformations to automatically generate the implementation of MonitA specifications into a BPEL platform. The input of these transformations are the MonitA specifications, whereas the output is workflow code in BPEL and aspects code in Padus. The MonitA specification is translated into multiple artifacts in order to get an executable implementation.

Table 8.3 presents an overview of the transformations from MonitA to artifacts in a BPEL platform.

The following sections detail the three components we defined in the architecture for creating a MonitA generative infrastructure: the workflow code generator, the MonitA code generator, and the composition controller.

8.3.1 BPEL Workflow Code Generator

According to our generative strategy described in section 7.3, we control the workflow generation process in order to store the traceability information in

MonitA element	*Output artifact*	*Description*
MonitA model	Padus aspect	- It contains several advices representing the crosscutting M&A concerns - It contains Bpel code referencing the workflow application
	Weaver Java class	It is the weaving engine that composes the workflow application and the aspects
Monitoring events	Deployment file	It specifies how the Padus aspects are instantiated and composed to the workflow application
Measurement and control actions	WSDL file	It describes the web services created to manage the measurement data and to send notifications
Measurement data types	XSD elements	It describes the measure data types specified in the language
Measurement variables	BPEL variables	It defines a global variable to be added during the weaving process
Workflow variables	BPEL code	It references the message definitions

Table 8.3: MonitA specification into BPEL executable elements.

the mapping between BPMN process model and BPEL workflow implementation elements. In the creation of the MonitA platform to BPEL we did not implement the generation of JPDL workflow applications from BPMN process models. We took into account the mapping of BPMN to BPEL that has been defined in a number of references [GTP07] [Whi05] and that has been implemented in a number of tools such as the open-source tool known as BPMN2BPEL [BPM] and Intalio [Int].

The following three subsections detail the instantiation of the modules contained in the *workflow code generator* component defined in our architecture for creating a MonitA generative infrastructure (Figure 7.3 in section 7.2). The modules within the workflow code generator correspond to: BPMN translator, traceability manager, and process data manager (*cf.* implementation challenges IC1 and IC4 in section 6.1).

BPMN Translator to BPEL

We validated our workflow monitoring and analysis approach into a workflow application implemented in BPEL (see section 9.2). This workflow application was specified in BPMN through the Intalio business process platform [Int]. This platform generates partially the workflow application in BPEL.

Traceability Model Generator

The MonitA infrastructure developer has to control the workflow generation process in order to store the links between the source (*i.e.,* process model) and target (*i.e.,* workflow implementation) elements. Nevertheless, in this implementation we did not use a traceability model since we do have control over the existing workflow generation process. However, the automatic creation of a traceability model can be performed if the MonitA infrastructure developer gets access to the BPMN to BPEL generation system to customize it (see Section 8.2.1).

We identified and adopted the naming conventions used by Intalio to encode in our transformations the way to determine the link between BPMN elements and BPEL elements.

Process Data Manager in BPEL

For the BPEL platform we assumed that the workflow variables are managed externally to the workflow engine. We developed a web service to encapsulate the available methods that can be used by the MonitA specification to manage the workflow data (*Process Data Interface in Figure 7.4*). The web service manage the collection of workflow applications, the process scope variables defined for each workflow application, and the instance scope variables for each workflow instance.

Each time a MonitA concern references a workflow variable, specified in the data association model, the MonitA platform uses the web service to extract its data type, to define the required casting to access the value of the variable, and to return its value. The value of a workflow variable can be retrieved from a particular workflow instance and for the all workflow application.

8.3.2 MonitA Code Generator into BPEL

According to our generative strategy described in section 7.4, we generate MonitA code taking into account the modularization of M&A concerns and the management of measurement data (*cf.* implementation challenges IC2 and IC3 in section 6.1). This section validates our generative strategy by implementing a set of model transformations to generate executable monitoring and analysis BPEL code from the MonitA specifications.

We created a model-to-text transformation that takes as input the MonitA model to generate new BPEL elements required to instrument the existing BPEL code with the elements required for monitoring and analysis.

The following four subsections detail the instantiation of the modules contained in the *MonitA code generator* component defined in our architecture for creating a MonitA generative infrastructure (Figure 7.3 in section 7.2). The

modules within the MonitA code generator component correspond to: workflow code generator, AOP code generator, measurement data translator, and measurement and control actions manager.

BPEL Code Generator

This module defines how to translate monitoring concerns into control flow entities in the BPEL workflow language.

The BPEL workflow application is fully instrumented through aspects. This is because of the aspects language that allows the specification of crosscutting concerns directly on BPEL elements.

We created a model-to-text transformation (*Tc in Figure 7.4*) to generate the BPEL code required to implement the MonitA specifications. The following are the considerations taken to determine the outputs of this transformation:

- The properties defined for an analysis function are mapped into a BPEL variable in the same way that measurement variables. The value for these properties are assigned at the moment a monitoring event is generated. An *assign* element in BPEL is created for each property that parametrizes an analysis function. The name of the property is mapped into the *"To"* element of the assign. The expressions for collection of data in a monitoring event are mapped to the *"From"* element of the assign.

- A measurement action consisting of an assignment to a measurement variable has multiple considerations. Only if the assignment is done through a measurement variable with multiinstance persistence modifier, its value is retrieved from the measurement data store system. The value of this variable is queried through an assign activity by assigning the retrieved values of the variable (*e.g.,* process name, variable name). At the same time, an invoke activity is executed to invoke the services provided by the data manager to manage this information. Then, each invocation in the assignment expression is defined with an assign activity to retrieve their value and to compute the new value for the measurement. The new value of the measurement is stored in the measurement data store system.

- The control actions are mapped into invocations to services provided by the ControlManager interface. The parameters of the invocation are extracted from the control action (*e.g.,* send an email).

- The evaluation rules defined to evaluate workflow and measurement information are mapped to bpel conditionals (*i.e.,* bpel:if). The set of conditions are encapsulated into the *bpel:condition* element. All the actions defined in the evaluation rule are encapsulated into a *bpel:sequence* element. An else

expression in the evaluation rule is mapped into a *bpel:else* element and the set of actions of this conditional are mapped into bpel activities encapsulated in a sequence element.

An invocation in MonitA can correspond to one of the following:

- An invocation associated with a *data entity* is used to access the workflow variables through a flow entity. This invocation is represented in Java as an invocation of an object attribute previously declared. This object is accessed through a method in the class who manages the workflow information.

- An invocation associated with a *literal* element is used to specify basic elements such as a string or a number. These literals are represented in BPEL by Xpath expressions or directly by a literal as an XML expression (*e.g.,* an integer object).

- An invocation associated with a *engine invocation* element is used to access the information in the workfow engine. The functions provided by the workflow engine are defined by a namespace and indicated by a prefix (*e.g.,* $ode:). An *engine.instanceId* expression in MonitA can be translated into a *$ode:pid* expression in the apache ode engine, which retrieves the identifier of the workflow instance.

- An invocation associated with a *date time invocation* element is used to access the current date time and to retrieve properties of a variable with dateTime data type. The functions related to a dateTime data type (*e.g.,* now()) in MonitA are mapped to the BPEL XPath functions for assigning date or time (*e.g.,* getCurrentDateTime).

Measurement Data Translator

This module defines how to map the MonitA measurement variables into workflow data entities. The measurement variables are transformed into BPEL variables, which are declared by using the name and type of the measurement variable. The generated BPEL variable are contained in the *using* structure of a Padus aspect, which are global to the workflow application. Measurement variables are declared according to XML schemas (XSD) data types, therefore, there is a one-to-one mapping of data types and names between the measurement and BPEL variables.

When a measurement variable uses a primitive data type (*e.g.,* int), the BPEL variable is generated with this primitive data type. The complex data types associated to measurement variables are represented in the external XML schema file (measurement data types model in section 4.1.1) that is generated to define complex measurement data structures. A complex measurement data

type is created with the set of property names and data types associated to the measurement variable data type.

The MonitA-BPEL infrastructure generates an interface and Java class to access and modify the elements of a measurement variable. For example, the elements (problems, area) of the *ProblemsByArea* variable, defined for the trouble ticket scenario (see Section 1.2), can be accessed and modified through interface generated by the infrastructure.

Measurement Data and Actions Manager

This module defines how to access and manage the measurement information, and how to generate notification actions.

The measurement data manager and the measurement actions manager described in the MonitA-JPDL generative infrastructure are reused by the MonitA infrastructure developer that creates the MonitA-BPEL infrastructure. The following describes the role of each one of these managers:

- The *DataManager* artifact is a singleton class that manage the the measurement information such as measures, indicators and workflow records. The persistence management of measurement information is external to the workflow engine and is done through a web service.

- The *ContextManager* is a singleton class that encapsulates the BPEL invocations to restrict the services to manage workflow variables to only read-only access. The workflow developer has to provide the interface to communicate with the workflow data persistence system and to connect this interface with the manager.

- The *ControlManager* is a singleton class that offers the methods to execute the notification actions such as sending an email, creating an event log trace, and sending events to external systems (*i.e.,* dashboard).

MonitA Translator to Padus Aspects

This module defines how to modularize the implementation of MonitA specifications into the workflow application implementation.

We present the automatic transformations performed from MonitA specifications to executable aspects. We represent these transformations by mean of Padus, an aspect-oriented extension to BPEL, which aims to overcome its lack of support for modularization of crosscutting concerns.

The Padus language facilitates the separation of M&A concerns from the workflow implementation. The monitoring events in MonitA describe how an analysis function has to interact with a workflow element, acting as a pointcut and joinpoint statement. The measurement and control concerns specified in

MonitA define what actions the analysis function (aspect) does according to the event, thus acting as an advice.

Table 8.4 illustrates the most relevant transformations from MonitA elements to Padus elements.

MonitA	**Output : Padus Implementation**
MonitA model	Aspect defining the MonitA specification <pad:aspect name="" xmlns:xsi="" xmlns:bpel="" xmlns:pad=""> ... </pad:aspect>
Measurement variable	BPEL variables within the using element of an aspect <pad:using> <!– namespaces, partnerLinks, variables–> </pad:using>
Analysis function	Advice <pad:advice name=""> <!–Bpel code–> </pad:advice>
Workflow subject	Pointcut <pad:pointcut name="" pointcut=""/>
Monitoring event - *start* workflow event type - *finish* workflow event type - data (CRUD) event type	Joinpoint - Before advice <pad:before joinpoint="" pointcut=""> <bpel:advice name=""/> </pad:before> - After advice <pad:after joinpoint="" pointcut=""> <bpel:advice name=""/> </pad:after> - In advice or after advice <pad:in joinpoint="" pointcut=""> <bpel:advice name=""/> </pad:in>

Table 8.4: MonitA specification into a Padus aspect implementation.

We created a model-to-text transformation (*Tc in Figure 7.4*) to generate the aspects code required to implement completely the monitoring and analysis concerns defined in MonitA. The input of this transformation is the MonitA model, whereas the output is a set of Padus aspects that contain using, pointcut, joinpoint and advices elements.

Using element. The measurement variables in MonitA are transformed in BPEL variables, which are encapsulated in the *using* element of a Padus aspect that is represented in a XML file. A MonitA specification also generates the Padus code with the namespaces and partnerlinks required to invoke the services provided by the generated managers (measurement data manager and actions manager).

Pointcut element. A Padus pointcut is generated for each monitoring subject specified in a monitoring event in MonitA. The pointcut contains the property name and the property pointcut defining the joinpoint. The name property of the pointcut element corresponds to the name of the monitoring subject. The pointcut property of the pointcut element, which references the

bpel activity and their properties, is generated according to the type of the bpel activity related to the monitoring subject.

If the activity type is an invoke then the pointcut corresponds to an *invoking* construct (*e.g.,* pointcut="invoking(Jp,[operation('operationName')])"), which references an operation of the activity. In the activity type is an assign then the pointcut corresponds to an *assigning* construct (*e.g.,* pointcut="assigning(Jp,[name('assignName')])"). If the activity type is task then the pointcut corresponds to a *doingNothing* Padus construct (*e.g.,* pointcut=" doingNothing(Jp,[name('TaskName')])"), which references all the empty activities. If the activity type is loop then the pointcut corresponds to a *scoping* Padus construct, which references a cyclic activity (*e.g.,* poincut="scoping(Jp, [name('CyclicActivityName')])").

Joinpoint element. Each monitoring subject defined in a monitoring event generates a joinpoint element in Padus. If the workflow event type defined in the monitoring event is *start*, the relative position in the joinpoint is *before*. If the workflow event type defined in the monitoring event is *finish*, the relative position in the joinpoint is *after*. If the workflow event type defined in the monitoring event is a data event type (*e.g.,* create, change), the relative position in the joinpoint can be *in* or *after* depending the bpel element.

The pointcut property in the joinpoint element corresponds to the name of one of the pointcut elements previously generated from the workflow subjects. The properties associated with the analysis function specified in a monitoring event correspond to bpel assignations (assign element).

Advice element. Each analysis function in MonitA generates an advice, which is encapsulated in a Padus aspect along with the pointcut and joinpoint elements. The name of the analysis function defines the name of the advice. A Padus advice encapsulates the MonitA BPEL code to be included in the workflow application. The measurement and control actions specified in an analysis function are mapped to a set of BPEL activities as described before.

8.3.3 Composing MonitA Code with BPEL Applications

We compose automatically MonitA specifications with the workflow application (*cf.* generative strategy described in section 7.5). This section validates our generative strategy by generating the code required to compose M&A concerns with BPEL workflow applications (*cf.* implementation challenge IC2 in section 6.1). This section describes the instantiation of the weaver module contained in the *composition controller* component defined in our architecture for creating a MonitA generative infrastructure (Figure 7.3 in section 7.2).

After the MonitA specification is generated into BPEL code, the Padus weaver engine integrates this generated code with the existing workflow implementation. This composition is triggered through the execution of the aspect deployment file that was generated automatically according to the generated aspects. The resulting artefact, after the automatic composition, is a BPEL workflow application that can be deployed on a BPEL execution engine.

Figure 8.3 illustrates the Padus weaver architecture.

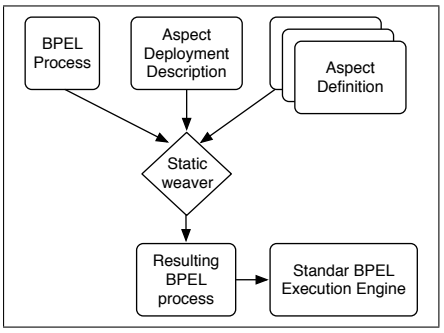


Figure 8.3: Padus Weaver Architecture.

In BPEL applications, the analysis functions are added before or after an activity, thereby the control flow of the workflow application is changed.

8.4 Infrastructure for Enacting MonitA Specifications

We present the overall strategy for enacting the MonitA specifications by introducing the main components involved in the technical approach.

Figure 8.4 illustrates the main elements of the technical approach when executing M&A concerns incorporated in the workflow application.

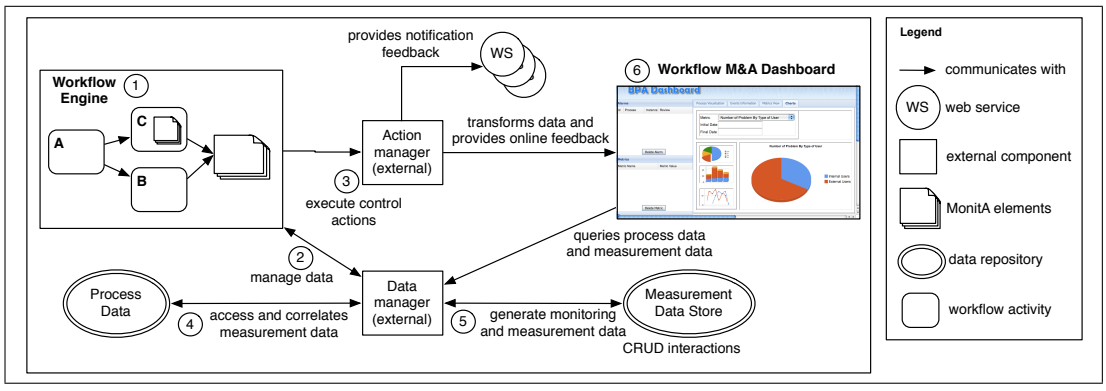


Figure 8.4: Architecture for Monitoring and Analysis Online

While workflow instances are in execution within a workflow engine (Figure 8.4 part 1), the M&A concerns are part the workflow application instances. The M&A concerns access to an interface to manage the measurement data (Figure 8.4 part 2) and to an interface to execute notification actions (Figure 8.4 part 3). These interfaces interact with the workflow engine to access execution information (Figure 8.4 part 2), with a workflow data system to retrieve the workflow application information (Figure 8.4 part 4), a measurement data system to manage analysis information (Figure 8.4 part 5), and a dashboard to visualize the M&A concerns (*e.g.,* measurements, alarms) for identifying potential workflow improvements (Figure 8.4 part 6).

The following sections detail the different systems and environments involved in the MonitA execution platform such as: the specification environment, the measurement data store system, and a dashboard.

8.4.1 Specification Environment

We created a textual editor to specify M&A concerns by using our MonitA DSL. The editor allows the navigation between process model elements and data model elements.

The MonitA-DSL editor offers the tool support required to navigate through the flow entities of the BPMN process model. When an element is selected, the name is displayed without spaces. However, this imposes some naming convention restrictions such as that the elements in the process model cannot have duplicated names or that the element has to be selected with a combination of properties (*e.g.,* name-id, name-reference).

8.4.2 Measurement Data Store System

Ensuring analyzability through workflow applications requires the creation of a data store system that covers the relevant aspects of the monitoring and analysis activities. We created a measurement data store system to record relevant monitoring and measurement information regarding monitoring events, reference to workflow entities, measurement variables, indicators, and produced notification actions into a centralized measurement data store. We do not consider a data integration solution since it would be heavyweight. This measurement data store is easily accessible.

The information is stored using timestamps as a mechanism to reason about the history. The information stored in the measurement data store is associated to common properties such as the workflow application identifier and instance identifier.

Figure 8.5 illustrates the measurement data model that we created to store the monitoring and measurement information relevant to the execution of workflow applications.

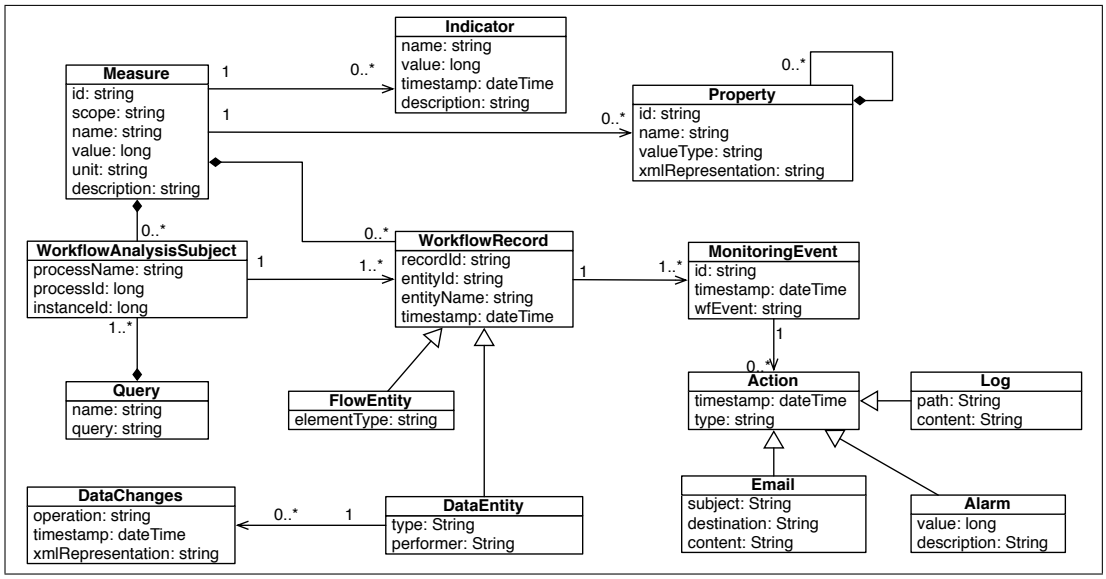


Figure 8.5: Measurement Data Model.

The information stored in the measurement data store falls into the following dimensions:

- *Workflow Analysis Subjects.* A workflow analysis subject represents each workflow instance that was monitored. Each workflow subject is identified by the process name, the process identifier, and a workflow instance identifier. These attributes are used to identify and classify the information stored in the measurement data model by referring to characteristics of the corresponding workflow elements.

- *Workflow Records.* A workflow record represents the execution of a flow entity or an operation performed on a data entity during the execution of a workflow analysis subject. Each workflow record references the actual domain entity with an entity name, entity type, and the information of the workflow analysis subject. When the entity type is a flow entity, the type of element (Activity, Event, Gateway) is also stored, whereas the data type and performer are stored when the entity type is a data entity.

- *Monitoring Events.* A monitoring event represents an intercepted workflow event and the data captured in it. Each record is associated with the workflow subject and entity involved in the workflow event, the event type (*e.g.,* start, create, read, update), and a timestamp.

- *Data Changes.* A data change record represents a data entity that was produced or changed during the workflow execution. Multiple records are stored for the same data entity representing each captured workflow event

of such data entity. Each record is associated with an operation performed, a timestamp, the reference to the flow entity involved in the event, and the representation of the information affected in the data entity.

When the data is managed by an external system to the workflow engine, the data related event can not be intercepted at a fine-grained level since we can not enhance the external systems with the monitoring concerns. Thus, the specific changes on attributes of data entities can not be captured. In these scenarios, the workflow data is captured when the activity finalizes and the complete data entity is stored as a DataChanges record. One possibility to avoid storing multiple copies of the same workflow data is to keep only a record of the last modification done in this data.

When the workflow data is managed by the workflow management system, we provide support to intercept fine-grained data events and to store only this information as a DataChanges record.

- *Measurements.* A measurement represents a measure or metric gathered from the workflow enactment. A performance measure represents a unit of measurement by comparing to a standard (*e.g.,* lines of code, seconds). A performance metric represents a composite indicator based upon multiple properties to filter or query the metrics. Each record contains a representation (*e.g.,* xml document) of the data processed. A measurement is composed of one or multiple workflow analysis subjects or a workflow record.

- *Indicators.* An indicator represents comparisons done in a measurement according to evaluation rules. Indicators are represented when a measurement is set to a prescribed state based on the occurrence of a specific condition (*i.e.,* a flag is one example of an indicator). For example, the generation of an alarm to be visualized in a dashboard if a measurement exceeds the specified conditions for which the evaluation rule is set. An indicator is stored with a measurement value and a timestamp.

- *Actions.* An action record represents a notification action that was produced during the workflow execution. These control actions include e-mails, logs and alarms, which are associated to a workflow analysis subject and a time stamp where they were generated.

- *Queries.* A query record represents the structure of a query related to one or multiple workflow analysis subjects.

The information in the measurement data model is created by the monitoring activity that captures information from the workflow execution. The measurement information is also created during the execution of the analysis functions.

8.4.3 Workflow Monitoring and Analysis Dashboard

We designed and implemented an interactive web-based dashboard to facilitate workflow analysts to be aware of the changes in the quality of workflow applications through the strategic measures defined and evaluated on them. These workflow execution quality changes can de established through the evaluation of application-specific measurements.

Figure 8.6 illustrates the visualization of the M&A concerns defined for the trouble ticket scenario.

Figure 8.6: Visualization of Monitoring and Analysis Concerns.

The following describes the set of functionalities provided by the dashboard:

1. Display measurements by capturing information from the measurement data store, workflow engine and workflow variables. These measurements are visualized in a pie and chart representation.

2. Display alerts associated with indicators defined in the MonitA specifications.

3. Select and display a representation of the workflow application with analysis information associated with each activity (*e.g.*, activities are highlighted to indicate performance measures evaluated in true).

4. Build queries on-demand by combining information from the workflow engine (*e.g.*, workflow instances), workflow variables (*e.g.*, expert assigned to a

problem), and measurement information previously defined (*e.g.*, number of problems reported by area of expertise). The dashboard offers a drag and drop functionality to build measurements that were not incorporated in the first place, in the M&A concerns implementation.

This information helps identifying potential improvements to workflow applications. For example, for a high number of problems reported in a specific area of expertise, the manager of that area can visualize this information immediately and decide to stop the execution of the workflow application to include a new quality assurance activity.

Figure 8.7 illustrates the main elements of the technical approach when specifying M&A concerns from the dashboard.

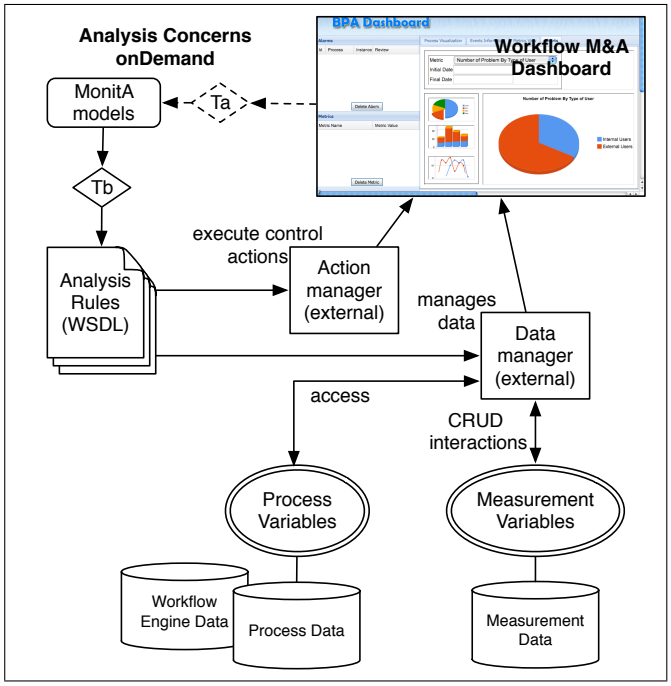


Figure 8.7: Architecture of Monitoring and analysis on demand

All the measurement that is queried on demand through the dashboard is not incorporated into the workflow application. We developed a set of components that access and retrieve directly the measurement data. The MonitA workflow developer provides the libraries required to interact with the dashboard.

8.5 Summary

The validation of the defined architecture for creating a MonitA generative infrastructure was performed by targeting two different workflow platforms. We

used our generative strategy to create the infrastructure required to automate the implementation of MonitA specifications into JPDL workflow applications and into BPEL workflow applications. The implementation of these MonitA infrastructures are an instantiation of the generative strategy described in the previous chapter. This validates the challenges defined for implementing MonitA specifications. These challenges correspond to: a) target diverse execution platforms (*i.e.,* BPEL, JPDL), b) modularize the MonitA code, c) customize the workflow code, and d) access and navigate through workflow data and measurement data.

We have presented the elements of the infrastructure created for enacting MonitA specifications. We created a DSL editor to specify M&A concerns, a measurement data store system to manage historic measurement information, and a dashboard to visualize this information.

The following chapter presents the evaluations performed to validate our our research goals.

Part IV

Validation and Conclusion

Chapter 9

Validation

Now that we have detailed our approach to specify monitoring and analysis (M&A) concerns in workflow applications, we dedicate this chapter to present whether our approach fulfills the goals we defined in section 1.3.

Figure 9.1 illustrates an overview of the problems, goals and assessment goals that are involved in this research and the relation between them.

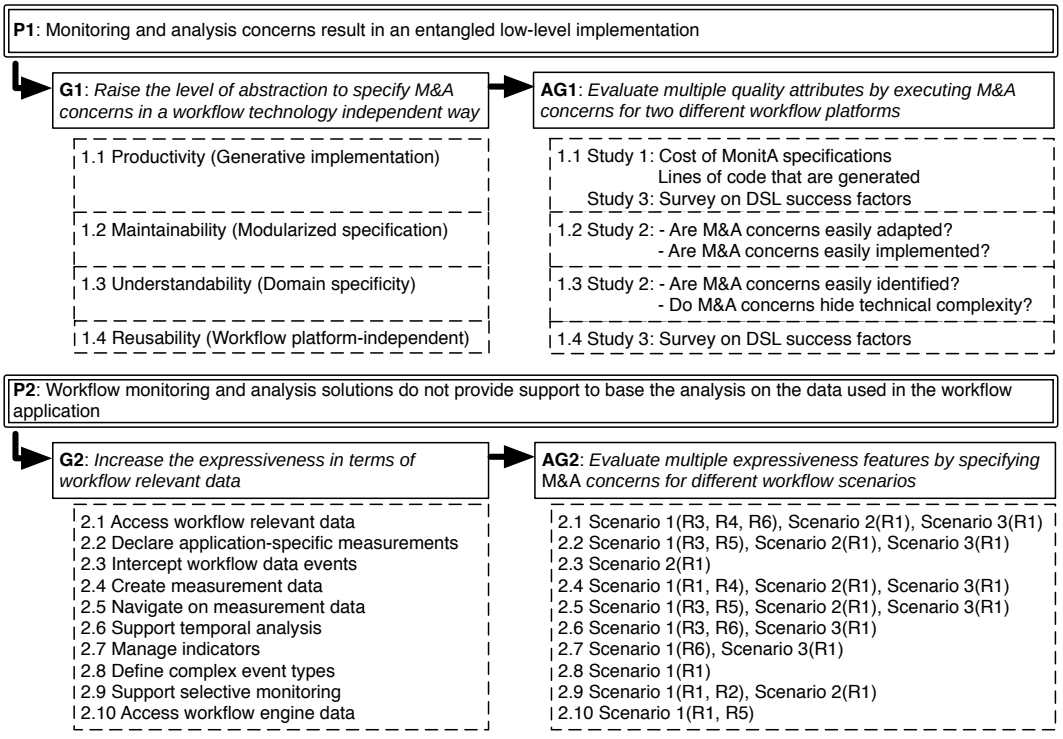


Figure 9.1: Relation between Problems, Goals, and Assessment Goals.

We evaluate our research goals by using both qualitative and quantitative criteria.

Sections 9.1, 9.2, and 9.3 present the application of our approach in a set of workflow applications in order to validate its applicability in different domains. The application of MonitA in different workflow scenarios validates the goal of increasing the expressiveness of M&A concerns in terms of workflow relevant data (*i.e.,* G2). There are mainly 10 sub-goals that are associated the goal G2 to evaluate the expressiveness of MonitA specifications.

Our assessment goal (*i.e.,* AG2) is to determine a large set of possible M&A concerns that can be specified in terms of workflow relevant data. We used the MonitA language to express M&A concerns into three workflow applications: two well known and documented, and one used in a University. These scenarios are of great relevance because they correspond to typical and real workflow applications. We implemented two of these workflow applications in different workflow platforms to validate our approach, whereas the third application was implemented by external people.

Sections 9.4, 9.5, and 9.6 discuss a number of case studies conducted to assess the goal of raising the level of abstraction for specifying M&A concerns (*i.e.,* G1). For each case study we present the context, design, execution, and the results obtained in connection with our assessment goals. The goal G1 is composed of 4 sub-goals such as productivity, maintainability, understandability, and reusability.

Our assessment goal (*i.e.,* AG1) is to apply our approach to different workflow platforms to evaluate to which extent the M&A concerns are applicable in a workflow technology independent way and whether they can be specified compactly. We have validated the quantitative criteria (*i.e.,* Study 1) related to the sub-goals against the following three hypotheses:

- The use of the MonitA language results in a decreased development time for specifying M&A concerns.
- The initial setup cost of a MonitA infrastructure for a new workflow platform can be neglected if more than five M&A requirements are specified.
- The lines of code of a monitoring and analysis specification is smaller when using the MonitA language and thus reduces the code base to be maintained.

These hypotheses were assessed in the context of two separate implementations of the MonitA generative infrastructure. One MonitA execution infrastructure was created for a JPDL workflow platform, whereas another execution infrastructure was created for a BPEL workflow platform. These implementations were performed with the purpose of analyzing the genericity of our approach to be adapted to a number of workflow applications and platforms.

There are a number of threats to the validation of the assessment goals. We describe and analyze these in each case study. The threats we discuss are currently under investigation and will be used as drivers for future improvements

to the MonitA execution platform. On a side note it should be mentioned that most of the threats we report also manifest themselves in environments where MonitA is not used. As such we think it is fair to say that the assessment goals were adequately validated by the different case studies.

9.1 Scenario 1: Trouble Ticket Workflow Application

A short description of the trouble ticket application [Nor98] and its graphical representation in BPMN were presented in section 1.2. In this section we present a more detailed description of this workflow application.

A Trouble Ticket workflow starts when a problem is identified by an internal or an external user (originator) in a software product. The originator uses a form in the *Describe Problem* activity to detail and record the problem. This generates a new workflow instance. An instance ID allows customer support (CS) to check the progress of work on a particular trouble ticket. Once the problem details are provided, quality assurance (QA) checks the trouble ticket report in the *Reproduce Problem* activity. If the problem can be reproduced the workflow continues in the *Provide Problem Solution* activity, otherwise the workflow goes to the *Correct Problem Description* activity. If the problem has a known solution, it can be added at this stage and the workflow continues to the *Communicate Ticket Result* activity. If the problem is identical to another one, it is recorded as its corresponding duplicate and the workflow continues with the *Verify Problem Solution* activity.

If the *Correct Problem Description* activity is reached, the originator must clarify the problem and the workflow gets back to the *Reproduce Problem* activity. When the workflow reaches the activity *Provide Problem Solution*, a development expert ensures that the problem belongs effectively to this area of expertise. If not, the problem is reassigned, the responsible of the activity changes, and the trouble ticket stays in this state until a resolution is determined. Once there is a resolution, the *Verify Problem Solution* activity is executed by QA to proceed to communicate the results, or to get back to identify the problem and resolution in case it is not solved. The *Communicate Ticket Result* activity notifies the results of the workflow to the originator. The *Audit Trouble Ticket Handling* activity, which is executed in parallel, determines whether the problem has to be included in a knowledge repository.

9.1.1 Monitoring and Analysis Requirements

Additionally to the M&A requirements presented in the preliminary sections, the following are a set of M&A concerns defined for the trouble ticket application. These M&A concerns cover the main monitoring and analysis situations:

- R1: Capture the processing time for each activity (Listing 9.1).

```
concern myConcerns import AssociationModel, MeasureDataTypesModel

transient entity dateTime startTime
persistent entity duration pTime

on start/finish [root.!Activity]
   trigger computeProcessingTime(t=dateTimeType.now(), type=engine.wfEvent())

mmcfunction computeProcessingTime (dateTime t, string type)
  if type=='start'
  then startTime = t;
  else pTime = t - startTime; endif
endfunction
```

Listing 9.1: MonitA Specification: Processing Time in the Trouble Ticket Scenario.

The main important characteristics in the specification of this requirement are quantification, compositional events, and declaration of measurements. Both measurements indicate that each value of the measure is associated to a particular flow entity within a workflow instance. The measurement variable named `pTime` is persisted in the measurement data store system. This MonitA specification declares how to compute the processing time for all activities in the workflow application. The use of quantification to refer to a set of monitoring subjects decreases the necessity to create a monitoring event for each monitoring subject involved in the workflow application. In the same way, the compositional event (*i.e.,* `start/finish`) reduces the specification code that would be necessary to observe the workflow application at the beginning and at the end of each activity.

- R2: Add an event log each time the severity of a problem is changed for any activity (Listing 9.2).

```
on finish [FlowEntity fe | root.!Activity writes vProblem.severity]
   trigger auditProblems(fe)

mmcfunction auditProblems(FlowEntity act)
  trace(path='/log', message='severity modified in activity'+act.name);
endfunction
```

Listing 9.2: MonitA Specification: Monitoring Event Pattern in the Trouble Ticket Scenario.

The main important characteristic in the specification of this requirement is quantification. This MonitA specification declares how to monitor only the activities that modify the value of a particular attribute in a data entity

(*i.e.,* `vProblem`). The use of quantification to refer to a set of monitoring subjects (*i.e.,* activities) facilitates the specification of monitoring events in terms of data events. This set of monitoring events is independent of the changes that can be done on the workflow application (*e.g.,* remove an activity).

- R3: Send a notification if the number of reports generated by a user in the last month is higher than 5 (Listing 9.3).

```
persistent instance Collection<Report> reports

on finish [root.SubmitForm]
   trigger ticketByUser(user=root.SubmitForm:vReport.submitter)

mmcfunction ticketByUser(string user)
  int numbertickets = reports->select(Report r | r.originator.name == user
                          and r.dateReceived>dateTimeType.now().month(1-));
  if numbertickets > 5
  then notify(destination='xx@yy.com',subject='high number of tickets',
      content='the tickets generated by the user'+
                             user + ' in the last month is higher than 5');
  endif
endfunction
```

Listing 9.3: MonitA Specification: Temporal Analysis in the Trouble Ticket Scenario.

The main important characteristics in the specification of this requirement are navigation on measurement data, temporal analysis, and creation of application-specific measurements. The measurement variable named `reports` is declared in terms of workflow relevant data such as a data entity with Report data type. This variable indicates that each value of the measure is associated to a workflow instance. The navigation on the historic measurement information is done through the reports variable. Using this variable it is possible to retrieve the collection of data entities with Report data type that have been persisted in the measurement data store system. This specification selects a subset of the collection of reports by filtering those ones reported by a specific user and created in a specific interval of time. The collection of reports is filtered by considering the measurement created in the last month.

- R4: Trigger an alarm if a ticket is assigned to a developer with more than 3 tickets without resolution (Listing 9.4).

  The main important characteristic in the specification of this requirement is the management of variables that have not been created. A measurement action in this MonitA specification declares how to navigate on the collection of data entities with Problem data type assigned to a particular developer.

```
persistent instance Collection<Problem> problems

on finish [root.SubmitForm]
      trigger ticketAssigned(developer=root.SubmitForm:vProblem.expert)

mmcfunction ticketAssigned(string developer)
  int numberProblems = problems->select(Problem p | p.expert == developer
                                                    and p.resolution==null);
  if numberProblems > 3
  then alert(variable=numberProblems, message='the '+developer
          +' developer has more than 3 tickets assigned without resolution');
  endif
endfunction
```

Listing 9.4: MonitA Specification: Null Variable Values in the Trouble Ticket Scenario.

The `null` expression is used to filter the collection of problems without resolution.

- R5: Notify if the processing time for providing a resolution is twice the average processing time to identify a problem and provide a resolution. Consider that the processing time was computed in the requirement R1 (Listing 9.5).

```
//persistent entity duration pTime

on finish [root.IdentifyProblemandResolution] trigger avTime()

mmcfunction avTime()
  duration av = pTime.allInstances()->select(Measure m |
          m.activityName =='IdentifyProblemandResolution') / engine.instances;
  if pTime >= 2*av
  then notify(destination='yy@xx.com', subject='high resolution time',
              content='too much time providing a resolution');
  endif
endfunction
```

Listing 9.5: MonitA Specification: Navigation on Measurement and Workflow Information in the Trouble Ticket Scenario.

The main important characteristics in the specification of this requirement are navigation on measurement information, usage of workflow engine information, and definition of measurements on top of existing measurements. This MonitA specification declares how to retrieve the collection of measure values associated with the `pTime` variable. The `allInstances` expression is used to access the collection of values of the pTime variable for all the flow entities executed by the workflow instances. This collection is filtered by the measurements associated with the `IdentifyProblemandResolution` activity. The information about the number of workflow instances executed by the workflow engine (*i.e.,* `engine.instance`) and the pTime variable are used to compute a new measurement named `av`.

- R6: Trigger a notification if the indicator that evaluates when the number of problems reported by area of expertise is higher than 10 has been triggered more than 3 times in the last month. Evaluate this indicator when a problem is created (SubmitForm activity) (Listing 9.6).

```
on finish [root.SubmitForm]
   trigger evaluateIndicators(area=root.SubmitForm:vProblem.area)

mmcfunction evaluateIndicators(string area)
  ProblemsByArea p1 = pba.current();
  //The indicator i1 was created previously
  int temp = p1.indicators->select(Indicator ind | ind.name=='i1'
             and ind.timestamp > dateTimeType.now().month(1-))->size();
  if temp > 3
  then notify(destination='yy@xx.com', subject='indicator triggered frequently',
              content='indicator triggered frequently in the last month');
  endif
endfunction
```

Listing 9.6: MonitA Specification: Navigation on Indicators in the Trouble Ticket Scenario.

The main important characteristics in the specification of this requirement are the access to information under execution and the evaluation of indicators. This MonitA specification declares how to retrieve the value of the measurement of the `pba` variable for the workflow instance under execution. A collection of indicators associated to this measurement is retrieved and filtered by the name of the desirable indicator and a temporal constraint.

These M&A concerns can be used to evaluate the workflow application at runtime and to provide feedback to workflow analysts. These M&A concerns can be used to define potential improvements regarding a particular business goal. For example, for a high number of problems reported in a specific area of expertise, the manager of that area can visualize this information immediately and decide to stop the execution of the workflow application to include a new quality assurance activity. This new activity can for example be assigned to a new resource that can help to reduce the number of problems in that specific area.

Other examples of M&A concerns can be targeted to reduce the time spent fixing a problem. This requires capturing the operational workflow execution time (measuring), right after the execution of the *Create Ticket* activity (monitoring). Then, a timer must be started to evaluate when this time is higher than 1 day and to notify this performance information by email (control). For execution times higher than 1 day, the workflow analyst can decide to scale the problem by reassigning responsibilities, by changing the priority of the ticket, or by involving a new resource into the process to solve problem tickets.

9.1.2 Generative Implementation and Composition

The trouble ticket workflow application is implemented in JPDL. Thus we use the MonitA-JPDL infrastructure to generate and compose automatically the MonitA specifications into the JPDL implementation code (see Section 8.2).

The MonitA specifications are automatically translated into aspect code by using the aspect language AspectJ. This aspect language can be integrated with Java as the underlying implementation language for JPDL activities. The implementation of M&A concerns and the workflow implementation are composed to generate a new trouble ticket application instrumented with M&A concerns. Finally, this instrumented workflow application was executed in an jBPM workflow engine to start analyzing its execution.

9.2 Scenario 2: Loan Approval Workflow Application

This section describes another workflow application used to validate the expressiveness of our workflow analysis approach. This workflow application corresponds to a typical and real life loan process [IBM02] used to manage loans in a financial company.

Figure 9.2 illustrates the specification of the BPMN process model for the loan workflow application.

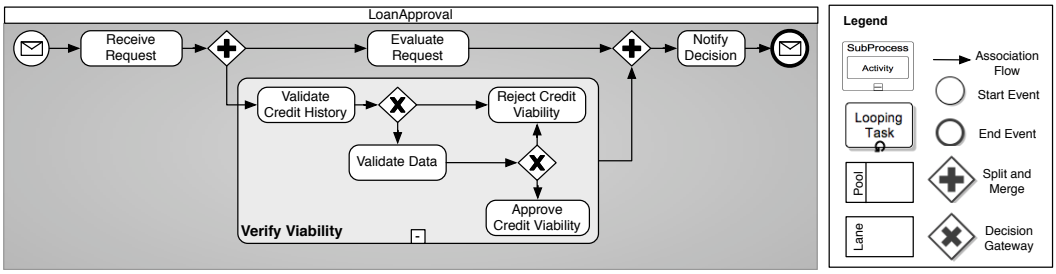


Figure 9.2: Loan Workflow Application.

A Loan workflow application starts with a loan request, where a client uses the bank web page to introduce his personal information and the amount requested. Then, the workflow application executes a set of activities to decide whether the loan is approved or rejected. The first task evaluates the information provided by the client by using a set of rules defined by the bank (*e.g.,* if the client income is less than 10% of the requested amount). At the same time, a subprocess validates the loan viability by evaluating the risk associated with the requester. One task in this subprocess validates the qualification of the client against a credit entity, and then another one validates the authenticity

of the data provided by the client. Finally, the loan expert bases the approval decision on the information gathered in the previous activities and the client is notified.

This example introduces many process elements such as splits, merges, a sub-process, activities, and decision gateways. We will use this example to illustrate the data association model, which describes the workflow variables used by the workflow application and their use by each flow entity.

9.2.1 Data Association Model

Listing 9.7 illustrates the data association model created to complement the loan approval process model with the variables used by the flow entities in the workflow implementation. This specification illustrates how a workflow developer uses our data association DSL to define the variables manipulated by the workflow application and the operations that flow entities perform on these variables. The activities correspond to flow entities contained in the process model (LoanApprovalBpmnModel).

```
process LoanApproval import LoanApprovalBpmnModel, LoanApprovalDataTypesModel

//Specify process variables
instanceScope Client vClient
instanceScope Request vRequest
instanceScope Evaluation vEvaluation

//Associate data entities with flow entities
ReceiveRequest creates(vClient, vRequest)
EvaluateRequest reads (vClient) writes (vRequest)
ValidateCreditHistory reads (vClient) creates (vEvaluation)
ValidateData reads (vClient) writes (vEvaluation)
RejectCreditViability writes (vRequest)
ApproveCreditViability writes (vRequest)
NotifyDecision writes (vRequest)
```

Listing 9.7: Data Association Specification for the Loan Scenario.

This specification illustrates that there are 3 workflow variables that can be accessed by each workflow instance and that are visible for all flow elements within a workflow instance. All variable data types (*e.g.,* Client) correspond to data entities defined in the data types model (*i.e.,* xml schema) named `LoanApprovalDataTypesModel`. The `vClient` variable is declared with a complex data type named Client, which contains a sequence of elements (*i.e.,* int identification, int annualEntrance, string lastName, string firstName, int age) with their associated primitive data types. The `vRequest` variable is declared with a complex data type named Request, which contains a sequence of elements (*i.e.,* int requestAmount, boolean loanResult, string description) with their associated primitive data types. The `vEvaluation` variable is declared

with a complex data type named Evaluation, which contains a sequence of elements (*i.e.,* int creditHistory, boolean criminalRecord) with their associated primitive data types.

This data association model also illustrates the association between activities and workflow variables. For example, the above specification indicates that `vClient` variable is created by the `ReceiveRequest` activity, and read by other 3 activities (*i.e.,* EvaluateRequest, ValidateCreditHistory, ValidateData).

9.2.2 Monitoring and Analysis Requirements

A strategic analysis goal defined by a workflow analyst is to increase the number of approved loan requests to more than 50% of the total requests. The MonitA application developers have to identify and specify the M&A concerns that support this goal. The following illustrates the requirement identified by a MonitA application developer and its corresponding specification:

- R1: Compute the rate of loans grouped by a specific decision (*i.e.,* approved or rejected). This decision must be captured for every workflow instance when the corresponding workflow variable (*i.e.,* loanResult) is changed in the NotifyDecision activity. Then, a measurement action must evaluate if the rejected requests rate is higher than 50%, which means that the causes for the rejection (*e.g.,* the requested amount is too high) must be verified. When this condition is triggered, a control action must be taken to notify via email such situation to the quality assurance area (Listing 9.8).

```
concern WF1 import LA_DataAssociationModel

presistent multiinstance int approvedLoan
presistent multiinstance int rejectedLoan

on change [root.NotifyDecision:vRequest.loanResult]
   trigger af1(root.NotifyDecision:vRequest.loanResult)

mmcfunction af1(boolean decision)
  if decision==true then
    rejectedLoan = rejectedLoan+1;
  else
    approvedLoan = approvedLoan+1;
  endif
  if (rejectedLoan / (rejectedLoan+ approvedLoan)) > 0.5
  then notify(destination= 'quality@xx.co', subject= 'Requests Rejected',
              content='High Rejected Requests Rate');
  endif
endfunction
```

Listing 9.8: MonitA Specification: Measuring Loans by Decision in the Loan Approval Scenario.

This requirement is associated with application-specific measurements since they involve information that the workflow is modeling. The measurement

variables named `approvedLoan` and `rejectedLoan` are dependent of workflow relevant data such as the workflow variable `loanResult`. The measurement variables indicate that each value of the measure is associated to multiple workflow instances. The navigation on the historic measurement information is done through the measurement variables to retrieve and persist the information in the measurement data store system.

The information provided by the execution of the M&A concerns allows the identification of potential improvements (*e.g.,* adapting workflow data constraints, reordering the control flow, reassigning responsibilities) on the workflow application.

9.2.3 Generative Implementation and Composition

In contrast with the trouble ticket workflow application, described in the introduction, the loan approval workflow application was implemented in BPEL. This implementation consists of 25 activities specified in 366 BPEL LOCs. We use the MonitA-BPEL infrastructure to generate and compose the MonitA specifications automatically into the BPEL code (see Section 8.3).

The MonitA specifications are automatically translated into aspects described in the aspect language Padus [BVJ+06], which can be integrated with the BPEL language. The implementation of analysis concerns and the workflow implementation are composed to generate a new instrumented loan workflow application with analysis concerns. Finally, this instrumented workflow application was executed in an Apache ODE workflow engine to start analyzing the execution of the workflow application.

9.3 Scenario 3: Trip Expenses Workflow Application

This section presents the third workflow application we used to validate the expressiveness and usability of MonitA. This workflow application was defined to manage the trip expenses for researchers and professors in a university.

Figure 9.3 illustrates the complete set of activities involved in such a workflow, which is specified in BPMN.

The trip expenses workflow application involves different stakeholders such as: professors, department head, research group head, associate dean, financial director, and financial assistant. In the first activity (*Perform Request*), a professor performs a funding request by filling a web form with the information about the target event and the required budget (*i.e.,* plane tickets, event registration, and trip fees). Then, the request has to be reviewed by the department head related to the originator department to decide on approval

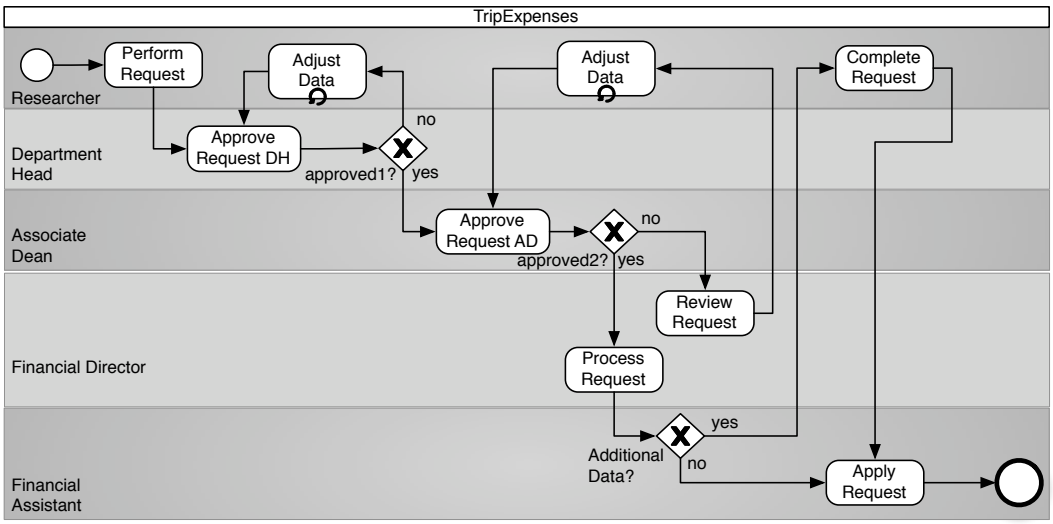


Figure 9.3: Trip Expenses Workflow Application.

or rejection (*Approve Request DH* activity). The department head can associate observations to the request to indicate the information that has to be fixed by the professor before the request is approved. The activity to fix the observations made to the request is performed until the request is approved (*Adjust data* activity). The requests that are approved have to be reviewed by the associate dean, who decides on making observations or on approving the request (*Approve Request AD* activity). If the request is rejected, the financial director has to review the request and to add additional observations regarding the available budget (*Review Request* activity). If the request is approved the financial director has to process the request and indicate if additional information is required (*Process Request* activity). When additional information is required, the professor has to provide this information, otherwise the request is processed by the financial assistant and a tracking number has to be assigned for each budget item (*Apply Request* activity). Once the tracking number has been generated for each budget item (*i.e.,* plane tickets, event registration, trip fees), the request is finished.

Every time a request is assigned, rejected, or approved, a notification is generated and sent to the responsible of the activity activated in the workflow application. The notification is sent daily until the request is handled. A web interface allows all the stakeholders to visualize the requests that have been assigned and processed and the state of each one of them.

9.3.1 Data Association Model

Listing 9.9 illustrates a fragment of the data association model created to complement the trip expenses process model with the variables used by the

flow entities in the workflow implementation. This specification illustrates how a workflow developer uses our data association DSL to define the variables manipulated by the workflow application and the operations that flow entities perform on these variables. The activities correspond to flow entities contained in the process model (LoanApprovalBpmnModel).

```
process TripExpenses import TripExpensesBpmnModel, TripExpensesDataTypesModel

//Specify process variables
instanceScope Request vRequest
instanceScope RequestUserInfo vRequestUserInfo
...

//Associate data entities with flow entities
PerformRequest creates (vRequest, vRequestUserInfo)
ApproveRequestDH reads (vRequest) writes (vRequest)
AdjustData writes (vRequest)
...
```

Listing 9.9: Fragment of the Data Association Specification for the Trip Expenses Scenario.

This specification illustrates that there are 2 workflow variables that can be accessed by each workflow instance and that are visible for all flow elements within a workflow instance. All variable data types (*e.g.,* Request) correspond to data entities defined in the data types model (*i.e.,* xml schema) named `TripExpensesDataTypesModel`. For example, the complex data type named Request contains a sequence of elements (*i.e.,* dateTime date, string id, string generalState, RequestUserInfo rui, Observation [0..*] obs, BudgetRequest [0..*] br, RequestType rt, DivulgationInfo di) with their associated primitive and complex data types. This data association model also illustrates the association between activities and workflow variables. For example, the above specification indicates that the `vRequest` variable is created by the `PerformRequest` activity, and modified by other activities such as `ApproveRequestDH` and `AdjustData`.

9.3.2 Monitoring and Analysis Requirements

The following are a set of M&A concerns defined for the trip expenses application:

- In the workflow application, a request is automatically rejected if more than 2 requests have been created by the same professor in the current year. In terms of monitoring and analysis, a notification has to be sent to the professor and to the associate dean if this rejection scenario is generated more than 3 times in the same year (Listing 9.10).

```
concern RequestsManagement import TE_AssociationModel, TE_MeasureDataTypesModel

persistent multiinstance Collection<RejectedByProfessor> rbp

on finish [root.PerformRequest]
   trigger requestsControl(root:vProfesor.externalId)

mmcfunction requestsControl(string id)
  RejectedByProfessor rejected = rbp->select(RejectedByProfessor r |
                                    r.professorId==id)->first();

  //Measurement data creation
  if rejected==null then
    rejected = RejectedByProfessor(number=1, professorId=id);
    rbp->add(RejectedByProfessor t| t=rejected);
  else
    rejected.number = rejected.number+1;
  endif

  //Indicator creation
  int temp1 = rbp->select(RejectedByProfessor r| r.professorId==id
                    and r.timestamp.year()==now().year())->size();
  if temp1 > 2
  then Indicator i1 =
        Indicator(measure=rbp,value=temp1,description='Requests > 2'); endif

  //Indicator evaluation
  int temp2 = rejected.indicators->select(Indicator i | i.name=='i1'
                      and i.timestamp.year() == now().year())->size();
  if temp2 > 3
  then notify(destination='xx@yy.com',subject='the request is not allowed',
      content=temp2+'times the maximum requests per year has been exceeded');
  endif
endfunction
```

Listing 9.10: MonitA Specification: Rejected Requests in the Trip Expenses Scenario.

The main important characteristics in the specification of this requirement are a) declaration and creation of application-specific measurements, b) navigation on measurement data, c) creation and evaluation of indicators, and d) temporal analysis. Although this specification contains all these characteristics within the same analysis function, they could be specified in different analysis functions. The measurement variable data type correspond to a data structures defined in the measurement data types model (`TE_MeasureDataTypesModel`). The workflow variables that can be referenced within this specification correspond to the data entities contained in the data association model (`TE_AssociationModel`).

The measurement variable named `rbp` is declared in terms of workflow relevant data since the RejectedByProfessor data type involves workflow-specific information (*i.e.,* externalId attribute of the vProfesor workflow variable). This measurement variable indicates that each value of the measure is associated to multiple workflow instances. The navigation on the historic mea-

surement information is done through the `rbp` variable. Using this variable is possible to retrieve the collection of data entities with RejectedByProfessor data type that have been persisted in the measurement data store system. This specification selects a subset of the collection of `rbp` by filtering those ones generated by a specific professor. The measure value is created if it does not exist, otherwise its value is increased.

This specification defines a temporary variable (*i.e.,* `temp1`) to store the number of requests performed by a a professor in the current year. The `rbp` variable is used to navigate on this historic measurement information by filtering the requests created in a specific interval of time (*i.e.,* current year). An indicator is created if the value of the temporary variable `temp1` is higher than 2. A collection of indicators associated to the filtered measurement (*i.e.,* rejected variable) is retrieved, which are filtered by the name of the desirable indicator and a temporal constraint. If this specific indicator has been created more than 3 times in the current year, a notification action is triggered.

Additional M&A requirements identified for these workflow scenarios can be found at https://soft.vub.ac.be/soft/members/oscargonzalez/research.

9.3.3 Generative Implementation and Composition

The trip expenses workflow application was implemented in JPDL. We used the MonitA-JPDL infrastructure to generate and compose automatically the MonitA specifications into the JPDL code (see Section 8.2).

The MonitA specifications are automatically translated into AspectJ aspects, which can be integrated with the Java code specified for the underlying implementation. The implementation of analysis concerns and the workflow implementation are composed to generate a new trip expenses workflow application instrumented with M&A concerns. The resulting workflow application was executed in a jBPMN workflow engine to start analyzing the execution of the workflow application.

9.4 Study 1: Measuring Development Costs by Using MonitA

This section presents the case study performed to reason about the success for reducing the development costs of MonitA specifications.

The development costs (productivity) in DSLs helps developers to specify domain requirements that are normally time-consuming to implement [JB88]. Typically, the DSL specifications generate automatically the corresponding source code to reduce development costs and time-to-market. The measures

associated to productivity quantify the monitoring and analysis specification inputs required to produce the output artifacts in the workflow implementation. We selected the following quantitative criteria to measure productivity:

- Measure the time required to specify and implement M&A concerns.
- Measure the number of generated lines of code (LOCs) against the LOCs added manually.

9.4.1 The Exploratory Study

We evaluated our approach against two different research experiments for each target workflow platform (*i.e.,* BPEL, JPDL). In the first research experiment, application developers specify the M&A concerns directly in the workflow implementation. In the second research experiment, the same application developers use MonitA to specify the M&A concerns. In both research experiments, we evaluated the above criteria related to productivity. These research experiments were performed in two different workflow applications (*i.e.,* trouble ticket, loan approval).

Set-up for the JPDL Workflow Platform

We asked 5 workflow developers to perform these research experiments for the trouble ticket workflow application. These workflow developers were experts in workflow applications implemented in JPDL. The manual specification in JPDL of the M&A concerns involved the five developers, whereas the specification of M&A concerns in MonitA was performed by one of these developers.

Set-up for the BPEL Workflow Platform

We asked 1 workflow developer to perform these research experiments for the loan approval workflow application. This workflow developer was expert in workflow applications implemented in BPEL. This workflow developer performed the manual specification in BPEL of the M&A concerns as well as the specification of M&A concerns by using MonitA.

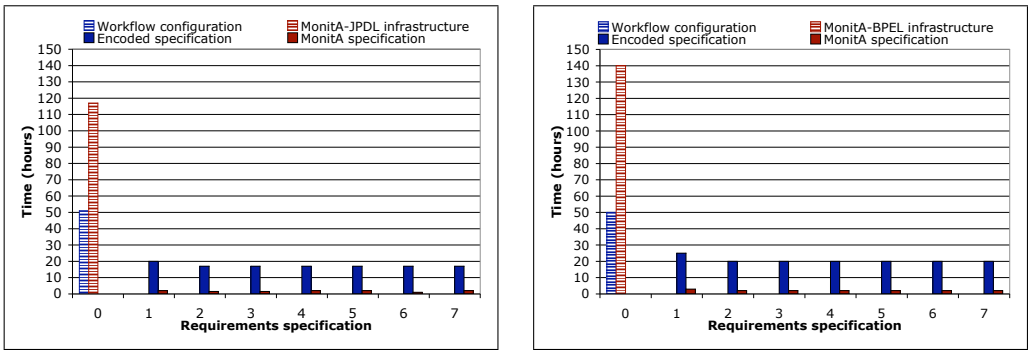
Despite the programming skills of these workflow developers, they had to learn about the workflow technologies and about MonitA.

9.4.2 Quantitative Results

The following illustrate the results after reasoning about the productivity properties according to the defined quantitative criteria.

Development Costs in Terms of Time Criteria

Figure 9.4 illustrates the empirical results obtained in terms of time criteria.



a) JPDL platform b) BPEL platform

Figure 9.4: Trend of Specification Time.

Results for the JPDL Workflow Platform. The manual implementation in JPDL of the three initial M&A requirements for the trouble ticket workflow application (see section 1.2) took around 51 hours. These requirements comprise around fifteen M&A concerns. In contrast, the implementation of these M&A requirements by using the MonitA generative infrastructure took around 2 hours. In order to stablish a more realistic comparison between both research experiments, we took into account the time spent to create the MonitA-JPDL generative infrastructure involved in the second research experiment. The creation of the MonitA-JPDL generative infrastructure took around 117 hours and facilitates the automatic implementation of M&A concerns into JPDL workflow applications.

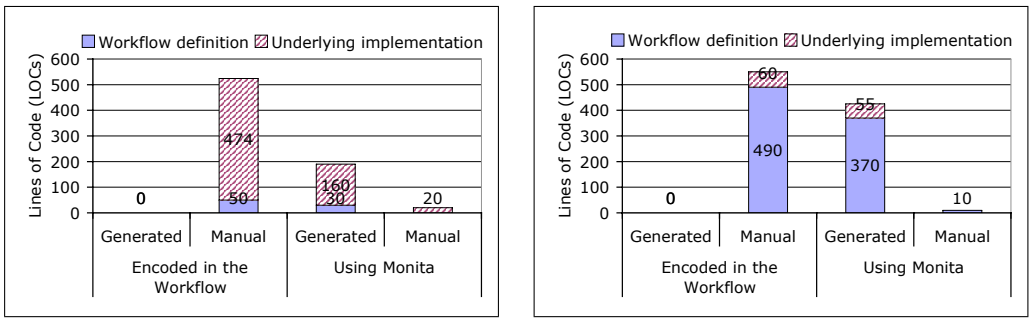
Results for the BPEL Workflow Platform. The manual implementation in BPEL of the M&A requirement for the loan approval workflow application (see section 9.2) took around 50 hours. In contrast, the implementation of the corresponding M&A concerns by using the MonitA generative infrastructure took around 1 hour. In order to stablish a more realistic comparison between both research experiments, we took into account the time spent to create the MonitA-BPEL generative infrastructure involved in the second research experiment. The creation of the MonitA-BPEL generative infrastructure took around 140 hours and facilitates the automatic implementation of M&A concerns into BPEL workflow applications.

These results show that there is a high implementation time to create the generative infrastructure for a new workflow platform. Nevertheless, this creation cost can be recovered after five MonitA specifications. Our approach improves the efficiency for specifying and implementing M&A concerns by avoiding wasted time and effort. This is illustrated by the lower time implementing M&A concerns in an existing workflow application and in a new

workflow application of a supported workflow platform. The automatic generation of the corresponding monitoring code reduces development costs and shortens time-to-market.

Development Costs in Terms of Size Criteria

Figure 9.5 illustrates the productivity empirical results in terms of LOCs criteria.



a) Manual vs Generated Code in the JPDL platform

b) Manual vs Generated Code in the BPEL platform

Figure 9.5: Empirical Evaluation in terms of Size Measures.

Results for the JPDL Workflow Platform. Around 49 JPDL LOCs and 474 Java LOCs were manually added in the JPDL workflow implementation to specify the initial M&A requirements of the trouble ticket scenario (see section 1.2). In contrast, the number of lines of code generated from the MonitA specification is 30 JPDL LOCs plus 160 Java LOCs in the underlying implementation. 20 Java LOCs were added manually, thus, the reduction of manually encoded LOCs in JPDL is around 95%.

Results for the BPEL Workflow Platform. Around 490 BPEL LOCs and 60 Java LOCs were manually added in the BPEL workflow implementation to specify the M&A requirements of the loan approval scenario (see section 9.2). In contrast, the number of lines of code generated from the MonitA specification is 370 BPEL LOCs plus 55 Java LOCs in the underlying implementation. 10 BPEL LOCs were added manually, thus, the reduction of manually encoded LOCs in BPEL is around 98%.

These results show that the MonitA generation process improve the implementation of M&A concerns since the generated code is optimized to deliver higher quality software. The automation of large parts of the development process for M&A concerns leads to fewer errors in their implementation.

9.4.3 Discussion

The time for specifying M&A concerns and for automating the analysis infrastructure depends on the practices used for programming and the experiment or learning curve in the languages. We have to evaluate the impact on time to specify M&A concerns when some element in our approach evolves (*e.g.,* grammar, workflow language).

The main threat to productivity validity is that the M&A generation process has to be evaluated to identify and deliver potential optimizations on it. We have to evaluate the impact that each element in our generative infrastructure (*e.g.,* transformation) has over the development costs of implementing M&A concerns.

The measurements taken from our research experiments depend on the developers' discipline in programming M&A concerns. Nevertheless, we believe the results are sufficiently reliable since the research experiments show similar results on development costs.

If the workflow implementation is changed and the process model is not updated, the analysis specification is not going to correspond with the workflow implementation. Thus, inconsistencies can arise between the M&A concerns specification and their implementation. This issue is described in detail in the future work section.

9.5 Study 2: Evaluating Maintainability and Understandability

We have analyzed maintainability and understandability in our approach against a set of qualitative attributes.

Maintainability helps workflow developers to add or to alter M&A concerns in workflow applications. We evaluated maintainability by analyzing the following criteria:

- Validate the simplicity to adapt M&A concerns.
- Validate that the specification of M&A concerns is modularized from their implementation.

Understandability helps workflow developers to express M&A concerns in terms of the workflow monitoring and analysis domain to offer a higher-level of abstraction during their specification. We evaluated understandability by analyzing the following criteria:

- Evaluate if the M&A concerns are easily identified.
- Evaluate if the specification of M&A concerns hide technical complexity to workflow developers.

9.5.1 Evaluation Results

Maintainability. Our workflow M&A generation process modularizes the M&A concerns specification and its implementation to enhance their maintainability. The M&A concerns specification can be easily identified and adapted since they are separated from the workflow specification. We have implemented the M&A concerns of both workflow scenarios manually and using MonitA. We saw duplication of code in the manual implementation, which is detrimental for the maintainability by compromising the reliability of the workflow implementation. When changes to M&A concerns are required, their specification is modified in a single place and its implementation can be re-generated and composed with the existing workflow implementation. Thus, there are no clones in the MonitA specification and the implementation of M&A concerns is reliable.

Additionally, the code required to maintain is less in MonitA and it is modularized to ease its maintainability.

Understandability. At a conceptual level the language constructs allow workflow analysts to identify the M&A concerns specification that satisfies the business requirements. At the implementation level, workflow developers easily identify which module implements a MonitA specification. This increases understandability and facilitates reasoning about the specification (syntax) and implementation (semantics) of M&A concerns.

9.5.2 Discussion

Maintainability. Although the MonitA specifications can evolve independently from the workflow specification, when the language grammar is updated, the changes in all existing analysis specifications have to be identified to ensure their consistency. Consequently, the existing MonitA specifications should fit into the new syntax and semantics. Moreover, multiple versions of a process model can be in execution, thus it is necessary to determine which version of the analysis specifications are suitable for each process model. This is for example when there are still workflow instances in execution and the process model is changed for creating instances of the new workflow application.

Understandability. This quality property of the software depends on the intended audience: business manager, domain expert, workflow developer, or user. In our approach the intended stakeholders are workflow developers and workflow analysts. For these stakeholders, understandability is related to hiding technical complexity by increasing the level of abstraction in the specification of M&A concerns. Although understandability is achieved for workflow developers, there is still a big gap to specify M&A concerns directly by workflow analysts.

9.6 Study 3: Evaluating DSL Success Factors in MonitA

We conducted a case study to evaluate DSL success factors in MonitA by means of a questionnaire [1]

9.6.1 Basic Study

The survey was conducted with 2 experimented users of MonitA (*i.e.,* MonitA infrastructure developers) to investigate, in addition to expressiveness and learnability (see section 5.2), DSL success factors such as usability, productivity, and reusability. Both users acted as MonitA application developers.

Table 9.1 illustrates the questions adopted and defined to evaluate multiple DSL success factors in MonitA.

These questions facilitates the measuring of the DSL success factors mentioned in this section. The following explains the considerations adopted for measuring these DSL success factors:

- *Reusability.* To evaluate the reusability of MonitA specifications we asked the MonitA application developers whether they reuse specifications of other projects or they create a new specification (Question 13 and 14). We evaluate if for a MonitA specification application developers create a new MonitA model, or if they look at old models to create a new one, or if they copy an existing model and modify it.

- *Usability.* To evaluate the usability of MonitA we evaluated whether application developers decided for a manual implementation instead of using MonitA (Question 17). We also asked the level of agreement/disagreement, by using a five-point scale, with a set of statements to investigate whether MonitA: eases the specification of M&A concerns (Question 6a), eases the implementation of M&A concerns (Question 6b), is powerful (Question 6c), and if it is difficult to use (Question 9a).

- *Productivity.* We asked a set of questions to application developers in order to measure the productivity of MonitA. The first question was to estimate the percentage of time spent on implementing tasks without using MonitA (Question 18). These tasks correspond to: design M&A concerns, write monitoring events, write analysis function classes, write measurement data representation classes, write configuration files, write M&A logic, and connect M&A code with workflow code. Then we asked the time spent on implementing tasks when using MonitA (Question 19). These tasks correspond to:

[1] The complete questionnaire for MonitA infrastructure developers can be found at https://soft.vub.ac.be/soft/members/oscargonzalez/research

Id	*Question*	*DSL Success Factor*
Q13	Was this a new MonitA project or build on an existing version?	Reusability
Q14	If you start a new MonitA project, how do you proceed?	Reusability
Q15	Did the MonitA user interface help you modeling?	Usability
Q16	Did you use other tools for modeling in this project, next to the MonitA interface?	Usability
Q17	Did you ever consider to use MonitA but decided against?	Usability
Q6a	Did you easily specify the monitoring and analysis requirements using MonitA?	Usability
Q6b	Does MonitA ease the implementation of monitoring and analysis concerns?	Usability
Q9a	Do you agree with the statement: MonitA is difficult to use ?	Usability
Q18	Please estimate the percentage of time spent on the following tasks without using MonitA?	Productivity
Q19	How much time did you approximately spend on the following the tasks when using MonitA?	Productivity
Q20	Please estimate the percentage of code that was generated	Productivity
Q21	How many lines of code did this project consist of?	Productivity
Q22	Do you think the use of MonitA increases the quality of the delivered code?	Productivity

Table 9.1: Questions Used to Evaluate DSL Success Factors in MonitA

(a) design M&A concerns, (b) generate monitoring events, analysis function classes, measurement data representation classes, configuration files, and (c) write M&A logic. Finally, we asked application developers to estimate the percentage of code that was generated (Question 20) to be compared with the number of lines of code in each MonitA project (Question 21).

We asked application developers the level of agreement/disagreement, by using a five-point scale, with a set of statements to investigate whether the use of MonitA increases the quality of delivered code (Question 22). Specifically we asked if they consider that the code is more readable, that fewer errors occur, and that the product complies better with customer requirements.

The measurement of various success factors is done through two five-point Likert [Lik32] scales used by Hermans et al. [HPvD09], which have an addi-

tional neutral option [PK01]. The first scale ranges from strongly disagree, disagree, neutral, agree, strongly agree. The second scale ranges from very often, often, sometimes, seldom, and never.

9.6.2 Results and Discussion

Reusability

There are multiple elements in our approach that can be reused at the model level:

- The association data model that contains the declaration of the workflow variables and their association with the flow entities is specified externally to the M&A concerns model. Thus, it can be reused in multiple MonitA specifications.
- The analysis functions in MonitA are defined once, thus they can be reused by multiple MonitA specifications.
- Generic measurements can be reused in many workflow applications and in many points within a workflow application, whereas application-specific measurements can be reused in many points within a workflow application for a specific domain.

This makes it easier to reuse partial or entire solutions rather than pieces of code. Despite that MonitA specifications are conceptually reused, we observed that MonitA specifications are never reused directly.

Currently monitoring events and analysis functions in MonitA are integrated in a single specification. Separating analysis activities from their connections allows reusing both parts independently. Thus, a mechanism to import and export partially or completely MonitA specifications (*e.g.,* analysis functions) is required to actually reuse MonitA models. This would improve the reusability of MonitA specifications.

The MonitA application developers indicated that the specification of M&A concerns is independent of particular workflow platforms. The existing MonitA generative infrastructures (*i.e.,* MonitA-JPDL, MonitA-BPEL) can be used to integrate M&A concerns for two different workflow platforms.

Usability

Both MonitA application developers indicate that the MonitA editor helped them to specify M&A concerns at a higher abstraction level. Nevertheless, they indicated the necessity to improve the tool support to facilitate the navigation on process and data models.

Productivity

The answer to the questions indicate that the use of the MonitA language and infrastructure reduces the time for implementing M&A concerns. The main reason indicated by both application developers is that using MonitA improves the time implementing repetitive and irrelevant tasks. According to the answers we observe that the time spent on implementing the M&A concerns is reduces from 85% to 40%. This is basically due to application developers concentrate on the domain-specific M&A requirements. This complements the results observed in the case study 2 (see section 9.4) about development costs of M&A concerns. The application developers feel that the MonitA execution platform decreases the complexity to specify and implement M&A concerns, thereby reducing the time and effort to perform these tasks. Moreover, they agree that the use of MonitA increases the quality of M&A code since this code is more readable and complies better with customer requirements. One of the MonitA application developers considers that the workflow application instrumented with M&A concerns becomes less error prone since most monitoring and analysis tasks are automated.

9.7 Summary

Assessing the expressiveness of our DSL requires to use it in multiple scenarios. As mentioned previously, we have used MonitA to declare M&A concerns in three workflow applications. These applications are well known and documented scenarios, which are typically used as a reference to assess work in the domain of workflow management systems. In these scenarios it was possible to specify how to capture information during the workflow execution, how to define and compute application-specific measurements, and how to evaluate these measurements to notify quality issues in the execution of the workflow application. The application of MonitA in different workflow scenarios validated our goal of increasing the expressiveness of M&A concerns in terms of workflow relevant data.

We have discussed a number of case studies to assess our goal of raising the level of abstraction for specifying M&A concerns. We used our approach for different workflow platforms to evaluate that M&A concerns are applicable in a workflow technology independent way.

The definition and materialization of the above experiments resulted in several suggestions to improve the MonitA DSL and its implementation platform. Several suggestions were improved in the current implementation, whereas other ones are considered for future work.

The following chapter compares our workflow monitoring and analysis approach against related work.

Chapter 10

Comparing MonitA with Related Work

This section presents the analysis of various related proposals involving the monitoring and analysis of workflow applications.

We present the definition of multiple characteristics required to understand and evaluate the related work (section 10.1). Although we present an approach for workflow monitoring and analysis at runtime (section 10.2), we discuss related work which specify different techniques to monitor and analyze workflow applications (section 10.3). We also present related work involving data management in process models (section 10.5) and domain-specific languages in other application domains (section 10.6). Finally, we compare our workflow monitoring and analysis approach against related work based on the defined characteristics (section 10.8).

10.1 Monitoring and Analysis Characterization

We distinguish several monitoring and analysis (M&A) characteristics for positioning our approach against related work. These characteristics are a synthesis of existing works. These characteristics are only described in this section, but they are evaluated at the end of this chapter. The characteristics considered to study the related work are characterized according to their nature: execution environment capabilities, and monitoring and analysis capabilities.

Execution Environment Capabilities

- *External specification* of M&A concerns with respect to the workflow application specification. This is required to provide companies with an independent mechanism to capture measurement information, which can help to identify efficiency constraints and to audit workflow applications accuracy. Moreover, M&A specifications must be easily evolved according to the ever

changing business requirements. The M&A approach must be able to react and adjust to these changes.

- *Domain-specific* notations for monitoring and analyzing workflow applications at a higher-level of abstraction. This enables workflow analysts to specify M&A concerns uniformly and independently of particular workflow technologies.
- *Conceptual level* specification to express M&A concerns in terms of the process model rather than in terms of its workflow implementation. Thus, a conceptual specification can be applied to multiple workflow technologies.
- *Events correlation* of workflow events intercepted for the processing of measurements. For example, computing the processing time of a set of activities requires the correlation of two events *activity started* and *activity finished* for each activity instance involved in the measurement. Thus, some patterns to describe this event correlation must be provided.
- *Custom measurements specification* to define the structure of new custom measures required to analyze a workflow application.
- *Custom measurements processing* to define how measurements are computed/calculated.
- *Control actions* specification to define a response to a special situation during the execution of the workflow application.
- *Executable* M&A specification along with the workflow application execution.
- *Generative implementation* of the M&A concerns for a workflow application.
- *Modularized implementation* of M&A concerns along with the workflow application implementation to avoid code duplication.
- *Automatic composition* of M&A concerns with the workflow application when the specification of these concerns is done externally to the workflow application.
- *Automatic event instrumentation* of the workflow management system to monitor a workflow application.
- *Centralized measurement storage* to provide a unified interface to access and manage this information. In this way, the measurement information can be accessed by multiple workflow applications and analysis parties. Users can use measurement information on demand, which enables multiple users to define how and when to use this information.

- *Flexible measurement interface* to ease the access and management of measurement data and its integration with other systems.

- *External processing* of the logic to analyze workflow applications with respect to the workflow management system.

- *Standard based* specification and communication of measurement data to facilitate its interoperability with other systems.

Monitoring and Analysis Capabilities

- *Monitoring at runtime* provides information on how the workflow application is changing continuously. This proactive monitoring of workflow applications integrates several events at runtime to support gathering performance measurements and applying reactive improvement actions without lag time.

- *Monitoring a posteriori* is performed offline for non critical monitoring requirements based on historical information.

- *Fine-grained monitoring* for the intercepting state changes in the workflow variables. These fine-grained complement the interception of state changes in flow entities at the instance level (*e.g.,* when an activity instance moves to a state *finished*).

- *Active monitoring* to capture the measurement information in response to an event occurred during the workflow application execution.

- *On-demand monitoring* to gather the measurement information and running workflow instances information upon request.

- *Technical measurement* to support the analysis of measurements such as system response time, and system load.

- *Organizational measurement* is used for measuring and analyzing the *efficiency* of the workflow application (*e.g.,* idle times, workload analysis).

- *Application-specific measurement* to support the analysis in terms of the data managed by the workflow application. Application-specific workflow analysis measures the *quality* of the workflow application (*e.g.,* workflow data analysis).

- *Inter-organizational measurement* involves a complex interaction of many interconnected (self-contained) workflow applications. The definition of measurements crosses the boundaries of a single workflow application.

- *Operational decision-making* analyses the monitoring and measurement information during the workflow application execution. The operational evaluation of service level agreements (SLAs) help to improve the speed and effectiveness of workflow applications. This also helps to detect operational bottlenecks faster.

- *Informative analysis* provides feedback through notification actions (*e.g.,* send email) about the operational evaluation of monitoring and measurement information. An informative analysis facilitates the identification of potential improvements to prevent problems in the future.

- *Adaptive analysis* uses the operational evaluation of monitoring and measurement information to modify the state of a workflow application to fix a problematic situation during its execution.

10.2 Workflow Monitoring and Analysis at Runtime

The monitoring and analysis at runtime is typically performed by using the logged audit trail data provided by the workflow management systems. The approaches in this area focus on real-time access to workflow performance indicators, interactive and real-time dashboards, and proactive alert generation. An error detection mechanism can be defined by evaluating the loops in the workflow application.

The following sections illustrate different solutions for monitoring and analyzing workflow applications at runtime. These solutions are grouped according to their approach.

10.2.1 Architectures for Business Activity Monitoring

One possibility for analyzing workflow applications is the use of workflow audit trail information. The first survey on analytical opportunities by using audit trail information was provided by McLellan in 1996 [McL96]. McLellan discussed the analysis of historical workflow data stored by evaluating audit trail data in terms of workflow metrics. These metrics perform statical evaluations and detect late cases and overdue tasks at runtime. In addition to the technical M&A dimension presented in this work, our approach deals with the M&A from an application-specific dimension.

Several commercial business process management products and architectures offer solutions for Business Activity Monitoring (BAM) [MHH07] [vdDB06] [LPY07] [JSC03] [MS04]. These solutions offer rich dashboards to visualize

predefined measurements and to write queries on demand to extract the information required for the reporting tools. The information is extracted from audit trails, in which workflow metrics are added to the workflow architectures for analysis. Typically, workflow developers have to instrument the workflow platforms and applications by adding adapters to generate custom workflow events, to add custom measurements, and to send this information to the BAM architecture. These BAM architectures instrument the IT systems that manage workflow applications for supporting their monitoring, measurement and analysis. Nevertheless, the instrumentation of the workflow system is very time consuming since it has to be performed manually for each workflow language and execution engine. Typically, M&A tools use proprietary workflow platforms, thereby these solutions are difficult to use in a wide range of contexts.

The following subsections present different architectures for business activity monitoring.

Monitoring Frameworks

The Process Performance Manager [AG00] is a commercial tool that analyzes performance by calculating predefined ratios, which are both frequency and time-related [vdDB06]. McGregor et al. [MS04] propose a framework based on web services for measuring and analyzing business performance. In this approach, the whole BAM is encapsulated in a decision making system, named Solution Manager framework, which provides a standardized interface for workflow monitoring and analysis. Nevertheless, the workflow platform (*i.e.,* BPEL) has to be extended to transfer the information required by the framework for evaluating performance measurements. We include a data association model in our approach to declare explicitly the data used by the workflow models for easing application-specific M&A specifications.

The work presented in [JSC03] illustrates an agent-based framework that aims at providing continuous analytics for workflow applications. This framework can detect exceptions or special situations in a business environment to compare them with desired management goals. This framework comprises five components for a) analytical processing, b) real-time transformation of workflow events, c) storing workflow-base process metrics, d) tracking all management agents incorporated within the business environment, and e) the visualization of metrics and analytical results in a dashboard. The instrumentation of the workflow system can be very time consuming since it has to be performed manually for each workflow application and execution engine. The implementation is based on proprietary technologies, which restricts the monitoring to inter-organizational workflow applications. Similar to the agent-based framework, our approach incorporates M&A concerns into the code of workflow applications to analyze them at runtime. Furthermore, we provide a solution to automate the implementation of monitoring and analysis specifications into

an existing workflow implementation. The M&A at runtime can support a business intelligence process to monitor, measure, evaluate, and notify special situations during the execution of workflow applications in a timely fashion.

Monitoring by Re-Engineering Workflow Applications

BizAgi [Biz] applies a re-engineering process to the applications and their data models to convert them into a workflow application. A workflow application is defined around the data model, and XPath is used to access the data that is shared by all workflow elements. The workflow variables are defined at modeling time and they correspond to a field in the database. BizAgi offers a rich visual framework to define business rules and execution process rules. They also have a set of templates to send emails, and to create forms. In contrast to BizAgi, our approach provides independence of the process models, a separated data model for monitoring purposes, independence of the workflow engine, and explicit analysis functions modularized of the workflow application.

In contrast to these approaches, we propose a domain-specific language to specify M&A concerns independently of specific implementation languages.

Separation of Monitoring Concerns

The idea of separating crosscutting monitoring code is not new and is present in some approaches [VCJ04] [Sch07]. However, they do not provide the mechanisms to specify and access the information inside the workflow activities. These approaches investigate how concepts from monitoring activities relate to AOP concepts. However, they do not provide adequate support for all the required monitoring concepts since their specifications of monitoring activities are specified using AOP languages instead of monitoring languages.

10.2.2 Model-driven Approaches

The following subsections present different workflow M&A approaches which use a model-driven strategy.

High-level Business Process Measurement

The idea of high-level abstraction in business process measurement is presented by Valdis in [Vit04]. This approach presents a methodology for defining business process measures based on meta-modeling. The measures are related to business concepts using Unified Modeling Language (UML) activity diagrams with extensions to associate object measures. However, we observe that this approach only considers definitions of typical process measures in the initial model. In contrast, Our DSL provides abstractions to specify new measurements based on existing measurement and workflow data. In our approach

we specify M&A concerns in terms of BPMN process models, which provide a standardized formalism to model workflow applications.

Model-driven Development of Monitored Processes

Momm et al. [MMA07] present a model-driven methodology for a top-down development of monitored processes based on a service-oriented architecture. The authors acknowledge the problem of a time consuming effort to manually implement the instrumentation for triggering business events. This is due to the fact that IT systems for workflow applications are very specific to each company, thus the monitoring implementation is specific to the employed IT system and execution engine. The authors consider *Service-oriented Architecture* (SOA)) platforms such as BPEL and web services, or other SOA platforms (*e.g.,* CORBA and a workflow engine). This work presents how to systematically develop a BPEL instrumentation in a top-down approach to automate the generation of monitored workflow applications. The authors introduce metamodels a) to create models for the monitoring of workflow applications, and b) to create models to define process performance indicators (PPI) (based on a UML profile) into specific monitoring points. The models created with these metamodels are transformed into process models, which contain the required monitoring activities.

The PPI model retrieves monitoring information (*e.g.,* state changes) for a specified monitored object (*i.e.,* process instance, flow object instance) and determines a PPI value by means of predefined monitoring metrics. The monitoring information is retrieved by using the events fired by the process execution engine (EventProbes). To support these EventProbes, Momm et al.present certain extensions to instrument a BPMN-based orchestration model (closer to BPEL concepts) with monitoring messages and monitoring tasks. Monitoring messages hold information about the current state, whereas monitoring tasks send this information to the associated monitoring infrastructure. However, the monitoring tasks have to be placed appropriately in the process model.

In contrast, our DSL facilitates the specification of a set of monitoring events indicating where to capture information and what to do with this information. We discuss the problem of crosscutting M&A concerns in workflow applications and present an approach to keep these concerns modularized and to compose them automatically with the workflow implementation. Thus, the required instrumentation can be added in multiple places of the process model and in the workflow implementation by using AOP to retrieve state information. We use model transformations to control the generated artifacts. We keep trace of the mappings done between the process model and workflow implementation to automate the MonitA generation process. We also make workflow variables explicit in process models, which enables to specify and gather application-specific information. The PPI model can be compared with

our model for specifying measurement variables. In addition to this measurement variables specification, we provide a model, as part of our DSL, to specify how to compute and evaluate this information.

In the work of Momm et al. [MMA07], the composition of workflow activities and monitoring activities is performed at the domain level, rather than at the implementation level. Since the process is not orthogonal to the M&A concerns, it has to be evolved at the same time than monitoring requirements. In contrast, we delay the composition between both domains at the implementation level. Section 7.5.1 details the advantages and disadvantages for composition depending the level of abstraction.

The approach presented by Momm et al. [MMA07] supports the monitoring and measurement in a platform-independent way. In addition, we provide a concrete syntax and notations to specify the M&A concerns. This approach is supported into a SOA platform based on BPEL and web services. In addition to this SOA platform we also target platforms that are not based on SOA such as the jBPM platform based on JPDL and Java.

10.3 Workflow Monitoring and Analysis a Posteriori

The monitoring and analysis a posteriori is typically based on the event logs created in the execution of workflow applications. Multiple techniques can be identified in this approach such as data mining, process mining, and semantics business process mining.

10.3.1 Architectures for Workflow Applications

Solution Manager Service

The authors in [MSzM06] present the Solution Manager Service (SMS), which is a web service-based system for on-demand workflow management. This work states that monitoring and analyzing information about the performance of workflow applications, especially the one crossing the organizational boundaries, is complex due to the lack of a common infrastructure to capture, transform, and accumulate audit trails from distributed workflow applications. The SMS infrastructure condenses the information gathered in a centralized repository. All the interactions with this system are via a set of web services: define web service, log web service, and monitor web service.

The *"define web service"* provides services with a way to describe a web service of the process with performance measurements information. They use BPEL to define workflow applications. This corresponds to a low-level specification since it involves a language that is not targeted at monitoring and

analysis, and since the measures get combined with the process information. In addition, to publish the web service, the workflow developer has to provide the service description containing details of its data types, operations, bindings, and network location. This web service enables the specification of general performance measures for both the web service and for a partner group that participates in the web service.

The *"log web service"* provides a mechanism to gather all state changes of a workflow application, which are recorded in an audit trail. The state of a workflow application is constituted by the messages that are exchanged, which are held in BPEL variables. When a message is received, the variables are populated and the subsequent requests can access data. An auditing mechanism must be implemented and incorporated in the workflow application, or by using probes in the engine to track the state changes. When invoking an external service within the workflow definition to log audit information, a log request can be executed in any stage of the BPEL application. However, the overhead of invoking the external service for each state change can be a potential bottleneck. When considering the interception of web service requests to extract the data needed for auditing, only an incomplete audit trail data can be extracted. This is due to the fact that the BPEL web service requests have limited visibility to internal workflow information. Information such as workflow instances or variables used by the workflow engine to control the workflow execution are not included in the web service requests. This typically results in complex and expensive audit trail processing for gathering the complete information. When tracking BPEL flows in a WFMS, it is possible to access the complete runtime data and contextual information about the workflow application. Typically, WFMS provide listener interfaces that get invoked when particular flow-related events (*e.g.,* when a process starts, when an activity finishes) take place. When workflow applications are not managed by WFMS, the option to track workflow state changes is to add sensors (probes) in the source systems (*e.g.,* database layer, business logic layer, presentation layer) to capture information and generate audit data.

The *"monitor web service"* enables the definition or modification of sensors. These sensors observe periodically the performance data from the data warehouse. The monitor web service can be used to define sensors for a web service and for partners. This system enables management for inter-organizational workflow applications and the ability to share the monitoring infrastructure among business partners. This infrastructure also provides an approach for processing events in near real-time trying to make the performance information available earlier than ETL (extraction, transformation, and load) approaches. It integrates and transforms events buried in the audit trails of workflow applications at runtime. One advantage of this approach is the outsourcing of data processing to external parties to handle data intensive operations.

In contrast, we propose a mechanism to intercept events related to state

changes in workflow variables to capture execution context information and to invoke an analysis action. In our approach, flow-related and data-related events are captured by means of aspects technology. The performance of the M&A code can be improved since the aspects are generated.

In our approach, the monitoring and measurement information is captured, processed, and analyzed at runtime for improving the decision-making process. This analyzable information can quickly increase to a sizable amount and could impact the operational performance of the workflow execution [LR99]. However, we consider our approach suitable to evaluate critical performance measurements during the workflow execution. We consider that the *Solution Manager Service* system can be used to complement ours by outsourcing the analysis of the measurement information that is non-crucial at runtime to external parties.

We present an approach of general applicability that can be used in multiple workflow platforms. Both approaches target similar goals, however, the specifics of each one are different such as the abstractions for the specification of M&A concerns, our generative approach, and the technique to gather information. Similar to the SMS approach, we have a centralized measurement storage system to store and collect measurement information.

10.3.2 Business Process Intelligence

Certain approaches use data mining and data warehousing techniques to recover the workflow execution information and to discover structural patterns for identifying the characteristics that contribute to determine the value of a metric [GCC+04] [CCDS04]. These approaches present a set of integrated tools for managing workflow execution quality, which are referred to as Business Process Intelligence. Casati et al. [CCD+02] at HP laboratories use process mining algorithms to derive patterns from workflow audit trail data. Time-based workflow attributes (*e.g.,* activities processing time) can be captured and illustrated by using the Business Process Cockpit [SCDS02] front-end.

In our approach, there is an explicit definition of the relations between existing metrics and workflow data to built new application-specific metrics. This information is captured and analyzed during the execution of a workflow application. We could benefit from these approaches since they can provide values for new measures that were not specified and implemented as part of the workflow application.

10.3.3 Semantic Business Process Management

Other works [GCC+04] [HLD+05] introduce the idea of using semantics to perform business process management (BPM). Grigori et al. [GCC+04] present an architecture for the analysis, prediction, monitoring, control and optimization

of workflow execution using taxonomies to capture semantic aspects. Hepp et al. [HLD+05] propose an approach for building Semantic BPM (SBPM) systems through the combination of BPM, semantic web, and web services.

The following subsections present different approaches that use ontologies for workflow M&A a posteriori.

Ontologies for Business Process Management

Other approaches [PDB+08] [dMPvdA+07] [PD07] use ontologies to capture the semantic aspects and process mining techniques for retrieving information. Ontologies capture semantics information, which allows reasoning about workflow applications and its execution. Process Mining aims at automatically discovering analysis information based on event logs [vdAW05]. These logs contain information about the workflow execution, in which it is assumed that the workflow instances are identified and the tasks are registered in the order they were performed. Most of the process mining techniques are freely available in the open source tool ProM [vDdMV+05]. The work by Medeiros et al. [dMPvdA+07] presents the opportunities and challenges for semantic process mining and monitoring. The work by Pedrinaci et al. [PD07] shows an ontology for process mining.

This work is aligned with ours because the authors also consider analysis capabilities at the conceptual level, the definition of application-specific metrics, and subsumption relations between measurement information. The main difference, apart of using ontologies, is that our approach is focused on process management through the M&A of workflow activities instead of using process mining techniques. Thus we provide the mechanisms to extract application-specific information according to a special behavior in the workflow execution.

In addition, we introduce the idea of a DSL dedicated to workflow M&A, using a generative approach to automate the implementation of M&A concerns into particular workflow languages and engines. We provide a practical way to use the semantic knowledge for M&A. We complement the high-level process models with the description of process data and the interactions with workflow entities required to perform a application-specific analysis (data-centric analysis). Our approach facilitates to perform a similar performance analysis (measures) to the ones using a mining approach. Our DSL can be used to support process mining approaches by specializing the control action *trace* to provide the facilities necessary for creating event logs with structured analysis information. Moreover, our DSL allows defining explicit relations between monitoring data and workflow data to build new measurements

Strategy-driven Business Process Analysis

Pedrinaci et al. [PMHD09] present an approach for supporting strategy-driven business process analysis by using an ontology to capture strategic concerns. This approach is similar to ours in the sense that M&A concerns (or strategic concerns in terms of this strategy-driven approach) are declared at a conceptual level of abstraction. In our approach, we complement this conceptual M&A specification by defining explicitly its execution semantics with a set of model transformations to generate code into a specific workflow platform.

At the conceptual level, a close relation can be established between the concepts defined in both approaches. They use a business motivation ontology to describe a strategy for business process analysis referring to ends, means, metrics and influences. These abstractions are comparable to those we use for workflow M&A considering the definition of high-level analysis goals (ends), M&A requirements (means), measurements specification (metrics), and an implicit explanation while defining the M&A concerns (influencer). We use our DSL to capture monitoring events, whereas the strategy-based approach uses an events ontology to capture monitoring logs. We use our DSL to declare and compute measurements, whereas the strategy-based approach uses an operational analysis ontology. In addition, we offer mechanisms to declare complex measurement data facilitating the specification of application-specific metrics and to declare control actions over these data. At the implementation level, the strategy-driven approach is based on process mining to recover the strategic information, whereas in our approach we recover, build, and analyze the measurement information at runtime by combining M&A concerns with workflow code. Our implementation approach for conceptual M&A specifications using our DSL could be used as a validation for the strategy driven approach.

10.3.4 Process Analysis based on Event logs

The following sub sections present different approaches that use event logs for workflow M&A a posteriori.

Business Provenance

The authors in [CDM+08] present business provenance as a technology for tracking and correlating relevant aspects of business end-to-end operations (workflow applications). In this way, a typical data and process re-engineering is not required since relevant aspects such as the data that is produced, the resources that are involved, and the tasks that were executed are discovered and correlated to provide a representation of the workflow application. These relevant aspects (tasks, resources, data, relations) are stored in a generic data model in order to utilize the operational data. Thus, provenance technologies

help understanding what actually happened during the life cycle of a workflow application. The visibility of all related aspects to a workflow application is required to manage compliance against a set of business rules.

This technology is also related with Business Activity Monitoring (BAM) since business provenance traces operations to extract relevant information. These traces are based on specific instrumentation through appropriate configuration of recorders (probes), which are adapter components that are able to listen to events of the underlying information systems. The construction of these adapter components can be time demanding and costly. Business provenance is similar to process mining [RvdA08] since both allow process analysis based on event logs.

In contrast, we generate the code to capture M&A information by using the adopted workflow infrastructure (*e.g.,* workflow language constructs) and underlying implementation language (*e.g.,* aspect language, programming language). Whereas business provenance supports monitoring by using rules that enrich the workflow application (provenance graph), we offer an approach for the explicit specification of M&A concerns modularized with respect to the workflow implementation. Business provenance, as we do, treats data as first class entities since they outrun the life cycle of the workflow application. That is why we focus on supporting events and relations in terms of data interactions that are important for a particular specification. This business provenance technology could complement our approach by adding a historical perspective for enabling root cause analysis by providing time series navigational views (*e.g.,* events, user's actions) over the execution of workflows. In this way, it would be possible to realize, when a particular behavior occurs, the resource responsible for it and why this took place. In the business provenance approach, the interaction with the provenance data could be improved using our domain-specific language to create, browse, and query this information.

In our approach, the monitoring of workflow relevant data can be specified and captured selectively and explicitly in order to reduce the complexity of analysis activities at processing large volumes of workflow data.

Conformance Checking

The authors in [RvdA08] present an approach to check the conformance of a process model and an event log. Conformance checking is measured by two different types of metrics: fitness and appropriateness. *Fitness metrics* measure the extent to which the log traces can be associated with valid execution paths specified by a process model. In addition, *appropriateness metrics* measure the degree of accuracy in which the process model describes the observed behavior, combined with the degree of clarity in which it is represented. In contrast, the focus of our approach is on monitoring and analyzing performance and efficiency measurements rather than monitoring (un)desirable behavior on the

workflow application.

10.3.5 Tool Support

There are multiple architectures and prototypes created for monitoring and analyzing workflow applications. Casati et al. [CCF+00] introduce a language named Chimera-Ex to model and monitor exceptions in workflow management systems. The exceptions that can be monitored are specified at workflow build-time and are integrated with a specific workflow management system.

The authors in [zMR00] present the PISA (Process Information System based on Access) tool, which extracts performance metrics by analyzing workflow event logs. This work presents monitoring facilities for three different analysis views: processes and functions, resources, and process objects (data). The available information defines the quality and scope of the analysis that can be performed on a workflow application. To support the analysis in terms of an object-view in the PISA tool, the workflow model was modified by adding an activity for creating a log-entry into the audit trail with the relation between a workflow instance ID and the process object ID.

In contrast, we keep explicit and external the M&A specifications, which can be combined with data from multiple information sources. We enhance the workflow implementation by adding automatically M&A concerns required to evaluate the workflow applications at runtime. We propose a uniform workflow M&A approach to be applied in diverse workflow management systems. In addition to the data recorded by the workflow management system (state-changes in processes and activities), we acknowledge the need to capture state-changes regarding workflow variables and to construct application-specific measurements. This information must be stored to support specialized analysis.

We enhance the analysis of workflow applications since we evaluate quality criteria on the data associated with multiple workflow elements. We provide an explicit specification of workflow and measurement variables to analyze a workflow application. In contrast to the approach of the PISA tool, we perform the M&A at runtime. In our approach, the information required for analysis is always available since M&A concerns are integrated with the workflow infrastructure. If we need to add a concern to the workflow, it is not added directly to the process model, but it is specified externally and incorporated automatically to the workflow application. This allows traceability and maintainability facilities when adding or changing M&A concerns.

10.4 Dynamic and Static Program Analysis

The behavioral information in a program can be obtained through program analyses techniques such as dynamic analysis and static analysis. In dynamic

analysis the behavioral information can be obtained by monitoring the programs behavior at runtime, whereas in static analysis the behavioral information can be predicted by analyzing the program's code at compile-time.

The focus of this dissertation is on dynamic analysis of measurement information in workflow applications. We present a general overview of different dynamic analysis approaches in general programs since they are related with our research work. Each of these approaches follow different goals such as program verification through assertion checking [SB06], debugging through event-related breakpoints [GDJ02] [Duc99], program testing and profiling [GOA05] [DFW04], program understanding [DGD05] [Ric02], and program events monitoring [KF04].

We instrument source code with customized monitoring events by considering the development of M&A concerns as part of the life cycle of workflow applications and not a debugging environment. We do not consider program verification and understanding such as checking safety properties, design recovery, or visualization of a program's behavior. Nevertheless we provide an static analysis in terms of the data types of MonitA specifications along with the workflow implementation. In contrast to the object-oriented and procedural language paradigms used by the above approaches, our approach is used in workflow languages to analyze workflow applications.

In dynamic analysis the means for events selection are typically performed by stepwise program execution and by instrumentation. A debugger performs a stepwise program execution to skip events that are not important to the program analysis. The events can be selected by instrumenting an entire program [GOA05] or by a selective instrumentation by using enumeration of individual methods [DFW04], regular expressions on bytecode operations [KF04], an AspectJ-like pointcut language [SB06], annotations, and runtime events related to programming language constructs. Our approach supports a selective instrumentation of flow and data related events by using an aspect-oriented pointcut language and by using workflow language constructs to create runtime events. Our approach instruments automatically the workflow application source code in order to monitor only the requirements of workflow analysts.

The approaches for dynamic analysis offer a different language support for reasoning over the monitored events. For example, the above approaches support the expression of a) assertions in temporal logic over AspectJ joinpoints, b) user-defined monitors for passively processing events, c) behavioral tests in a logic programming language, d) partial matches against program traces in a fuzzy logic programming language, e) visualizations of query results in Prolog, and f) behavioral debugging or profiling queries in Prolog or in a SQL-like query language. Our approach offers a DSL to specify user-defined measurement and control actions for reasoning over the monitored events. The DSL includes quantifiers and aggregate operations over workflow events to incorporate M&A concerns over the workflow application.

10.5 Process Data Models

Although, BPMN provides basic support for an informational perspective, these process models suffer from the lack of support to specify the data used by their activities [Dub04]. This drawback is of special interest in our approach to support M&A specifications in terms of workflow relevant data.

10.5.1 Data Modeling in Workflow Applications

Several works have acknowledged the importance of a data-centric perspective in business process modeling [dlVFS+09] [NC03] [RLVDA03] [SZNS06].

Data Models in Business Process Diagrams

The authors in [dlVFS+09] present a requirements engineering approach to complement business process diagrams with a data perspective for modeling a workflow application. Functional requirements are defined by the specification of the task descriptions of a workflow application by customized BPMN process diagrams. These diagrams indicate the behavioral perspective (control flow) of the system. Data models indicate the storage perspective of a system. These data models are derived from functional requirements by customizing an existing approach named *info cases*, which model use cases and domain data models in an integrated model. These use cases are used to describe the information flows between an information system and its actors (*i.e.,* domain entities that are used in the task descriptions). These information flows capture a specification of the composition of flows, and a dictionary of elementary items of information (domain class diagram).

In contrast, we use standardized specifications widely accepted and supported by interoperable tooling to specify process models (*i.e.,* BPMN) and data models (*i.e.,* XSD). We use an XML schema (XSD), conceived as a data type model, to represent explicitly the structure of the workflow data hidden in the internals of the workflow application. We complement this data types specification with a workflow variables model, which captures and describes the information used by the workflow activities. We use model transformations to integrate process models and data models and to provide an automatic generation process for deriving M&A executable code.

Business Artifacts

Nigam et al. [NC03] introduce the notion of business artifact to manage data. They use graphical elements to specify artifacts, which are business entities (*e.g.,* insurance claim, order, financial deal) guiding the operation of the business. A business artifact includes specifications to hold relevant data about the artifact as it moves through the workflow, and the possible life-cycles it

might follow. Each artifact has a unique identifier. This approach could be used to support our target of intercepting state changes over the data since it models the possible life-cycles of business artifacts. A business artifact can be compared with data entities in our approach.

Data-Flow Perspective for Business Process Management

Sun et al. [SZNS06] address the issue of detecting data flow anomalies such as missing, redundant, and conflicting data. This approach includes a framework that enables the specification of data flows and their analysis. Data flow specifications associate the operations done by an activity over a datum and represent the input and output data of an activity. This input/output representation is supported by an extension of UML activity diagrams. The specification of data flow analysis provides a notation to establish a dependency between activities, and also provides an algorithm to implement a data flow verification artifact in workflow managements systems. This algorithm detects anomalies such as usage of data that have not been created, conflicts when multiple activities create different instances of the same datum and they known which one to use, and the creation of unnecessary data.

This approach can complement ours by adopting an algorithm to detect anomalies in terms of data (*e.g.,* creation, duplication). Sun et al. [SZNS06] acknowledge that it becomes illegible when combining control and data flow elements in a single model. We modularize the specification of workflow variables with respect to the process model to improve understandability.

Framework for Document-Driven Workflow Systems

The framework for document-based workflow systems presented by Wang et al. [WK05] focusses mainly on passing documents between activities. Typically, these workflow systems provide a model of event adapters to capture the data flow based on database triggers. However, they do not concentrate on the processing of a document inside an activity.

In contrast to this approach, we model explicitly the data entities that shall be part of the workflow application and the possible interactions with the flow entities rather than data flow through tasks. This data entities specification complements the BPMN process models providing a mechanism to describe enriched M&A concerns.

10.5.2 Data Modeling on Other Domains

Foster et al. [FPWM08] discuss the problem about how to define interactions among web services to support operations on state. This issue discusses how to model and implement service state and the associated interactions in that

state. The state is related to the data values associated with a service that persists across interactions. State interactions are related to name state, access that state, modify that state, and destroy it. This work presents four different approaches (*i.e.,* Web Services Resource framework (WS-RF), WS-Transfer, HTTP, and no-conventions) for modeling state in web service interactions. Modeling state refers to modeling a projection of the underlying system state that is exposed externally.

Although this work is focussed in a web services domain, we tackle a similar problem since we acknowledge the necessity to explicitly define and implement data variables (workflow and measurement) among the flow entities of a workflow specification. In addition, we discuss the issue about how to intercept CRUD (create, read, update, delete) interactions (state changes) on the data entities. This is useful to support the monitoring related to the data values associated with an activity. These interactions are used to exchange data representations between the workflow application and monitoring concerns.

10.6 Domain-specific Aspect Languages

Business Rules on Business Processes

The authors in [PCl+07] present a Rule-based AOP framework for separating and executing business rules contained in business processes. This approach presents a way to integrate a business rule engine (BRE) with BPEL processes and a strategy to adapt changes of business rules dynamically at runtime. The business rules are separated using the BRE instead of an existing AOP programming language. However, one limitation is that if the BRE does not support an event condition action (ECA) mechanism, then an event-based (forward chaining) mechanism has to be implemented. Generic monitoring and control specifications can be associated with business rules in this approach. In our current implementation approach, the method to accept changes is to stop the currently running workflow application, modify the M&A concerns or create new ones, and re-generate the analysis implementation that is automatically composed with the workflow implementation. Although, our generative implementation process offers a greater agility to evolve the monitoring and analysis concerns, this adaptation requires to redeploy the workflow application into a workflow execution engine. This work could be used as a basis to create a mechanism to offer dynamic workflow changes for supporting the specification and implementation of unanticipated M&A concerns.

10.7 Service-Oriented Computing

Service-Oriented Computing (SOC) [Pap03] [HS05] is a paradigm for distributed computing. SOC is composed of different elements such as services, service compositions, a service inventory, a service-oriented architecture (SOA), and a service-oriented solution logic [PTDL07]. These elements are inter-related with the goal of creating and assembling services for developing software applications that span organizations and heterogeneous computing environments. This inter organizational network of services in the applications can be materialized by using the service-oriented architecture. SOA defines a way of designing a software system to provide services.

A service is the fundamental unit in SOC and is a platform-independent, autonomous, and self-describing software program. Services can be described, published, discovered, invoked, and composed using standard languages and protocols. A service can reply simple requests but also can execute complex automated business process that deliver and consume multiple software services. Web services are the most known technology to implement a SOC platform.

Web Services Specification

A web service uses XML-based standard specifications such as a) the Simple Object Access Protocol (SOAP) for transmitting data, b) the Web Services Description Language (WSDL) for defining services, and c) the Universal Description, Discovery and Integration (UDDI) specification for publishing and discovering services [WCL+05].

A SOAP message consists of message headers containing non application-dependent information (*e.g.,* quality of service information), and a message body containing application data. A WSDL document defines port types, which specify the messages that a web service sends or receives and its operations (abstract description). A WSDL also defines the mapping of a port type to a transport protocol using a data encoding format and a physical address to access the web service (concrete description). A UDDI registry contains information about organizations that provide web services, and the technical and legal meta-data about those services (*e.g.,* binding templates, technical models).

There are important web services issues such as service composition, security, reliable messaging, and transactions that are not addressed by the basic web service specifications. Thus, additional WS-* specifications have been proposed to provide mechanisms quality of service (*e.g.,* WS-Security, WS-ReliableMessaging) and for service composition (*e.g.,* BPEL).

The following subsection presents the specification of web services composition, which is relevant to our research work.

Web Services Composition

Multiple web services can be combined to solve a complex problem, to achieve a policy goal, or to provide a new service. The goals of web services composition are to increase reusability and to reduce complexity to integrate applications in intra and inter enterprise settings.

The web services composition involves the specification by means of a composition language and the execution by means of a runtime environment. The specification of a web services composition defines the control and data flow between the participants (*i.e.,* web services and their clients) involved in the composition. A web services composition can be specified by using different approaches such as activity diagrams, state charts, petri nets, process algebra, programming languages, and workflow languages. Nevertheless, workflow languages oriented to web services composition such as WSFL [Ley01] and BPEL [IBM02] provide high-level constructs to focus the specifications in terms of the business process logic. The BPEL workflow language was presented in section 2.1.3. These specialized workflow languages free the programmers from handling low-level concerns such as converting program data from and to XML, creating SOAP messages and setting their payload, handling faults, and assigning messages to different conversations. These languages define the web service composition using a workflow process specification (workflow schema).

Our approach was scoped to monitor and analyze the control and data flow of workflow applications that support not only web service participants, but also participants such as humans, automatic tasks, and external applications. In our approach, the monitoring and analysis is performed in terms of the data defined in the workflow process specification, and not in terms of the data contained in the individual participants.

10.8 Discussion: Positioning our Approach

This section evaluates our approach by considering the M&A characteristics presented in section 10.1. We split the evaluation of these characteristics by characterizing them according to their nature: execution environment capabilities and M&A capabilities. The following two tables summarize the characteristics provided by the analyzed related work. A "+" means that the characteristic is supported, a "-" means that the characteristic is not supported, and a "±" means that the characteristic is partially supported. We highlight the main features of the general approaches, without presenting particular variations in specific works.

Table 10.1 illustrates a summary of the execution environment capabilities provided by our approach and by the related work.

In this table we can observe that the approaches for workflow monitor-

1. Business Activity Monitoring
2. Model-driven
3. MonitA DSL
4. Service-oriented Architecture
5. Process Intelligence
6. Semantic BPM
7. Process Mining

Approach	1	2	3	4	5	6	7
Specification							
External M&A specification	-	+	+	-	+	+	+
Domain-specific abstractions	-	+	+	-	-	+	-
Conceptual Level	-	±	+	-	-	+	-
Event correlation	-	-	+	+	-	-	-
Custom measurements specification	-	+	+	+	+	+	+
Custom measurements processing	-	-	+	+	+	-	+
Control actions	-	-	+	+	+	+	+
Implementation							
Executable	+	+	+	+	-	-	-
Generative (analysis infrastructure)	-	+	+	-	-	-	-
Modularized M&A implementation	-	-	+	-	-	-	-
Automatic composition	-	-	+	-	-	-	-
Automatic event instrumentation	-	+	+	-	-	-	-
Unified measurement interface		+	+	+	+		+
Flexible measurement interface	-	-	+	+	+	+	+
External processing infrastructure	+	+	±	+	+	+	+
Standard based	-	-	+	+	-	+	±

Table 10.1: Execution Environment Capabilities Evaluated on Related Work.

ing and analysis a posteriori are suitable to manage measurements. We can observe that our approach supports these measurement capabilities but in a runtime monitoring context. A main feature in our approach is to support the modularized specification and automatic composition of monitoring and analysis concerns with workflow applications. This facilitates the maintainability of the M&A specifications.

Table 10.2 illustrates a summary of the M&A capabilities provided by our approach and by the related work.

We can observe that the approaches for M&A a posteriori allow the analysis of application-specific measurements in contrast to runtime approaches. One of the main contributions of our approach is to support this essential feature but in a runtime context. Another, main feature of our approach is the capability to intercept fine-grained workflow events to support the M&A of workflow data events. Nevertheless, MonitA requires to adopt mechanisms to support a robust analysis a posteriori, inter-organizational analysis, and the management of organizational measurements. This can be done by adopting mechanisms already provided by the other approaches. An opportunity to future work is

1. Business Activity Monitoring
2. Model-driven
3. MonitA DSL
4. Service-oriented Architecture
5. Process Intelligence
6. Semantic BPM
7. Process Mining

Approach	1	2	3	4	5	6	7
Monitoring							
Monitoring at runtime	+	+	+	±	-	-	-
Monitoring a posteriori	±	-	±	+	+	+	+
Fine-grained monitoring	-	-	+	-	-	-	-
Active monitoring	+	+	+	±	-	-	-
On-demand monitoring	+	+	+	+	+	+	+
Measurement							
Technical measurement	+	+	+	+	+	+	+
Organizational measurement	+	+	±	+	+	+	+
Application-specific measurement	-	-	+	+	+	±	±
Inter-organizational measurement	-	-	-	+	+	-	+
Control							
Operational measurement analysis	-	-	+	±	-	-	-
Informative analysis	+	-	+	+	+	+	+
Adaptive analysis	-	-	-		-	-	-

Table 10.2: Monitoring and Analysis Capabilities Evaluated on Related Work.

on supporting the specification of adaptive control actions on the workflow applications as response to the evaluation of M&A concerns at runtime.

10.9 Summary

This chapter has discussed the main features offered by different monitoring and analysis approaches and their main missing features. We have presented different approaches related to workflow monitoring and analysis at runtime, monitoring a posteriori, data management in process models, and domain-specific languages in other application domains. A comparison of our workflow monitoring and analysis approach against related work was presented based on a set of relevant characteristics for monitoring and analyzing workflow applications. The following chapter summarizes our research work, results and contributions presented in this dissertation.

Chapter 11

Conclusion

This chapter first summarizes and discusses the problems addressed, the results, and the strengths of our research work (see Section 11.1). Finally, we discuss on limitations and future work (see Section 11.2).

11.1 Conclusions

The goal of this research is on raising the level of abstraction for workflow developers for monitoring and analyzing workflow applications at runtime. To this end, we have defined and validated an approach for the specification and implementation of monitoring and analysis (M&A) concerns in workflow applications.

The main contributions of this research are: (1) an architecture for workflow monitoring and analysis, which offers the possibility to specify M&A concerns independently of specific workflow platforms and to target these specifications into different workflow platforms; (2) a domain-specific language to specify M&A concerns in terms of the workflow relevant data and application-specific measurements; (3) a generative implementation strategy to create the infrastructure to automate the implementation of M&A concerns on a specific workflow platform.; and (4) a MonitA execution platform that integrates automatically the M&A specifications with the implementation of workflow applications for two different workflow platforms.

Our approach is based on five pillars: maintainability, reusability, expressiveness, productivity, and understandability.

Understandability

The first pillar of our approach is on improving *understandability* of monitoring and analysis specifications. Current solutions require a low-level implementation of M&A concerns by using concepts from the workflow language,

the underlying implementation language, and the workflow engine.

To tackle this limitation we created a domain-specific language named MonitA that provides a set of abstractions for the workflow monitoring and analysis domain. We improve understandability of monitoring and analysis code by hiding technical complexity to workflow developers who are the intended audience. These abstractions extract essential properties and omit non-essential details required to move the specification of the M&A concerns to a higher level. Workflow monitoring and analysis abstractions focus on what the M&A concerns specify and not how they are implemented.

Maintainability

The second pillar of our approach is on the *maintainability* of the MonitA specifications. This pillar is motivated by the lack of support for the separation of M&A concerns observed in the current state-of-the-art on workflow applications. The need for separation of concerns is due to the fact that contemporary monitoring and analysis approaches do not evolve accordingly to the continuous evolution of workflow applications as a consequence of the business evolution. Existing approaches fail at separating M&A concerns from workflow applications, thus, their implementation results in crosscutting and entangled code that affects the maintainability of the workflow application and of the monitoring and analysis implementation.

In order to tackle this limitation, we modularize M&A concerns to be treated in isolation of the workflow application. In this way, we identify, manage and maintain the domain knowledge of M&A concerns, that is inherently changeable in workflow applications, according to the business needs and workflow application evolution. We consider the workflow application to be specified using BPMN and to be implemented using any workflow language. We also consider M&A concerns to be decoupled and implemented using the MonitA DSL and the underlying AOP technology.

Reusability

The third pillar of our approach is on the *reusability* of MonitA specifications across multiple workflow platforms. Currently, the specification of M&A concerns is different between workflow platforms. Contemporary solutions fall short because they do not treat monitoring and analysis as a predominant element in the workflow implementation. As a result, workflow developers need to build ad hoc infrastructures and abstractions to instrument the workflow implementation with M&A concerns.

To overcome this limitation, our DSL eases the specification of M&A concerns in terms of flow entities described in the BPMN process models. This

ensures that M&A concerns can be specified in a uniform and technology-independent way. Thus, workflow developers are able to write and reuse these concerns across different workflow platforms.

Expressiveness

The fourth pillar of our approach is on improving the *expressiveness* in the specification of M&A concerns. This requires to involve the workflow data to incorporate custom measurements specialized in the business domain the workflow is modeling. In traditional monitoring environments, the M&A concerns are specified only in terms of predefined measurements about the operational state of the workflow engine (*e.g.,* the time a workflow is running, the number of workflow instances) and not in terms of the workflow relevant data. This is because the internal workflow variables are encoded in the workflow implementation, thereby they are difficult to localize, use, and share with other workflow developers that require to specify M&A based on this data.

We tackle this limitation by providing mechanisms to model a projection of the internal workflow data and to model the workflow variables that are used by the flow entities and the operations that the flow entities perform on these variables. These data specifications are accessed and shared by multiple workflow developers during the specification of M&A concerns. MonitA customizes the instrumentation of workflow applications to intercept fine-grained workflow events in terms of data entities (*e.g.,* workflow variable changed). In this way, MonitA supports the specification of M&A concerns in terms of the domain the workflow application is modeling by referring to the workflow variables used by their activities. MonitA also allows workflow developers to specify and manage custom measurements that can be evaluated at runtime.

Productivity and Simplicity

The last pillar in our approach is on increasing the *productivity* and reducing the *complexity* to specify and implement M&A concerns. Despite the tools and techniques developed for workflow monitoring and analysis at runtime, workflow developers have to manually intervene in the workflow implementation to include code that implements custom workflow monitoring and analysis solutions. This is a complex task and requires a big effort from workflow developers since the instrumentation has to be performed manually for each workflow platform. This requires a thorough knowledge of the underlying implementation technology.

To overcome this limitation, the MonitA DSL offers improvements to write faster MonitA specifications. We also defined a strategy to allow MonitA specifications to be executable. This means that the MonitA specifications can be automatically composed with an existing workflow application to be executed

in a workflow engine. To this end, the MonitA platform automatically generates executable implementations from MonitA specifications. We consider that the traceability link between process model elements (BPMN) and workflow implementation elements is made explicit. In this way, the automatic generation of M&A concerns facilitates workflow developers to remain indifferent to the low-level workflow implementation details. The MonitA platform incorporates ideas from MDE to support the generative and traceability approaches, and from AOP as target of the M&A concerns transformation. The use of AOP allows us to keep the implementation of the M&A concerns with non-invasive changes to the existing workflow application.

11.2 Limitations and Future Work

This section describes a detailed analysis of a set of potential scenarios and limitations in which our approach has to be improved. Although these problematic scenarios are not covered in this dissertation, we acknowledge and describe them for introducing the kind of experiments that are to be developed in the near future.

11.2.1 Composing M&A Concerns at the Conceptual Level

The creation of a generative infrastructure using our generative strategy eases the implementation of M&A concerns to MonitA application developers. However, the implementation of a generative infrastructure is a complex and time-consuming task for a MonitA infrastructure developer. This requires a high expertise in the target implementation platforms (*i.e.,* workflow language, workflow engine, aspect language). Moreover, the mappings to create the translational semantics (model transformations) of the MonitA specifications have to be redefined for a new target implementation platform. Thus the semantics of executing MonitA specifications can vary between workflow platforms.

One of a our main research directions is to define an approach to compose the M&A concerns, which are modularized in the MonitA specifications, with the workflow application at the conceptual level. This involves a mechanism to represent the composition between M&A concerns, process models, workflow data entities, and measurement data entities.

We consider that our generative strategy can be improved by defining an implementation metamodel and a mapping that enable the composition of M&A concerns concerns at the conceptual level. This implementation metamodel must define the execution semantics for the MonitA specifications in terms of concepts associated to multiple domains such as: process modeling [OMG06a] [KV06], aspect-oriented modeling (AOM) [SSK+06], and data

modeling [FPWM08]. This implementation metamodel represents the relations between these domains to generalize the execution semantics of the MonitA language and avoids having knowledge of the specific target languages and platforms where the MonitA specifications are going to be implemented (*e.g.,* workflow language, aspects language, data management). Thus, an implementation model can be specialized to generate M&A code for specific workflow platforms.

Whereas the MonitA specifications describe a need for M&A of workflow applications, the M&A implementation model describes uniformly the way to realize the M&A concerns.

Our first approach towards the composition of M&A concerns at a conceptual level was the definition of a M&A concerns implementation model based on the pivot model presented by Correal et al. [Cor07]. The pivot model incorporates viewpoints in process models by using aspect-oriented modeling concepts, and facilitates the detection of interference problems between multiple viewpoints. We extend this pivot model by adding concepts for data modeling and the interception of elements associated with the workflow underlying implementation.

A model transformation must generate a *MonitA implementation model*. The concepts in the MonitA implementation model are closely related to the elements of the specific workflow platform. Thus, the mapping between them can be more easily established than going directly from the MonitA model to the particular workflow platform. The latter case is not desirable since it requires workflow infrastructure developers to redefine the execution semantics of MonitA models in terms of workflow code, aspect code, and data representation each time a new workflow platform is targeted.

The M&A concerns implementation model also motivates the necessity to build our own compiler and/or engine to validate the execution of M&A concerns. This can validate MonitA specifications related to a workflow application, independently of the workflow platform that will execute it.

11.2.2 Co-evolution of Process and MonitA Models

Even though our approach offers some evolution facilities, we have identified a number of subtleties and problems related to co-evolution and consistency when M&A concerns are modularized from the workflow specification.

We have identified different scenarios that need to be considered to cope with evolution when there is a delta in: a) process models, b) workflow implementation, c) workflow data, d) measurement data, e) MonitA specifications, f) MonitA grammar, g) target language, and h) workflow engine. These scenarios can be classified into intra-model and inter-model dimensions. Intra-model is related with the consistency rules defined within the same model, whereas inter-model involves multiple models.

Intra-model dimension. The evolution of the workflow specification

should result in the evolution of the workflow implementation. Thus, if the BPMN model is changed after the workflow implementation (*e.g.,* BPEL) was instrumented with the M&A concerns, then regenerating the BPEL code from the BPMN model is not going to contain the M&A concerns. Although with our workflow approach these concerns can be re-generated in case the workflow changes, the M&A concerns specification has to be coordinated with the newly defined high-level process model. However, if the workflow implementation is changed and the workflow specification is not updated, the analysis specification is not going to correspond with the workflow implementation. Thus, the process model represents a design point of view that is different from its representation in the workflow implementation. Consequently inconsistencies arise between the monitoring and analysis specification and its implementation.

A common solution for this problem is to unidirectionally synchronize the process model with its workflow implementation by using forward or reverse engineering. These approaches typically create new artifacts, replacing the old versions [SK04]. In these approaches, only one part is updated, however, if the changes are present in both artifacts (model and source code) then information might be lost. There are other approaches that consider round-trip engineering as a solution to keep source and target artifacts synchronized by taking both artifacts into account [AC06]. Nevertheless, a big effort has to be done to create tools supporting this approach.

We also have to consider the problem of co-evolving the MonitA specifications when the language grammar evolves. The changes in all existing MonitA specifications need to be identified to ensure their consistency. This requires versioning techniques to update models conforming to the changed metamodels for preserving models compliant and valid to the metamodel. To deal with this problem we have considered the approach presented by Cicchetti et al. [CREP08], which deals with the co-evolution problems in the models when the metamodels evolve. This approach is based on a model difference representation to express the metamodel changes in a difference model. The difference model contains the differences of two versions of the same metamodel. The co-evolution (co-adaptation) is done by a higher-order model transformation, which receives as input the difference model and generates another model transformation for producing the model's co-evolution.

Inter-model dimension. The evolution of the process model should result in the evolution of the MonitA specifications. However, the main problem is that the changes in the high-level process model can affect the corresponding MonitA specifications. Therefore, the generated MonitA code are not going to fit into the new generated workflow implementation. Consequently, it is necessary to define how the changes in the process model can be notified to its corresponding MonitA specifications and how to keep them updated.

Our initial idea to tackle these problems is to use the same approach that we propose to monitor and analyze workflow applications. This is a) creating

an intermediate model for tracing the changes done over the original workflow specification and MonitA specifications, and b) offering capabilities to notify the required control actions over these models. The traceability model would be used to detect and propose solutions to solve evolution problems between the workflow and MonitA models.

When multiple changes are expressed in a process model, workflow developers are not aware of potentially conflicting situations with the MonitA model. To detect inconsistencies of this type, we plan to use the traceability model, proposing actions to keep the consistency. This strategy takes advantage of the model transformations created to support the automated implementation of M&A concerns into a workflow implementation. During this transformation, we could analyze the dependency among the new process model, the MonitA model, and the original process model.

11.2.3 Managing Concerns Interactions

The specification of M&A concerns crosscut the specification of the workflow application. This can lead to interactions and interferences between analysis functions that have to be composed at the same point in the workflow application. This can affect the composition with the workflow application and the regular workflow execution. Thus, a mechanism to detect possible interactions and interferences is required to validate the resulting workflow application before it is executed.

We have found similar problems analyzing concern interactions using our language to those faced by the Aspect Oriented Modelling (AOM) community [SSK+07]. These problems are related with the fact that two analysis functions (aspects) could be applied to the same joinpoint, resulting in interference between them. In addition, there could be dependencies such as two analysis functions interacting with each other (circular referential dependency) or loops in the interactions between analysis functions.

We have identified a set of interaction scenarios that can generate conflicts and inconsistencies when workflow developers specify M&A concerns. Some of these interference problems can be associated with the constructs defined by Nagy, I., et al. [NBA05] such as: not ordering, precedence relation, and condition relation.

- *Validate the specification of analysis functions.* Two different analysis functions can be specified with the same name. In this case there can be an erroneous composition of an analysis function in the workflow implementation.

- *Two different application developers can define the same measurement variable with a different name (e.g., problemByArea, pba).* Thus, the same infor-

mation is stored in two measurement variables and none of them can contain a consolidated value to be used by other workflow developers.

- *The name of the workflow variables declared in the association model can be different from the actual variables in the workflow implementation.* In this scenario, the MonitA specifications defined in terms of the workflow variables cannot be composed with the actual implementation. Thereby, the measurements associated with the workflow data cannot be built accordingly to the monitoring necessities.

- *Not ordering.* This case occurs when two different analysis functions are defined for the same workflow event (*e.g.,* on finish [root.SubmitForm]). If the two different analysis functions do not depend on each other, then the ordering between them is not important.

- *Precedence relation.* This case occurs when the computation of a measurement variable within an analysis function depends on the execution of another analysis function. In this case, the dependent analysis function must be executed after the analysis function with the independent measurement data.

- *Condition relation.* In some cases there are problems that cannot be solved by ordering the analysis functions. This is when the execution of an analysis function depends on the outcome of another analysis function in the same joinpoint. In this case, the invoked function should be executed first.

An initial approach towards the detection of interferences is the creation of an intermediate implementation model for MonitA (see Section 11.2.1). This can be used to detect interferences before the MonitA specifications are translated and composed with the target workflow platform. This offers the following advantages:

* The interferences and interactions between analysis functions can be validated and resolved before the workflow is in execution. This allows to detect conflicts independently of the target workflow technology. This verification of interferences is performed once to ensure that the MonitA specifications are going to be instrumented correctly in every workflow execution technology.

* Workflow developers can analyze how the workflow application it is going to behave according to the MonitA specifications. Thus, it is possible to identify risks according to quality requirements related with the data management.

11.2.4 Expressiveness of the MonitA Language

There is a set of improvements that can be considered to enhance the expressiveness of MonitA without loosing its domain specificity.

Temporal Constructs

The MonitA language does not allow the specification of M&A concerns with time-dependent constraints. Nevertheless, during the MonitA specifications application developers require to capture the time dependencies between the different monitoring events and analysis functions. For example, temporal constructs such as *previous*, *most recent*, or *in the past* can be useful to capture behavioral patterns about the execution of M&A concerns in workflow applications.

Advanced Control Actions

We envision extending the MonitA language by offering support for more specialized control actions. The MonitA language could be extended to provide special constructs to express actions such as opening a file, running a script, or executing an application. Moreover, similar to the way that functions are supported in Excel, the MonitA language could support the specification of functions (*e.g.,* mathematical, statistical) to enhance the analysis that can be performed on the measurement information.

Monitoring Events

The current MonitA language allows specifying monitoring events in terms of flow entities (*e.g.,* on finish [activityName]). However, the MonitA language must consider the specification of monitoring events in terms of the transitions in the workflow application to avoid conflicts. For example, consider a monitoring event type (on finish) specified in terms of an activity with two output control flows (*i.e.,* split). With the current expressiveness support, the set of analysis functions associated with a monitoring event are invoked for each possible execution path. To facilitate the correct execution of M&A concerns, a pattern mechanism must be provided to refer to specific transitions or to a set of them. The same can arise when a monitoring event is specified in terms of an activity with two input control flows (*i.e.,* loop).

Measurements Management

We envision extending the MonitA language to support the specification of measurement variables that cross the boundaries of a single workflow application. The main research direction is to support the enterprise-wide and

cross-enterprise workflow analysis by using a common MonitA specification. Another improvement that must be supported by MonitA is the association of measure scale types (*e.g.,* nominal, ordinal, interval, ratio) during the specification and management of measurement variables.

11.2.5 Specification at a Higher-Level of Abstraction

The specification of M&A concerns with the MonitA language still requires programming skills. This is due mainly to the support for managing the measurement and in particular for the support of collections. A possible research direction is to define a higher abstraction syntax for the MonitA language.

11.2.6 Performance Evaluations

We consider that our approach introduces an overhead on the operational performance of the workflow execution since measurement data is captured, processed, and analyzed at runtime. We require to evaluate the impact on runtime performance on whether the analysis functions are interleaved with the workflow application or if they are performed after workflow execution. The execution performance of MonitA specifications matters but we can optimize how these concerns are implemented in our generation process. For scenarios with a high amount of measurement data, we consider to integrate our approach with approaches specialized in analysis on-demand to delegate the processing of non-crucial measurement data to external parties.

Appendix A

Formal Grammar of the MonitA Language

Non-Terminals

$$
\begin{array}{rcl}
< MonitaModel > & ::= & concern < ID > \\
 & & < Data > (< MeasureVariable >)^{*} \\
 & & (< AnalysisFunction > | < MonitoringEvent >)^{+} \\
< Data > & ::= & import[< Path >] < ID > [as < ID >] \\
 & & (\text{“,”} < ID > [as < ID >])^{*} \\
< Path > & ::= & < ID > (\text{“/”} < ID >)^{*} \text{“/”} \\
< MeasureVariable > & ::= & (persistent|transient) \\
 & & < ID >< DataType >< ID > \\
< TransientVariable > & ::= & (< DataType > | < ID >) < ID > \\
< IndicatorVariable > & ::= & < INDICATORTYPE >< ID > \\
< VariableReference > & ::= & < ID > | < ID > | < ID > \\
< MonitoringEvent > & ::= & on < WorkflowEventType > \text{“[”} < MonitoringSubject > \text{“]”} \\
 & & trigger < FunctionInvocation > \\
< WorkflowEventType > & ::= & < ID > [\text{“/”} < ID >] \\
< MonitoringSubject > & ::= & [< ID >< ID > \text{“|”}] \\
 & & (< FlowEntityRef > | < DataEntityRef >) \\
< FlowEntityRef > & ::= & root(< FlowEntityNavigation > | < FlowEntityPattern >) \\
< FlowEntityNavigation > & ::= & (< DOT >< ID >)^{*} \\
< FlowEntityPattern > & ::= & < DOT > (\text{“*”}|\text{“!”} < ID >) \\
 & & [< ID > (\text{“/”} < ID >)^{*} < VariableNavigation >] \\
< DataEntityRef > & ::= & < FlowEntityNavigation > \text{“:”} < VariableNavigation >
\end{array}
$$

$$
\begin{aligned}
< VariableNavigation > &::= < ID > (< DOT >< ID >)^{*} \\
< FunctionInvocation > &::= < ID >< LPAR > [< DataCollection >] < RPAR > \\
&\quad (and[< AnalysisFunction > | < ID >] \\
&\quad < LPAR > [< DataCollection >] < RPAR >)^{*} \\
< DataCollection > &::= < PropertyValue > (\text{“,”} < PropertyValue >)^{*} \\
< PropertyValue > &::= < ID >< ASSIGNMENT > \\
&\quad (< Invocation > | < DataEntityCreation >) \\
< DataEntityCreation > &::= < ID >< LPAR >< DataCollection >< RPAR > \\
< Invocation > &::= < MeasureVariableRef > | < DataEntityRef > \\
&\quad | < Literal > | < EngineInvocation > \\
&\quad | < DateTimeInvocation > | < NULL > \\
< MeasureVariableRef > &::= < VariableReference > \\
&\quad (\text{“}- > \text{”} < CollectionOperationCall > \\
&\quad | < DateTimeOperation > | < DOT >< ID > \\
&\quad | < DOT > (allInstances|current) < LPAR >< RPAR >)^{*} \\
< CollectionOperationCall > &::= (size|notEmpty|isEmpty|first) < LPAR >< RPAR > \\
&\quad | < SelectOperation > | < AddOperation > \\
< SelectOperation > &::= select < LPAR >< TransientVariable > \text{“|”} \\
&\quad < ConditionSet >< RPAR > \\
< AddOperation > &::= add < LPAR >< TransientVariable > \text{“|”} \\
&\quad < AssignmentFunction >< RPAR > \\
< Literal > &::= < INT > | < ID > | < STRING > \\
< EngineInvocation > &::= engine < DOT >< ID >< LPAR >< RPAR > \\
< DateTimeInvocation > &::= dateTimeType < DOT > (now|today) \\
&\quad < LPAR >< RPAR > [< DateTimeOperation >] \\
< DateTimeOperation > &::= < DOT > (year|month|day|hour|min|sec) \\
&\quad < LPAR > [< INT > (\text{“+”}|\text{“−”})] < RPAR > \\
< AnalysisFunction > &::= mmcfunction < ID > \\
&\quad < LPAR > [< PropertySet >] < RPAR > \\
&\quad (< MeasurementAndControl >)^{+} endfunction \\
< PropertySet > &::= < TransientVariable > (\text{“,”} < TransientVariable >)^{*} \\
< MeasurementAndControl > &::= (< Action > | < EvaluationRule >)\text{“;”} \\
< Action > &::= < ControlAction > | < MeasurementAction > \\
< ControlAction > &::= (notify|alert|trace) < LPAR > \\
&\quad (ID < ASSIGNMENT >< STRING >)^{+} < RPAR > \\
< MeasurementAction > &::= (< AssignmentFunction > | < MeasureVariableRef >) \\
< AssignmentFunction > &::= (< MeasureVariableRef > \\
&\quad | < TransientVariable > | < IndicatorVariable >) \\
&\quad < ASSIGNMENT >< DomainExpression >
\end{aligned}
$$

$$
\begin{array}{rcl}
< EvaluationRule > & ::= & if < ConditionSet > then(< Action >)^{+} \\
 & & [else(< Action >)^{+}]endif \\
< ConditionSet > & ::= & < AndCondition > (or < AndCondition >)^{*} \\
< AndCondition > & ::= & < EvalCondition > (and < EvalCondition >)^{*} \\
< EvalCondition > & ::= & < NotCondition > | < LogicExpression > \\
< NotCondition > & ::= & not < LPAR >< ConditionSet >< RPAR > \\
< LogicExpression > & ::= & < DomainExpression > \\
 & & (< OPERATOR >< DomainExpression >)^{*} \\
< DomainExpression > & ::= & < Invocation > (< OPER >< Invocation >)^{*} \\
< DataType > & ::= & (< ID > | < CollectionType > | Indicator) \\
< CollectionType > & ::= & Collection\text{“}<\text{”}(< ID > | Indicator)\text{“}>\text{”}
\end{array}
$$

Terminals

$$
\begin{array}{rcl}
< ID > & ::= & [\text{“}\text{”}](\text{“}\text{”}|\text{“}a\text{”} - \text{“}z\text{”}|\text{“}A\text{”} - \text{“}Z\text{”}) \\
 & & (\text{“}\text{”}|\text{“}a\text{”} - \text{“}z\text{”}|\text{“}A\text{”} - \text{“}Z\text{”}| < DIGIT >)^{*} \\
< INT > & ::= & [\text{“}-\text{”}] < DIGIT > \\
< STRING > & ::= & \text{“}\text{”}\text{”}(< ID > |\text{“}\text{”})^{*}\text{“}\text{”}\text{”} \\
< DIGIT > & ::= & (\text{“}0\text{”} - \text{“}9\text{”}) \\
< DOT > & ::= & \text{“}.\text{”} \\
< LPAR > & ::= & \text{“}(\text{”} \\
< RPAR > & ::= & \text{“})\text{”} \\
< ASSIGNMENT > & ::= & \text{“}=\text{”} \\
< OPERATOR > & ::= & \text{“}==\text{”}|\text{“}<\text{”}|\text{“}<=\text{”}|\text{“}>\text{”}|\text{“}>=\text{”} \\
< OPER > & ::= & \text{“}+\text{”}|\text{“}-\text{”}|\text{“}*\text{”}|\text{“}/\text{”} \\
< KEYWORD > & ::= & concern, import, persistent, transient, on, triggers, \\
 & & notify, alert, trace, if, then, else, endif, null, Indicator, \\
 & & mmcfunction, endfunction, Collection, engine, as, root \\
< NAVKEYWORD > & ::= & allInstances, current, size, notEmpty, \\
 & & isEmpty, first, select, add \\
< DATEKEYWORD > & ::= & dateTimeType, now, today, \\
 & & day, month, year, hour, min, sec
\end{array}
$$

Appendix B

Semantics of MonitA Constructs

This appendix describes the semantics of the most relevant constructs of MonitA. Section 4.1.3 points to attributes, the specific syntax, and more specific information about each specific element.

Figure B.1 illustrates different constructs of the language and those ones that are relevant to detail their semantics.

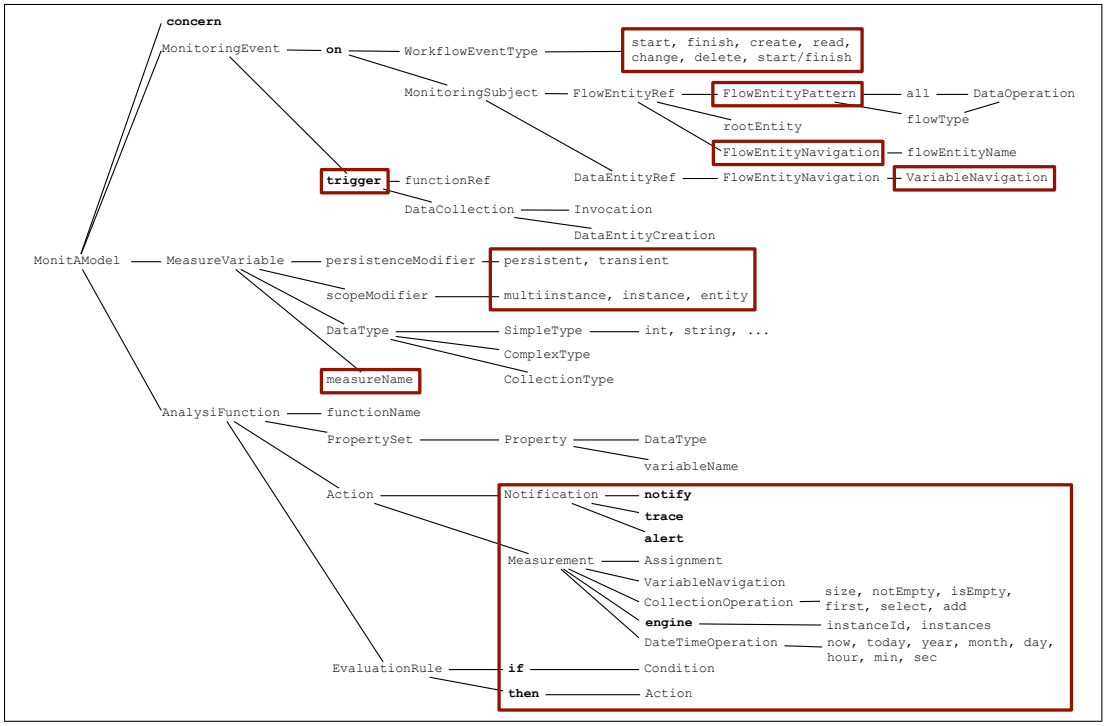


Figure B.1: MonitA Syntax Diagram.

Table B.1 describes the semantics of the selected MonitA constructs presented above.

Constructs	Semantics
MonitoringEvent	specifies what workflow event to observe, how to capture data, and the invocation of analysis functions
WorkflowEventType	
start	intercepts the start of the workflow root or a flow entity when it is reached by a transition
finish	intercepts the finalization of the workflow root or a flow entity when it triggers a transition
create	intercepts the creation of a workflow variable directly in the workflow root or through a flow entity
read	intercepts when a workflow variable is read directly in the workflow root or through a flow entity
change	intercepts when a value is assigned to a workflow variable in the workflow root or through a flow entity
delete	intercepts when a workflow variable is eliminated directly in the workflow root or through a flow entity
MonitoringSubject	
FlowEntityPattern	defines how to qualify a workflow event type with a workflow application context such as: the workflow root element, a specific flow entity, any flow entity in the workflow root, any flow entity of a specific type (Activity, Event, Gateway), any flow entity that operates a workflow variable, any flow entity type that operates a workflow variable, the workflow root when it operates on a workflow variable, a specific flow entity that operates a workflow variable, any flow entity that operates a workflow variable, any flow entity type that operates a workflow variable
FlowEntityNavigation	navigates through flow entities in the process model
VariableNavigation	navigates through data entities and their properties in the association data model
FunctionInvocation	
trigger	defines the instrumentation point (before, after, in) in a monitoring subject depending on the workflow event specification
MeasureVariable	define how to specify custom measures required to analyze the execution of a workflow application
persistenceModifier	
persistent	defines a measure that has to be made persistent in a external measurement data store system
transient	defines an intermediate measure that is used for the computation of another measure
scopeModifier	
multiinstance	defines that the value of a measurement variable is assigned to multiple or to all workflow instances of a workflow application
instance	defines that the value of a measurement variable is assigned to a workflow instance
entity	defines that the value of a measurement variable is assigned to a specific flow entity within a workflow instance
DataType	
SimpleType	corresponds to the built-in primitive and derived types provided by an XML schema

ComplexType	defines the data structure of new measurement information
Collection	defines a set of elements (flow, workflow data, measurement data) with the same data type
measureName	defines the persistence root used to access the values of the measurement information
AnalysisFuntion	defines how to instrument the workflow application
NotificationAction	
notify	sends an email to communicate relevant information
alert	generates an alarm to be visualized in a dashboard
trace	creates event logs with the analysis information
MeasurementAction	
Assignment	defines how to compute the value of a measurement variable
size	returns the number of elements in the collection
notEmpty	returns true if the collection is not empty, otherwise returns false
isEmpty	returns true if the collection is empty, otherwise returns false
select	returns a sub set of a Collection (measurement information)
first	returns the first element of a collection
allInstances	returns a collection with all measure value instances that are related with a measurement variable in persistence
engine	returns predefined measurements (instance identifier, number of instances) from the execution context provided by the workflow engine
now, today	assign a date and time value to a variable
DateTimeOperation	returns the values (year, month, day, hour, min, sec) of a variable with dateTime data type
EvaluationRule	
if	evaluates prescribed states established on measurement information
then	defines the measurement and notification actions to be executed based on the occurrence of a specific condition

Table B.1: Semantics of MonitA Constructs

Appendix C

Formal Grammar of the Data Association Language

Non-Terminals

$$
\begin{aligned}
< ProcessDataModel > &::= process < ID >< ModelRef > \\
&\quad (< VariableDeclaration >)^{+}(< AssociationData >)^{+} \\
< ModelRef > &::= import[< Path >] << ID >> [as << ID >>] \\
&\quad (\text{“,”} << ID >> [as << ID >>])^{*} \\
< Path > &::= << ID >> (\text{“/”} << ID >>)^{*}\text{“/”} \\
< VariableDeclaration > &::= < ScopeModifier >< DataType >< DataSet > \\
< ScopeModifier > &::= processScope|instanceScope \\
< DataSet > &::= < ID > (\text{“,”} < ID >)^{*} \\
< DataType > &::= < ID > |Collection\text{“} < \text{”} < ID > \text{“} > \text{”} \\
< AssociationData > &::= (root| < ID >)(< OperationOnData >)^{+} \\
< OperationOnData > &::= (writes|reads|removes|creates) < VariableReference > \\
< VariableReference > &::= \text{“(”} < ID > (\text{“,”} < ID >)^{*}\text{“)”}
\end{aligned}
$$

Terminals

$$
\begin{aligned}
< ID > &::= [\text{“”}](\text{“}_{\text{”}}|\text{“}a\text{”} - \text{“}z\text{”}|\text{“}A\text{”} - \text{“}Z\text{”}) \\
&\quad (\text{“}_{\text{”}}|\text{“}a\text{”} - \text{“}z\text{”}|\text{“}A\text{”} - \text{“}Z\text{”}| < DIGIT >)^{*} \\
< DIGIT > &::= (\text{“0”} - \text{“9”}) \\
< KEYWORD > &::= process, import, processScope, instanceScope, as, Collection, \\
&\quad writes, reads, removes, creates, root
\end{aligned}
$$

Appendix D

Model Transformations

This appendix contains a set of model transformations specified in the Xpand transformation language. This subset of transformations describe how to generate automatically AspectJ code from MonitA models. We chose to use these model transformations to transform MonitA models into AspectJ code when the workflow application is implemented in JPDL.

mmc2AspectJ.xpt

```
«IMPORT bpm_mmc»
«IMPORT datammc_bpm»
«IMPORT processdatammc_bpm»

«EXTENSION template::extensions::mmcUtil»
«EXTENSION template::extensions::namesUtil»

«DEFINE root(BPADataModel bpaDataModel, ProcessDataModel processDataModel) FOR MMCModel»
  «EXPAND aspect(this,bpaDataModel,processDataModel) FOREACH this.controlFlowEvents()-»
«ENDDEFINE»

«DEFINE aspect(MMCModel model,BPADataModel bpaDataModel, ProcessDataModel processDataModel) FOR
     LogicEvent-»
«FILE getAspecJFileName()-»
package «getAspectPackage()»;

import java.util.LinkedList;
import «getAspectJPackage()-».«this.eventName.toFirstUpper()-»;
import «getManagersPackage()-».«getMonitAManagerName()-»;
import «getManagersPackage()-».«getContextManagerName()-»;
import «getHandlersPackage()-».«getActionHandlerClassName()-»;
import «getObjectsPackage()-».*;

public aspect «getAspectClassName()» {
  «EXPAND Pointcut::pointcuts(model,bpaDataModel,processDataModel) FOR this-»
}
«ENDFILE»
«ENDDEFINE»
```

Pointcut.xpt

```
«IMPORT bpm_mmc»
«IMPORT datammc_bpm»
«IMPORT processdatammc_bpm»
«IMPORT ecore»

«EXTENSION template::extensions::mmcUtil»
«EXTENSION template::extensions::namesUtil»
```

```
«DEFINE pointcuts(MMCModel model,BPADataModel bpaDataModel, ProcessDataModel processDataModel) FOR
     LogicEvent-»
  «EXPAND pointcut(model,bpaDataModel,processDataModel) FOR this-»
«ENDDEFINE»

«DEFINE dataPointcuts(MMCModel model,BPADataModel bpaDataModel, ProcessDataModel processDataModel,
     EPackage dataSpecModel) FOR MMCModel-»
  «EXPAND dataPointcut(model,bpaDataModel,processDataModel, dataSpecModel) FOR this-»
«ENDDEFINE»

«DEFINE pointcut(MMCModel model,BPADataModel bpaDataModel, ProcessDataModel processDataModel) FOR
     LogicEvent-»
  pointcut «this.getPointCutName()-»(«getActionHandlerClassName()» handler): target(handler) && call
       («this.getProcessMethodBody()»);
  «EXPAND Advice::advice(model,this.getPointCutName(),bpaDataModel,processDataModel) FOR this»
«ENDDEFINE»

«DEFINE dataPointcut(MMCModel model,BPADataModel bpaDataModel, ProcessDataModel processDataModel,
     EPackage dataSpecModel) FOR MMCModel-»
  «LET searchDataProcessEvents(model) AS events»
  «FOREACH events AS event-»
  «LET event.getEventDataEntity() AS dataEntity-»
  pointcut «event.getDataPointCutName()-»(«dataEntity.toFirstUpper()» «dataEntity.toFirstLower()»)
       : target(«dataEntity.toFirstLower()») && execution(public * «event.getEventMethod()-»);
  «ENDLET-»
  «EXPAND Advice::dataEventAdvice(model,event.getDataPointCutName(),bpaDataModel,processDataModel,
       dataSpecModel) FOR event»
  «ENDFOREACH»
  «ENDLET-»
«ENDDEFINE»
```

Advice.xpt

```
«IMPORT bpm_mmc» «IMPORT datammc_bpm» «IMPORT ecore» «IMPORT processdatammc_bpm»

«EXTENSION template::extensions::mmcUtil»   «EXTENSION template::extensions::mmcDataUtill»
«EXTENSION template::extensions::namesUtil» «EXTENSION org::openarchitectureware::util::stdlib::io»

«DEFINE advice(MMCModel model,String pointcut,BPADataModel bpaDataModel, ProcessDataModel
     processDataModel) FOR LogicEvent-»
«LET model.searchMMCRules(this.eventName) AS rules-»
«FOREACH rules AS rule-»
  before («getActionHandlerClassName()-» handler) : «pointcut-»(handler){
    Object[] objects = thisJoinPoint.getArgs();
    ContextManager contextManager = (ContextManager)objects[0];
    String nodeId = (String)objects[1];
    «pointcut.toFirstUpper()» refObject = («pointcut.toFirstUpper()-»)objects[2];
    long instance = contextManager.getProcessInstanceId();
    long process = contextManager.getProcessId();
    String processName = contextManager.getProcessName();

    «FOREACH rule.actionSet.actionStatement AS act-»
      «EXPAND ConditionAction::extractParametersForNew(bpaDataModel) FOR act-»
    «ENDFOREACH-»

    «LET (List[String]) {} AS measuresNames-»
    «EXPAND ActionStatement::actionStatement(bpaDataModel,processDataModel,null,null) FOREACH rule.
         actionSet.actionStatement-»
    «FOREACH rule.actionSet.eAllContents.typeSelect(AssignmentFunction) AS actionSt-»
    «LET actionSt.eAllContents.typeSelect(Measure) AS measures-»
    «LET measuresNames.add(actionSt.mmcDataModelIdentifier.measure.measureName) AS temp2»«ENDLET-»
    «LET fillList(measuresNames,measures) AS temp1»«ENDLET-» «ENDLET-» «ENDFOREACH»
    «LET rule.actionSet.eAllContents.typeSelect(MMCDataFunctions) AS dataFunctions-»
    «LET dataFunctions.eAllContents.typeSelect(Metric) AS notValidMetrics-»
    «LET rule.getValidMetricsList(notValidMetrics) AS metrics-»
    «EXPAND ConditionAction::conditionAction(bpaDataModel,processDataModel,measuresNames,metrics,model)
         FOREACH rule.conditionActions-»
    «ENDLET-» «ENDLET-» «ENDLET-» «ENDLET-»
  }
«ENDFOREACH» «ENDLET-» «ENDDEFINE»

«DEFINE dataEventAdvice(MMCModel model,String pointcut,BPADataModel bpaDataModel, ProcessDataModel
     processDataModel, EPackage dataSpecModel) FOR ProcessEvent-»
«LET model.searchMMCRules(this.interactionEvent.eventName) AS rules-»
«FOREACH rules AS rule-»
  «LET getActionHandlerClassName2() AS actionHandler-»
  «LET this.getEventDataEntity() AS dataEntity-»
  «LET this.getEventAttribute1() AS attribute-»
  before («dataEntity.toFirstUpper()» «dataEntity.toFirstLower()») : «pointcut-»(«dataEntity.
       toFirstLower()»){
    «getAttributeDataTypeE(dataSpecModel,dataEntity,attribute)-» «dataEntity.toFirstLower()-
         »«attribute.toFirstUpper()-» = «dataEntity.toFirstLower()».get«attribute.toFirstUpper()
         »();
```

```
    «LET searchLogicEvent(model,this.interactionEvent.eventName) AS logicEvent-»
    «logicEvent.eventName.toFirstUpper()-» refObject = new «logicEvent.eventName.toFirstUpper()-»();
    «EXPAND eventObjectAttributes2(this) FOR logicEvent-»
    «ENDLET-»

    «getContextManagerName()» contextManager = «actionHandler-».getInstance().contextManager;
    String nodeId = «actionHandler-».getInstance().processNode;
    long instance = contextManager.getProcessInstanceId();
    long process = contextManager.getProcessId();

    «LET (List[String]) {} AS measuresNames-»
    «EXPAND ActionStatement::actionStatement(bpaDataModel,processDataModel,null,null) FOREACH rule.
        actionSet.actionStatement-»
    «FOREACH rule.actionSet.eAllContents.typeSelect(AssignmentFunction) AS actionSt-»
    «LET actionSt.eAllContents.typeSelect(Measure) AS measures-»
    «LET measuresNames.add(actionSt.mmcDataModelIdentifier.measure.measureName) AS temp2»«ENDLET-»
    «LET fillList(measuresNames,measures) AS temp1»«ENDLET-»
    «ENDLET-» «ENDFOREACH»

    «LET rule.actionSet.eAllContents.typeSelect(MMCDataFunctions).eAllContents.typeSelect(Metric) AS
        notValidMetrics-»
    «LET rule.getValidMetricsList(notValidMetrics) AS metrics-»
    «EXPAND ConditionAction::conditionAction(bpaDataModel,processDataModel,measuresNames,metrics,model)
        FOREACH rule.conditionActions-»
    «ENDLET-» «ENDLET-» «ENDLET-»
  }
  «ENDLET-» «ENDLET-» «ENDLET-»
«ENDFOREACH-» «ENDLET-» «ENDDEFINE»

«DEFINE eventObjectAttributes2(ProcessEvent processEvent) FOR LogicEvent-»
«FOREACH this.parameterSet.parameter AS param ITERATOR it-»
  «EXPAND seter2(processEvent,it.counter0) FOR param-»
«ENDFOREACH»
«ENDDEFINE»

«DEFINE seter2(ProcessEvent processEvent,Integer param) FOR Parameter-»
«IF this.type!=null-»
  «IF this.type.simpleType!=null-»
    refObject.set«this.parameterName.toFirstUpper()-»(«EXPAND template::aspectj::ConditionAction::
        invocation FOR processEvent.interactionEvent.parameterValueSet.parameterValue.get(param)-»);
  «ENDIF-»
«ENDIF-» «ENDDEFINE»
```

Bibliography

[AC06] Michal Antkiewicz and Krzysztof Czarnecki. Framework-specific modeling languages with round-trip engineering. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *MoDELS*, volume 4199 of *Lecture Notes in Computer Science*, pages 692–706. Springer, 2006. 11.2.2

[Act] ActiveBPEL Open Source BPEL Engine. http://www.activebpel.org/. 2.1.3

[AG00] IDS Scheer AG. Process performance manager. White paper, 2002-01-17 2000. 10.2.1

[Asp] AspectJ Team. The aspectj programming guide. http://www.eclipse.org/aspectj. 6.4.1

[BBH+94] Jeffrey M. Bell, Françoise Bellegarde, James Hook, Richard B. Kieburtz, Alex Kotov, Jeffrey Lewis, Laura McKinney, Dino Oliva, Tim Sheard, L. Tong, Lisa Walton, and Tong Zhou. Software design for reliability and reuse: a proof-of-concept demonstration. In *Proceedings of the conference on TRI-Ada, Baltimore, Maryland, USA*, pages 396–404. ACM, 1994. 3.1.1

[Biz] BizAgi. Bizagi bpm. http://www.bizagi.com/. 10.2.1

[BM04] Paul V. Biron and Ashok Malhotra. Xml schema part 2: Datatypes second edition. W3C Recommendation, October 2004. 4.1.1, 8.2.2

[BPM] BPMN-to-BPEL Eclipse plugin. http://code.google.com/p/bpmn2bpel/. 7.3, 8.3.1

[BVJ+06] Mathieu Braem, Kris Verlaenen, Niels Joncheere, Wim Vanderperren, Ragnhild Van Der Straeten, Eddy Truyen, Wouter Joosen, and Viviane Jonckers. Isolating process-level concerns using padus. In Schahram Dustdar, José Luiz Fiadeiro, and Amit P. Sheth, editors, *Business Process Management*, volume 4102 of *Lecture Notes in Computer Science*, pages 113–128. Springer, 2006. 6.4.2, 9.2.3

[CCD+02] Fabio Casati, Malú Castellanos, Umeshwar Dayal, Ming C. Hao, Mehmet Sayal, and Ming-Chien Shan. Business operation intelligence research at hp labs. *IEEE Data Eng. Bull.*, 25(4):32–35, 2002. 10.3.2

[CCDS03] Malú Castellanos, Fabio Casati, Umeshwar Dayal, and Ming-Chien Shan. Intelligent management of slas for composite web services. In Nadia Bianchi-Berthouze, editor, *DNIS*, volume 2822 of *Lecture Notes in Computer Science*, pages 158–171. Springer, 2003. 4.1.3

[CCDS04] Malú Castellanos, Fabio Casati, Umeshwar Dayal, and Ming-Chien Shan. A comprehensive and automated approach to intelligent business processes execution analysis. *Distributed and Parallel Databases*, 16(3):239–273, 2004. 10.3.2

[CCF+00] Fabio Casati, Silvana Castano, Maria Grazia Fugini, Isabelle Mirbel, and Barbara Pernici. Using patterns to design rules in workflows. *IEEE Trans. Software Eng.*, 26(8):760–785, 2000. 10.3.5

[CDM+08] Francisco Curbera, Yurdaer N. Doganata, Axel Martens, Nirmal Mukhi, and Aleksander Slominski. Business provenance - a technology to increase traceability of end-to-end operations. In Robert Meersman and Zahir Tari, editors, *OTM Conferences (1)*, volume 5331 of *Lecture Notes in Computer Science*, pages 100–119. Springer, 2008. 10.3.4

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *OOPSLA03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003. 6.3.3

[Cha07] Anis Charfi. *Aspect-Oriented Workflow Languages: AO4BPEL and Applications*. PhD thesis, 2007. Mira Mezini and Gustavo Alonso. 6.4.2

[CKO92] Bill Curtis, Marc I. Kellner, and Jim Over. Process modeling. *Commun. ACM*, 35(9):75–90, 1992. 2.1.1

[Cle10] Thomas Cleenewerck. Design principles for domain-specific languages. Technical Report vub-tr-soft-10-11, Software Languages Lab, Vrije Universiteit Brussel, 2010. 3.1.2, 5.1

[CLRC05] Charles Consel, Fabien Latry, Laurent Réveillère, and Pierre Cointe. A generative programming approach to developing dsl compilers. In Robert Glück and Michael R. Lowry, editors, *GPCE*, volume 3676 of *Lecture Notes in Computer Science*, pages 29–46. Springer, 2005. 3.1.1

[CM06] Anis Charfi and Mira Mezini. Aspect-oriented workflow languages. In Robert Meersman and Zahir Tari, editors, *OTM Conferences (1)*, volume 4275 of *Lecture Notes in Computer Science*, pages 183–200. Springer, 2006. 6.4.2

[CM07] Anis Charfi and Mira Mezini. Ao4bpel: An aspect-oriented extension to bpel. *World Wide Web*, 10(3):309–344, 2007. 6.4.2

[Cor07] Dario Correal. *Definition and execution of multiple viewpoints in workflow processes*. PhD thesis, 2007. 11.2.1

[CREP08] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. Automating co-evolution in model-driven engineering. In *Proceedings of the 12th International IEEE Enterprise Distributed Object Computing Conference (ECOC 2008), Munich, Germany*, pages 222–231. IEEE Computer Society, September 2008. 11.2.2

[Cro08] Douglas Crockford. The world's most popular programming language has fashion and luck to thank, March 2008. http://www.insideria.com/2008/03/the-worlds-most-misunderstood.html. 3.1.2

[Cza04] Krzysztof Czarnecki. Overview of generative software development. In Jean-Pierre Banâtre, Pascal Fradet, Jean-Louis Giavitto, and Olivier Michel, editors, *UPP*, volume 3566 of *Lecture Notes in Computer Science*, pages 326–341. Springer, 2004. 6.3.2

[dAFGD02] Ricardo de Almeida Falbo, Giancarlo Guizzardi, and Katia Cristina Duarte. An ontological approach to domain engineering. In *Proceedings of the 14th international conference on*

Software engineering and knowledge engineering (SEKE 2002), Ischia, Italy, pages 351–358. ACM, 2002. 3.1.1

[DFW04] Stéphane Ducasse, Michael Freidig, and Roel Wuyts. Logic and trace-based object-oriented application testing. In *Proceedings of the International Workshop on Object-Oriented Reengineering (WOOR04)*, 2004. 10.4

[DGD05] Coen De Roover, Kris Gybels, and Theo D'Hondt. Towards abstract interpretation for recovering design information. In *Proceedings of the First International Workshop on Abstract Interpretation of Object-oriented Languages (AIOOL05)*, volume 131 of *Electronic Notes in Theoretical Computer Science*, pages 15–25, May 2005. 10.4

[Dij76] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976. 1.3

[dlVFS+09] José Luis de la Vara, Michel Heluey Fortuna, Juan Sánchez, Cláudia Maria Lima Werner, and Marcos R. S. Borges. A requirements engineering approach for data modelling of process-aware information systems. In Witold Abramowicz, editor, *BIS*, volume 21 of *Lecture Notes in Business Information Processing*, pages 133–144. Springer, 2009. 10.5.1, 10.5.1

[dMPvdA+07] Ana Karla Alves de Medeiros, Carlos Pedrinaci, Wil M. P. van der Aalst, John Domingue, Minseok Song, Anne Rozinat, Barry Norton, and Liliana Cabral. An outlook on semantic business process mining and monitoring. In Robert Meersman, Zahir Tari, and Pilar Herrero, editors, *OTM Workshops (2)*, volume 4806 of *Lecture Notes in Computer Science*, pages 1244–1255. Springer, 2007. 10.3.3

[Dub04] Jean-Jacques Dubray. The seven fallacies of business process execution. (December), 2004. InfoQ,http://www.infoq.com/. 2.1.2, 10.5

[Duc99] Mireille Ducassé. Coca: an automated debugger for c. In *Proceedings of the 21st International Conference on Software engineering (ICSE99), Los Angeles, California, United States*, pages 504–513, 1999. 10.4

[Far85] David K Farkas. The concept of consistency in writing and editing. *Journal of Technical Writing and Communication*, 15(4):353–364, 1985. 3.1.2

[FDF98] William B. Frakes, Rubén Prieto Díaz, and Christopher J. Fox. Dare: Domain analysis and reuse environment. *Ann. Software Eng.*, 5:125–141, 1998. 3.1.1

[FF00] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. Technical report, 2000. 6.4.2

[FGY+04] Gerhard Fischer, Elisa Giaccardi, Yunwen Ye, Alistair G. Sutcliffe, and Nikolay Mehandjiev. Meta-design: a manifesto for end-user development. *Commun. ACM*, 47(9):33–37, 2004. 3.1.1

[FPWM08] Ian T. Foster, Savas Parastatidis, Paul Watson, and Mark Mckeown. How do i model state?: Let me count the ways. *Commun. ACM*, 51(9):34–41, 2008. 10.5.2, 11.2.1

[GBNT01] Jeff Gray, Ted Bapty, Sandeep Neema, and James Tuck. Handling crosscutting constraints in domain-specific modeling. *Commun. ACM*, 44(10):87–93, 2001. 6.4

[GC10] Sebastian Günther and Thomas Cleenewerck. Design principles for internal domain-specific languages: A pattern catalog illustrated by ruby. In *Proceedings of the 17th Conference On Pattern Languages Of Programs (PLOP 2010), Reno/Tahoe, Nevada, USA*, 2010. 3.1.2, 5.1

[GCC+04] Daniela Grigori, Fabio Casati, Malu Castellanos, Umeshwar Dayal, Mehmet Sayal, and Ming-Chien Shan. Business process intelligence. *Computers in Industry*, 16(3):321–343, April 2004. 10.3.2, 10.3.3

[GCD08] Oscar González, Rubby Casallas, and Dirk Deridder. Modularizing monitoring rules in business processes models. In Robert Meersman, Zahir Tari, and Pilar Herrero, editors, *OTM Workshops*, volume 5333 of *Lecture Notes in Computer Science*, pages 22–23. Springer, 2008. 1.5

[GCD09a] Oscar González, Rubby Casallas, and Dirk Deridder. Automating the implementation of analysis concerns in workflow applications. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009), Auckland, New Zealand*, pages 585–589. IEEE Computer Society, November 2009. 1.5

[GCD09b] Oscar González, Rubby Casallas, and Dirk Deridder. Mmc-bpm: A domain-specific language for business processes analysis. In Witold Abramowicz, editor, *BIS*, volume 21 of *Lecture Notes in Business Information Processing*, pages 157–168. Springer, 2009. 1.5

[GCD10] Oscar González, Rubby Casallas, and Dirk Deridder. Monitoring and analysis concerns in workflow applications: From conceptual specifications to concrete implementations. *Submitted to International Journal of Cooperative Information Systems*, 2010. 1.5

[GDJ02] Yann-Gaël Guéhéneuc, Rémi Douence, and Narendra Jussien. No Java without Caffeine – A tool for dynamic analysis of Java programs. In *Proceedings of the 17th Conference on Automated Software Engineering (ASE 2002)*, pages 117–126, September 2002. 10.4

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. 2.1.3

[GOA05] Simon Goldsmith, Robert O'Callahan, and Alex Aiken. Relational queries over program traces. In *Proceedings of the 20th annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages and Applications (OOPSLA05), San Diego, CA, USA*, pages 385–402, 2005. 10.4

[Gra01] Paul Graham. Five questions about language design, May 2001. Notes made for a panel discussion on programming language design at MIT, http://www.paulgraham.com/langdes.html. 3.1.2, 3.1.2

[GS03] Jack Greenfield and Keith Short. Software factories: assembling applications with patterns, models, frameworks and tools. In Ron Crocker and Guy L. Steele Jr., editors, *OOPSLA Companion*, pages 16–27. ACM, 2003. 6.3.2

[GTP07] Pau Giner, Victoria Torres, and Vicente Pelechano. Bridging the gap between bpmn and ws-bpel. m2m transformations in practice. In Nora Koch, Antonio Vallecillo, and Geert-Jan Houben, editors, *MDWE*, volume 261 of *CEUR Workshop Proceedings*. CEUR-WS.org, July 2007. 8.3.1

[Hel04] Pat Helland. Data on the outside vs. data on the inside. MSDN Library, 2004. http://msdn.microsoft.com/en-us/library/ms954587.aspx. 1.3

[HLD+05] Martin Hepp, Frank Leymann, John Domingue, Alexander Wahler, and Dieter Fensel. Semantic business process management: A vision towards using semantic web services for business process management. In Francis C. M. Lau, Hui Lei, Xiaofeng Meng, and Min Wang, editors, *ICEBE*, pages 535–540. IEEE Computer Society, 2005. 1.2, 10.3.3

[Hom04] Bart-Jan Hommes. *The Evaluation of Business Process Modeling Techniques*. PhD thesis, 2004. 2.1.2

[HPvD09] Felienne Hermans, Martin Pinzger, and Arie van Deursen. Domain-specific languages in practice: A user study on the success factors. In Andy Schürr and Bran Selic, editors, *MoDELS*, volume 5795 of *Lecture Notes in Computer Science*, pages 423–437. Springer, 2009. 5.2, 5.2.1, 9.6.1

[HS05] Michael N. Huhns and Munindar P. Singh. Service-oriented computing: Key concepts and principles. *IEEE Internet Computing*, 9(1):75–81, 2005. 10.7

[IBM02] IBM. Business process execution language for web services, July 2002. http://www.ibm.com/developerworks/library/specification/ws-bpel/. 1.1, 2.1.3, 2.1.3, 9.2, 10.7

[Int] Intalio, The Open Source Business Process Platform. http://www.intalio.com/. 2.1.2, 8.3.1, 8.3.1

[JB88] Robert M. Herndon Jr. and Valdis Berzins. The realizable benefits of a language prototyping language. *IEEE Trans. Software Eng.*, 14(6):803–809, 1988. 3.1.1, 9.4

[jBP] jBPM homepage. http://www.jboss.org/jbossjbpm/. 2.1.3

[JN04] Ivar Jacobson and Pan-Wei Ng. *Aspect-Oriented Software Development with Use Cases (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2004. 6.4

[JPD] JPDL homepage. http://www.jboss.org/jbossjbpm/jpdl/. 1.1, 1.2.2, 2.1.3, 2.1.3, 7.6

[JSC03] Jun-Jang Jeng, Josef Schiefer, and Henry Chang. An agent-based architecture for analyzing business processes of real-time enterprises. In *Proceedings of the 7th International Enterprise Distributed Object Computing Conference (EDOC 2003), Brisbane, Australia*, pages 86–97. IEEE Computer Society, September 2003. 10.2.1, 10.2.1

[Kas06] Peter S. Kastner. The business process management benchmark report. Technical report, Aberdeen Group, 2006. 2.1.4

[KCH+90] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-21, Pittsburgh, PA, USA, November 1990. 3.1.1

[KEvS+05] Gavin King, Steve Ebersole, Anton van Straaten, Mikheil Kapanadze, Greg Luck, Emmanuel Bernard, Mathias Bogaert, Jason Carreira, Doug Currie, Gabe Hicks, David Channon, Helge Schulz, Steve Molitor, Colm O' Flaherty, and etc. Hibernate api documentation. Web, 2005. 8.2.2

[KF04] Murat Karaorman and Jay Freeman. jmonitor: Java runtime event specification and monitoring library. In *Proceedings of the Fourth International Workshop on Run-time Verification (RV04)*, volume 113 of *Electronic Notes in Theoretical Computer Science*, pages 181–200, 2004. 10.4

[KLBM08] Tomaz Kosar, Pablo E. Martínez López, Pablo A. Barrientos, and Marjan Mernik. A preliminary study on various implementation approaches of domain-specific language. *Information & Software Technology*, 50(5):390–405, 2008. 3.1.1, 3.1.1

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997. 1.2.2, 1.4, 3.3, 6.4

[KMB+96] Richard B. Kieburtz, Laura McKinney, Jeffrey M. Bell, James Hook, Alex Kotov, Jeffrey Lewis, Dino Oliva, Tim Sheard, Ira Smith, and Lisa Walton. A software engineering experiment in software component generation. In *Proceedings of the 18th international conference on Software engineering (ICSE), Berlin, Germany*, pages 542–552. IEEE Computer Society, 1996. 3.1.1

[Kru92] Charles W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, 1992. 3.1.1

[KV06] Audris Kalnins and Valdis Vitolins. Use of uml and model transformations for workflow process definitions. *CoRR*, abs/cs/0607044, 2006. 11.2.1

[KWB03] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. 6.3.1

[Ley01] F Leymann. Web services flow language (wsfl 1.0). Technical report, IBM Corporation, 2001. Stuttgart. 10.7

[Lik32] R. Likert. A technique for the measurement of attitudes. *Archives of Psychology*, 22(140):1–55, 1932. 5.2.1, 9.6.1

[LPY07] Christina Lau, Scott Peddle, and Shili Yang. Gathering monitoring metrics to analyze your business process. Technical report, IBM, December 2007. 10.2.1

[LR94] David A. Ladd and J. Christopher Ramming. Two application languages in software production. In *Proceedings of the USENIX Very High Level Languages Symposium (VHLLS 1994), Santa Fe, New Mexico*, pages 10–10. USENIX Association, 1994. 3.1.1

[LR99] Frank Leymann and Dieter Roller. *Production Workflow: Concepts and Techniques.* Prentice Hall PTR, September 1999. 10.3.1

[MCG04] Tom Mens, Krzysztof Czarnecki, and Pieter Van Gorp. 04101 discussion - a taxonomy of model transformations. In Jean Bézivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, volume 04101 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2004. 6.3.1

[McK76] William M. McKeeman. Programming language design. In Friedrich L. Bauer and Jürgen Eickel, editors, *Compiler Construction*, volume 21 of *Lecture Notes in Computer Science*, pages 514–524. Springer, 1976. 3.1.2

[McL96] M. McLellan. *Workflow Metrics - One of the great benefits of workflow management*, pages 301–318. Praxis des Workflow-Management. Braunschweig, 1996. 10.2.1

[MHH07] Derek Miers, Paul Harmon, and Curt Hall. The 2007 bpm suites report. Technical report, BPTrends, 2007. 1.1, 1.2, 2.2, 10.2.1

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005. 3.1.1, 3.1.1, 3.1.1, 5.2.1

[MMA07] Christof Momm, Robert Malec, and Sebastian Abeck. Towards a model-driven development of monitored processes. In Andreas Oberweis, Christof Weinhardt, Henner Gimpel, Agnes Koschmider, Victor Pankratius, and Björn Schnizler, editors, *Wirtschaftsinformatik (2)*, pages 319–336. Universitaetsverlag Karlsruhe, 2007. 10.2.2

[MS04] Carolyn McGregor and Josef Schiefer. A web-service based framework for analyzing and measuring business performance. *Inf. Syst. E-Business Management*, 2(1):89–110, 2004. 10.2.1, 10.2.1

[MSzM06] Carolyn McGregor, Josef Schiefer, and Michael zur Muehlen. A shareable web service-based intelligent decision support system for on-demand business process management. *International Journal of Business Process Integration and Management*, 1(3):156–174, 2006. 3.2.1, 10.3.1

[Nar93] Bonnie A. Nardi. *A Small Matter of Programming: Perspectives on End User Computing.* MIT Press, Cambridge, MA, USA, 1993. 3.1.1

[NBA05] István Nagy, Lodewijk Bergmans, and Mehmet Aksit. Composing aspects at shared join points. In Robert Hirschfeld, Ryszard Kowalczyk, Andreas Polze, and Mathias Weske, editors, *NODe/GSEM*, volume 69 of *LNI*, pages 19–38. GI, 2005. 11.2.3

[NC03] Anil Nigam and Nathan S. Caswell. Business artifacts: An approach to operational specification. *IBM Systems Journal*, 42(3):428–445, 2003. 10.5.1, 10.5.1

[Nor98] Nortel. Workflow scenario: Trouble ticket. Technical Report OMG Document Number bom/98-03-10, March 1998. 1.2.1, 9.1

[OMG06a] OMG. Business process modeling notation, v1.1. Technical Report OMG Document Number formal/2008-01-17, February 2006. 1.1, 2.1.2, 2.1.2, 11.2.1

[OMG06b] OMG. Object constraint language, v2.0. Technical Report OMG Document Number formal/2006-05-01, May 2006. 3.3

[Ope] OpenArchitectureWare (oAW) website. http://www.openarchitectureware.org. 7.3.2, 8.1

[Oraa] Oracle Business Process Management. http://www.oracle.com/technologies/bpm/index.html. 2.1.2

[Orab] Oracle BPEL Process Manager. http://otn.oracle.com/bpel. 2.1.3

[OvdADH] Chun Ouyang, Wil M. P. van der Aalst, Marlon Dumas, and Arthur H. M. Ter Hofstede. Translating bpmn to bpel. http://is.tm.tue.nl/staff/wvdaalst/BPMcenter/reports/2006/BPM-06-02.pdf. 2.1.3

[Pal09] Nathaniel Palmer. 2009 bpm state of the market report. Technical report, BPM.com, 2009. 2.1.4

[Pap03] Mike P. Papazoglou. Service-oriented computing: Concepts, characteristics and directions. In *Proceedings of the 4th International Conference on Web Information Systems Engineering (WISE 2003), Rome, ItalyWISE*, pages 3–12. IEEE Computer Society, December 2003. 10.7

[PCl+07] Chankyu Park, Ho-Jin Choi, Danhyung lee, Sungwon Kang, Hyun-Kyu Cho, and Joo-Chan Sohn. Knowledge-based aop framework for business rule aspects in business process. *ETRI Journal*, 29(4):477–488, 2007. 10.6

[PD07] Carlos Pedrinaci and John Domingue. Towards an ontology for process monitoring and mining. In Martin Hepp, Knut Hinkelmann, Dimitris Karagiannis, Rüdiger Klein, and Nenad Stojanovic, editors, *SBPM*, volume 251 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007. 10.3.3

[PDB+08] Carlos Pedrinaci, John Domingue, Christian Brelage, Tammo van Lessen, Dimka Karastoyanova, and Frank Leymann. Semantic business process management: Scaling up the management of business processes. In *Proceedings of the 2th IEEE International Conference on Semantic Computing (ICSC 2008), Santa Clara, California, USA*, pages 546–553. IEEE Computer Society, August 2008. 10.3.3

[PK01] Shari Lawrence Pfleeger and Barbara A. Kitchenham. Principles of survey research. *SIGSOFT Softw. Eng. Notes*, 26(6):16–18, 2001. 5.2.1, 9.6.1

[PMHD09] Carlos Pedrinaci, Ivan Markovic, Florian Hasibether, and John Domingue. Strategy-driven business process analysis. In Witold Abramowicz, editor, *BIS*, volume 21 of *Lecture Notes in Business Information Processing*, pages 169–180. Springer, 2009. 10.3.3

[PTDL07] Mike P. Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-oriented computing: State of the art and research challenges. *IEEE Computer*, 40(11):38–45, 2007. 10.7

[RAvdAM06] Nick Russell, Arthur, Wil M. P. van der Aalst, and Natalya Mulyar. Workflow control-flow patterns: A revised view. Technical Report BPM-06-22, BPMcenter.org, 2006. 6.4.2

[Ric02] Tamar Richner. *Recovering Behavioral Design Views: a Query Based Approach*. PhD thesis, Universität Bern, Philosophisch-naturwissenschaftlichen Fakultät, May 2002. 10.4

[RLVDA03] Hajo A. Reijers, Selma Limam, and Wil M. P. Van Der Aalst. Product-based workflow design. *J. Manage. Inf. Syst.*, 20(1):229–262, 2003. 10.5.1

[RRIG06] Michael Rosemann, Jan Recker, Marta Indulska, and Peter Green. A study of the evolution of the representational capabilities of process modeling grammars. In Eric Dubois and Klaus Pohl, editors, *CAiSE*, volume 4001 of *Lecture Notes in Computer Science*, pages 447–461. Springer, 2006. 2.1.2

[RtHEvdA04] N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Workflow data patterns. Technical Report QUT number FIT-TR-2004-01, Queensland University of Technology, Brisbane, 2004. 5.3.1

[RtHEvdA05] Nick Russell, Arthur H. M. ter Hofstede, David Edmond, and Wil M. P. van der Aalst. Workflow data patterns: Identification, representation and tool support. In Lois M. L. Delcambre, Christian Kop, Heinrich C. Mayr, John Mylopoulos, and Oscar Pastor, editors, *Proceedings of the 24th International Conference on Conceptual Modeling (ER 2005), Klagenfurt, Austria*, volume 3716 of *Lecture Notes in Computer Science*, pages 353–368. Springer, October 2005. 4.1.2, 5.3.1

[RvdA08] Anne Rozinat and Wil M. P. van der Aalst. Conformance checking of processes based on monitoring real behavior. *Inf. Syst.*, 33(1):64–95, 2008. 10.3.4, 10.3.4

[SB06] Volker Stolz and Eric Bodden. Temporal assertions using aspectj. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 144(4):109–124, 2006. 10.4

[SCDS02] Mehmet Sayal, Fabio Casati, Umeshwar Dayal, and Ming-Chien Shan. Business process cockpit. In *Proceedings of the 28th international conference on Very Large Data Bases (VLDB 2002), Hong Kong, China*, pages 880–883. VLDB Endowment, 2002. 10.3.2

[Sch06] Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *IEEE Computer*, 39(2):25–31, 2006. 1.4, 6.3

[Sch07] Arno Schmidmeier. Aspect oriented dsls for business process implementation. In *Proceedings of the 2nd workshop on Domain specific aspect languages (DSAL 2007), Vancouver, British Columbia, Canada*, page 5. ACM, 2007. 10.2.1

[SG97] Diomidis Spinellis and V. Guruprasad. Lightweight languages as software engineering tools. In *Proceedings of the Conference on Domain-Specific Languages, Santa Barbara, California, USA*, pages 67–76. USENIX, October 1997. 3.1.1

[SJVD09] Mario Sánchez, Camilo Jiménez, Jorge Villalobos, and Dirk Deridder. Building a multimodeling framework using executable models. In Manuel Oriol and Bertrand Meyer, editors, *TOOLS EUROPE 2009*, volume 33 of *LNBIP*, pages 157–174, Berlin - Heidelberg, July 2009. Springer-Verlag. 2.1.3

[SK95] Kenneth Slonneger and Barry Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. 3.1.1

[SK03] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, 2003. 6.3.1

[SK04] Shane Sendall and Jochen Küster. Taming model round-trip engineering. In *Proceedings of the Workshop on Best Practices for Model-Driven Software Development*, Vancouver, Canada, 2004. 11.2.2

[Spi01] Diomidis Spinellis. Notable design patterns for domain-specific languages. *Journal of Systems and Software*, 56(1):91–99, 2001. 3.1.1, 5.2.1

[SSK+06] Andrea Schauerhuber, Wieland Schwinger, Elisabeth Kapsammer, Werner Retschitzegger, and Manuel Wimmer. Towards a common reference architecture for aspect-oriented modeling. In *Proceedings of the 8th International Workshop on Aspect-Oriented Modeling (AOM)*, 2006. 11.2.1

[SSK+07] Andrea Schauerhuber, Wieland Schwinger, Elisabeth Kapsammer, Werner Retschitzegger, Manuel Wimmer, and Gerti Kappel. A survey on aspect-oriented modeling approaches. Technical report, Business Informatics Group, Institute of Software Technology and Interactive Systems, Vienna University of Technology, 2007. 11.2.3

[SZNS06] Sherry X. Sun, J. Leon Zhao, Jay F. Nunamaker, and Olivia R. Liu Sheng. Formulating the data-flow perspective for business process management. *Info. Sys. Research*, 17(4):374–391, 2006. 10.5.1, 10.5.1

[The08] The Eclipse Foundation. SOA Tools Platform (STP) project, 2008. BPMN Modeler. 2.1.2, 7.3.1, 8.2.1

[tHvdAAR10] A. M. ter Hofstede, W. M. P. van der Aalst, M. Adamns, and N. Russell, editors. *Modern Business Process Automation: YAWL and its Support Environment*. Springer, 2010. 2.1.4

[TOHJ99] Peri L. Tarr, Harold Ossher, William H. Harrison, and Stanley M. Sutton Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st international conference on Software engineering (ICSE 1999), Los Angeles, California, USA*, pages 107–119. ACM, 1999. 1.2.2, 6.4

[VCJ04] Bart Verheecke, María Agustina Cibrán, and Viviane Jonckers. Aspect-oriented programming for dynamic web service monitoring and selection. In Liang-Jie Zhang, editor, *ECOWS*, volume 3250 of *Lecture Notes in Computer Science*, pages 15–29. Springer, 2004. 10.2.1

[vdAtHKB03] Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, Bartek Kiepuszewski, and Alistair P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003. 2.1.4

[vdAtHW03] Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and Mathias Weske. Business process management: A survey. In Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and Mathias Weske, editors, *Proceedings of the Business Process Management International Conference (BPM 2003), Eindhoven, The Netherlands*, volume 2678 of *Lecture Notes in Computer Science*, pages 1–12. Springer, June 2003. 1.1

[vdAvH04] Wil M. P. van der Aalst and Kees M. van Hee. *Workflow Management : Models, Methods, and Systems*, volume 1 of *MIT Press Books, Cooperative Information Systems*. The MIT Press, December 2004. 1.1, 2.1

[vdAW05] Wil van der Aalst and A.J.M.M. (Ton) Weijters. *Process Mining*, chapter 10, pages 235–255. Process Aware Information Systems: Bridging People and Software Through Process Technology. Wiley-Interscience, September 2005. 10.3.3

[vdDB06] Markus von den Driesch and Tobias Blickle. *Operational, Tool-Supported Corporate Performance Management with the ARIS Process Performance Manager*, pages 45–64. Corporate Performance Management. Springer, aris in practice edition, 2006. 10.2.1, 10.2.1

[vDdMV+05] Boudewijn F. van Dongen, Ana Karla A. de Medeiros, H. M. W. Verbeek, A. J. M. M. Weijters, and Wil M. P. van der Aalst. The prom framework: A new era in process mining tool support. In Gianfranco Ciardo and Philippe Darondeau, editors, *ICATPN*, volume 3536 of *Lecture Notes in Computer Science*, pages 444–454. Springer, 2005. 10.3.3

[vDK98] Arie van Deursen and Paul Klint. Little languages: little maintenance? *Journal of Software Maintenance*, 10(2):75–92, 1998. 3.1.1

[Vit04] Valdis Vitolins. Business process measures. *Computer Science and Information Technologies, Databases and Information Systems Doctoral Consortium, Scientific Papers University of Latvia*, 673:186–197, 2004. University of Latvia. 10.2.2

[WCL+05] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey, and Donald F. Ferguson. *Web Services Platform Architecture : SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall PTR, March 2005. 10.7

[Wei98] Gerald M. Weinberg. *The psychology of computer programming (silver anniversary ed.)*. Dorset House Publishing Co., Inc., New York, NY, USA, 1998. originally published in 1971. 3.1.2

[WfM98] WfMC. Workflow management coalition: Audit data specification, 1998. Draft 1.1.a. Document Number WfMC-TC-1015. Winchester. 2.2.4

[wfm99] Workflow Management Coalition Terminology & Glossary, 1999. Workflow Management Coalition, Document Number WFMC-TC-1011, Document Status - Issue 3.0. 1.1, 2.1, 2.1.1

[Whi04] Steven A. White. Business process modeling notation (bpmn). version 1.0 - may 3, 2004. Technical report, BPMI.org, 2004. 2.1.2

[Whi05] Stephen A. White. Using bpmn to model a bpel process, January 2005. IBM, http://www.ibm.com/us/. 8.3.1

[WK03] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. 3.3

[WK05] Jianrui Wang and Akhil Kumar. A framework for document-driven workflow systems. In Wil M. P. van der Aalst, Boualem Benatallah, Fabio Casati, and Francisco Curbera, editors, *Business Process Management*, volume 3649, pages 285–301, 2005. 10.5.1

[WL99] David M. Weiss and Chi Tau Robert Lai. *Software product-line engineering: a family-based software development process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. 3.1.1

[WV99] Mathias Weske and Gottfried Vossen. *Workflow Languages*, pages 359–379. Handbook on Architectures of Information Systems. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999. 2.1.3

[XPD] XPDL homepage. http://www.wfmc.org/xpdl.html. 1.1, 2.1.3

[zM00] Michael zur Muehlen. *Workflow-based Process Controlling - Or: What You Can Measure You Can Control*, pages 61–77. Workflow Handbook 2001. Future Strategies, 2000. 2.2

[zM04] Michael zur Muehlen. *Workflow-based Process Controlling. Foundation, Design, and Implementation of Workflow-driven Process Information Systems.*, volume 6 of *Advances in Information Systems and Management Science*. Logos, Berlin, 2004. 1.1, 2.1

[zMI10] Michael zur Muehlen and Marta Indulska. Modeling languages for business processes and business rules: A representational analysis. *Inf. Syst.*, 35(4):379–390, 2010. 2.1.2

[zMR00] Michael zur Muehlen and Michael Rosemann. Workflow-based process monitoring and controlling - technical and organizational issues. In *Proceedings of the 33rd Hawaii International Conference on System Science (HICSS)*, volume 6, pages 1–10. IEEE Computer Society, 2000. (document), 1.1, 2.2.1, 2.2, 2.2.2, 2.2.4, 3.2.1, 10.3.5

Index