Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS

Štěpán Poljak

Conceptual Modeling of Business Artifacts and their Implementation as Active XML

Department of Software Engineering
Malostranské náměstí 25
Prague, Czech Republic

Supervisor of the master thesis:	Mgr. Martin Nečaský, Ph.D.
Study programme:	Informatics
Specialization:	Software systems

Prague 2013

I would like to thank to my supervisor Mgr. Martin Nečaský, PhD. for his suggestions, thorough notes, provided related research material and text corrections. I would also like to thank to Mgr. Jakub Malý for his willingness to answer my questions regarding eXolutio.

Název práce: *Konceptuální modelování business artefaktů a jejich implementace v Active XML*
Autor: *Štěpán Poljak*
Katedra: *Katedra Softwarového Inženýrství*
Vedoucí diplomové práce: *Mgr. Martin Nečaský, Ph.D.*

Abstrakt: *V předložené práci se zabýváme konceptuálním modelováním business artefaktů a jejich implementací v Active XML. Business artefakty jsou klíčové entity business procesů, které se v průběhu těchto procesů vyvíjí v rámci svého životního cyklu. Pro popis životního cyklu existuje více možných metod. V naší práci vycházíme z nově vznikající metody zvané Guard-Stage-Milestone meta-model a zabýváme se otázkou, jak vhodně využít a rozšířit současný framework pro modelování XML schémat tak, aby podporoval modelování business artefaktů. Zároveň se zabýváme návrhem vhodné reprezentace business artefaktů pomocí Active XML. V neposlední řadě se zabýváme otázkou, jak přeložit vytvořený model do navržené reprezentace v Active XML tak, aby bylo možné okamžitě demonstrovat funkčnost navrženého modelu. Důležitou součástí práce je implementace navrženého rozšíření do frameworku pro konceptuální modelování a implementace prototypového systému pro spouštění Active XML reprezentací přeložených modelů. Práce také detailně uvádí čtenáře do jednotlivých použitých konceptů a představuje podobný existující přístup k reprezentaci artefaktů pomocí Active XML.*

Klíčová slova: *Active XML, business artefakty, konceptuální modelování*

Title: *Conceptual Modeling of Business Artifacts and their Implementation as Active XML*

Author: *Štěpán Poljak*

Department: *Department of Software Engineering*

Supervisor: *Mgr. Martin Nečaský, Ph.D.*

Abstract: *In the present work we study conceptual modeling of business artifacts and their implementation in Active XML. Business artifacts are key conceptual entities of business processes that develop in their lifecycle during these processes. There are several possible methods for definition of artifact lifecycles. In this work, we make use of emerging method called Guard-Stage-Milestone meta-model and we study the question on how to appropriately use and extend current framework for conceptual modeling of XML schemas in order to support modeling of business artifacts. We also deal with the issue of design of suitable representation of business artifacts using Active XML. Last but not least, we study the question how to translate defined model into proposed Active XML representation, so that it was possible to immediately use and demonstrate functionality of defined model. Important part of this work is an implementation of proposed extension and a prototype implementation of system for execution of Active XML representations of translated models. The present work also introduces the reader in individual used concepts and describes similar existing approach for Active XML representation of business artifacts.*

Keywords: *Active XML, business artifacts, conceptual modeling*

Contents

1. Introduction

1.1 Introduction and motivation

Business process modeling is a crucial part in describing, structuring and understanding the business operations of the enterprise. As such, it is becoming more and more important to model business processes effectively. Nowadays there are many business process modeling paradigms and probably the most widely used are activity-centric approaches based on activity flows. Popular representative in this category is *Business Process Modeling Notation* defined by OMG [29]. Work-flow based approaches describe primarily the functional aspects of business operations and focus on sequencing of individual activities into the overall process. This makes it relatively easy to understand the structure of the business processes and how these processes progress over time. However, the same cannot be said about the data that these business processes use and produce. The reason for this is that work-flow based approaches consider the data aspects merely as a secondary issue, usually only in a context of single activities. This makes it hard to understand the structure of the data and how they change during the business process.

Artifact-centric approaches provide a possible solution to this problem. These approaches consider data aspects as an integral part of business operations by focusing on key conceptual entities that are central points of business operations, so called business artifacts. These artifacts contain both information and lifecycle model. Artifact-centric model is described using interactions between these artifacts, specifying how these artifacts evolve during the business operations and describing activities and tasks associated to these artifacts.

Guard-Stage-Milestone meta-model is an emerging artifact-centric approach to modeling business processes introduced by IBM Research [7]. It is however an abstract model and therefore it does not strictly specify how its information and lifecycle model should be implemented. But if we want to represent artifacts, we have to think about their concrete representation. A promising approach is to realize artifacts information model with XML, because XML is a data format that is highly portable, extensible and widely used nowadays. It is also a popular exchange format that solves the problem with heterogeneity of distributed systems, so web services usually communicate by exchanging XML documents using SOAP protocol. XML is suitable for artifact data representation, because it enables us to easily represent objects using hierarchy.

Using XML as a realization of an artifact information model, we must define XML schema that defines this information model. The area of effective modeling of XML schemas is a subject for research itself. Currently, there is an interesting approach, called *Conceptual model for XML*. This approach is based on model driven architecture. It allows us to model XML on a level independent on target platform, then derive concrete XML formats based on this level and finally automatically generate XML Schemas based on these models, which greatly supports evolution of XML schemas. This model is implemented in a tool *eXolutio*, so we could use this tool to define information model of business artifacts. However we also need to define the lifecycle models of business artifacts, but eXolutio is currently only data oriented and does not have a support for modeling of business processes. For this reason we want to extend eXolutio in order to support modeling of business artifact lifecycles. This way, eXolutio would provide a way to connect abstract GSM model with the world of XML.

We said that artifact lifecycle model defines how the artifact evolves over time. Since we want to realize the artifact information model using XML, it makes sense to represent changes to the information model using XQuery. Also, artifact-centric model specifies concrete tasks related to the artifact instances. These tasks can have many forms and it often happens that they have a form of a web service invocation. This means that we have a combination of XML data and web services that operate over these data. Fortunately, there already exists a framework that combines XML with web services, called *Active XML*. This framework is centered around XML documents, which are valid XML documents that can contain embedded service calls. When these service calls are invoked, their results can enrich the original XML document. As a result of these observations, it seems promising to define business artifact-centric models using XML conceptual model in eXolutio and implement these models using Active XML.

1.2 Aim of this thesis

As we described, combination of the GSM model, Active XML and eXolutio conceptual modeling seems promising, because it allows us to bring the GSM models into the world of XML. Therefore, the aim of this thesis is to connect these three approaches into one single framework. This goal is composed from three main parts:

Propose Active XML representation At first, we need to propose a suitable implementation for abstract GSM models using Active XML. We need to analyze how business artifacts from abstract GSM model can be mapped into concrete XML representation and how to use Active XML to implement evolution of business artifacts. Also, we need to analyze how to implement a system on top of Active XML that would manage and execute concrete Active XML representation and demonstrate the functionality of defined model.

Propose eXolutio extension Next we need to analyze how to connect eXolutio conceptual modeling with definition of GSM information models. More or less in parallel, we need to analyze how to extend eXolutio to support definition of GSM lifecycle models and design an extension that would allow us to automatically generate proposed Active XML representation from defined GSM model.

Implementation Finally, we need to implement proposed extensions into eXolutio and implement an execution system on top of Active XML.

1.3 Organization of this thesis

The rest of this thesis is organized as follows. In Chapter 2 we introduce business artifacts and artifact-centric modeling and then we describe Guard-Stage-Milestone meta-model in detail and provide deeper motivation for this modeling approach. In Chapter 3, we describe Active XML framework and its current prototypical implementation. We also describe related existing approach to connect business artifacts with Active XML, called AXML Artifact Model. In chapter 4, we describe Conceptual model for XML and provide basic overview of eXolutio, a prototypical tool for conceptual modeling that implements concepts described in chapter 4.

In chapter 5, we analyze and design possible representations of GSM model and business artifacts in Active XML and discuss what we require from such executable representation. Building on that, in chapter 6 we analyze how to extend eXolutio to support modeling of business artifacts using Guard-Stage-Milestone model and how to generate defined models into an executable representation from chapter 5. In chapter 7 we describe concrete implementation of executable system that would complement and execute generated executable representation, called Artifact Service Center.

2. Business Artifacts and Guard-Stage-Milestone meta-model

This section provides an introduction to business artifacts, business artifact-centric modeling and Guard-Stage-Milestone meta-model. We first informally describe the topic and we provide formal foundations in the following section.

2.1 Motivation for Business Artifacts

Business artifact centric modeling is a new promising approach for defining business process models. Nowadays there are many other business process modeling paradigms, and probably the most widely used are activity-centric approaches based on activity flows. These approaches describe mainly the functional aspects of business operations and focus on sequencing of activities into the overall process. The problem is that they pay only secondary attention to data aspects and usually consider data only in a context of single activities, for instance as inputs and outputs of these activities or they ignore data aspects altogether. The problem that comes from considering data only as the secondary-citizens in the modeling process is that we do not have the complete view over the information model and its evolution during the execution of business operations. This can lead to problems when dealing with change, as indicated by many authors [3]. Authors also argue that activity-centric approaches are too restrictive by focusing primarily on what should be done, instead of what can be done, which results to inflexible work-flows, errors and inefficiencies.

The paper [3] describes this as *context tunneling.* The problem is that work-flow systems concentrate on the control flow of activities in order to achieve some business goals, but move to the background the context, the data related to the entire process and provide only the information that is needed for executing a specific task in a single activity. The context tunneling can be avoided by providing all information available. The paper describes case handling approach, which is a method that focuses on knowledge intensive business processes. Case handling is based on case files that contain all the relevant information to successfully complete the case. It also associates activities to this data that can or must be performed in order to complete the case. But unlike the work-flows the case handling permits more freedom in how the processing of case files is organized, because there is no control flow to determine how these activities have to be performed. Instead the knowledge worker who is responsible for a particular case actively decides on how the goal of that case is reached and the case handling system only assists him.

Artifact-centric modeling is closely related to the case handling, because both approaches have strong emphasis on the key conceptual entities [4] and both addresses the problem by promoting the conceptual business entities to become first-class citizens in a business operation model. Artifact-centric modeling is based on the notion of business artifacts, also called business entities with lifecycles (BEL). These are key conceptual entities that are central point of business processes. Artifact-centric modeling describes business process in terms of interactions between these artifacts and how the business operations change these artifacts. An artifact type contains both information model that specifies all of the business-relevant data about entities of that type and lifecycle model, that specifies how the entity evolves through the business process by responding to incoming events and invoking tasks. When the artifacts are created, they usually have only few attributes defined and as they evolve and move through their lifecycle, these attributes are updated and more attributes are filled. An important premise is that at any point in the processing, all business-relevant information about an artifact instance should be stored in that instance. "*This implies that relevant information about an artifact instance is never hidden in the instance's current position in an activity flow*" [9]. Artifact existence can span for several years [10, page 5]. For example the financial application described in [12] uses business artifact Financial Deal that represents a loan from one party to another secured by some collateral. This artifact instance can progress through activities like checking on the borrower and the collateral, completing negotiations and a contract, and managing the periodic payments until termination of the contract.

Business artifacts were first described in [1] and "*since 2003 IBM Research has been developing meta-models, methods, tools, user-centric paradigms, and other technologies in support of the artifact-centric paradigm*" [2]. IBM also applied the artifact-centric modeling in many real customer projects, see for example [5]. The team also develops the Barcelona prototype engine that supports the GSM meta-model. GSM and Barcelona prototype are being used by the EU-funded Artifact-Centric Service Inter-operation (ACSI) project [6].

FSM lifecycle model First works on artifact-centric models were based on finite state machines (FSM) to define artifact lifecycles. In FSM lifecycle approach, each machine state represents a possible stage in the artifact lifecycle and each state transition can have associated tasks that are started when the transition occurs. Transitions have associated conditions and when these conditions are satisfied, the transition occurs. Conditions can refer to changes in artifact information model or to externals events that artifact can receive. These events can be sent either by user or machine. The paper [2] recommends that conditions associated to transitions between artifact stages should be focused on the minimal business requirements needed to pass between them. Complex conditions should be specified at the level of associations. Associations are constraints on tasks that determine precedence relationships among the tasks and which tasks are invoked when the state transition occurs.

GSM lifecycle model More recent papers have introduced the Guard-Stage-Milestone (GSM) meta-model [4, 7, 8] that is the evolution of previous research on business artifacts. The GSM meta-model lifecycles are much more declarative than the finite state machine variants. Unlike previous work, that assumed that services must be performed in sequence, in GSM meta-model services and other aspects may be running in parallel. The GSM meta-model also supports hierarchy of activities.

The paper [7] discusses that the foundation of the research leading to GSM has been to create a meta-model for specifying business processes that:

- *"Will help business-level stakeholders to gain insight into and understanding of their business operations.*
- *Is centered around intuitively natural constructs that correspond closely to how business-level stakeholders think about the operations of their business.*
- *Can provide a high-level, abstract view of the operations, and gracefully incorporate enough detail to be executable.*
- *Can support a spectrum of styles for specifying business operations and processes, from the highly prescriptive (as found in for example BPMN) to the highly descriptive (as found in Adaptive Case Management systems).*
- *Can serve as the target into which intuitive, informal, and imprecise specifications of the business operations (for example in terms of business scenarios) can be mapped." [7]*

The core of the operational semantics is based on the rules inspired by Event-Condition-Action (ECA) rules. These rules specify how the artifacts evolve over the time and define conditions that determine when the activities start, when they terminate and how the business relevant objectives are achieved or invalidated during the business processes. This declarative approach provides greater flexibility when dealing with business changes then imperative approaches found in finite state machines variants. The operation semantics is centered around GSM Business steps (B-steps) that corresponds to the smallest unit of business-relevant change that can occur to a GSM system and they represent to the processing of single incoming event. The semantics focuses on how this incoming event changes the snapshot (description of all relevant aspects of a GSM system at a given moment of time).

The semantics for B-steps has three equivalent formulations: [4]

Incremental This corresponds to the incremental application of the ECA rules and provides an intuitive way to describe the operational semantics of a GSM model and provides a natural, direct approach for implementing GSM.

Fix-point This provides a concise top-down description of the description of the effect of a single incoming on an artifact snapshot. This is useful for developing alternative implementations for GSM, and optimizations of them, something especially important if highly scalable, distributed implementations are to be created.

Closed-form : This provides a characterization of snapshots and the effects of incoming events using what is essentially a first-order logic formula. This permits the application of previously developed verification techniques to the GSM context.

2.2 Informal description

We will now describe the GSM meta-model. As we said earlier, the GSM meta-model is a data-centric approach, which is the main contrast to traditional activity-based approaches. We will now illustrate simple business process example, first specified using *Business Process Model and Notation* and later using *Guard-Stage-Milestone meta-model*.

Example The following example describes a simple business process for a cash withdrawal. Customer comes to the bank, fills a withdrawal request form, comes to the reception desk and submits the form to the receptionist. The receptionist checks client identity and confirms that declared bank account exists. If not, the withdrawal is refused. Otherwise receptionist checks if balance of the client account is higher than requested monetary amount. If not, the withdrawal is refused. Otherwise, client account is updated and cash is delivered.

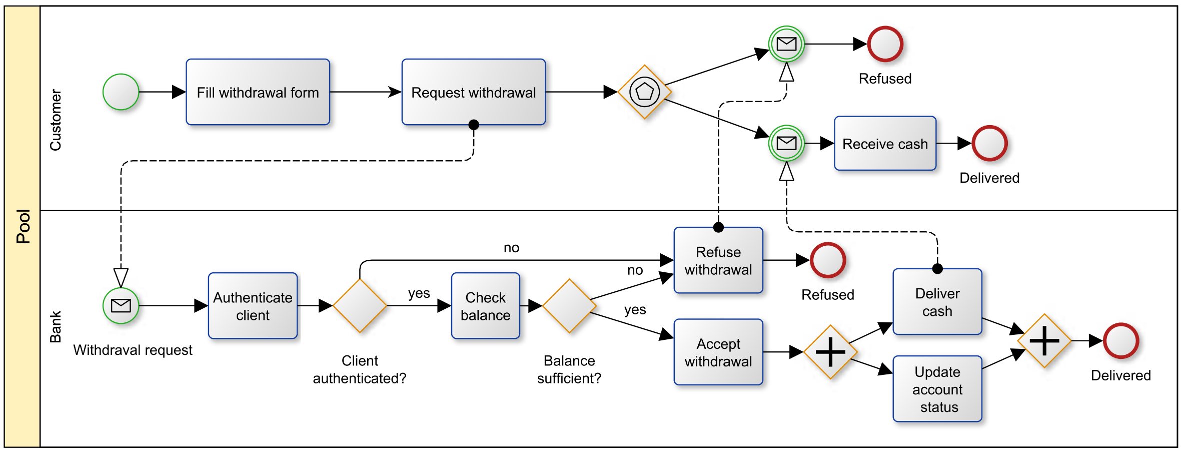


Figure 2.1: Withdrawal business process using BPMN

The figure 2.1 shows the withdrawal process using BPMN. We can see that this model focuses on activities and flows between them. The figure 2.2 shows (almost) the same withdrawal process using GSM meta-model. We can see that this model considers both activities, shown in the top part of the figure, and data aspects, shown in the bottom part of the figure. As this figure indicates, flows between activities are not specified by explicit lines, like in BPMN, but by declarative rules. These rules describe under which conditions individual activities start or terminate. This also intuitively implies that the conditional flows, specified using diamond gateways in BPMN, are described by declarative rules as well in GSM meta-model. Many visualization concepts used in BPMN have different meaning in GSM meta-model. We will now focus on core constructs of the GSM meta-model.

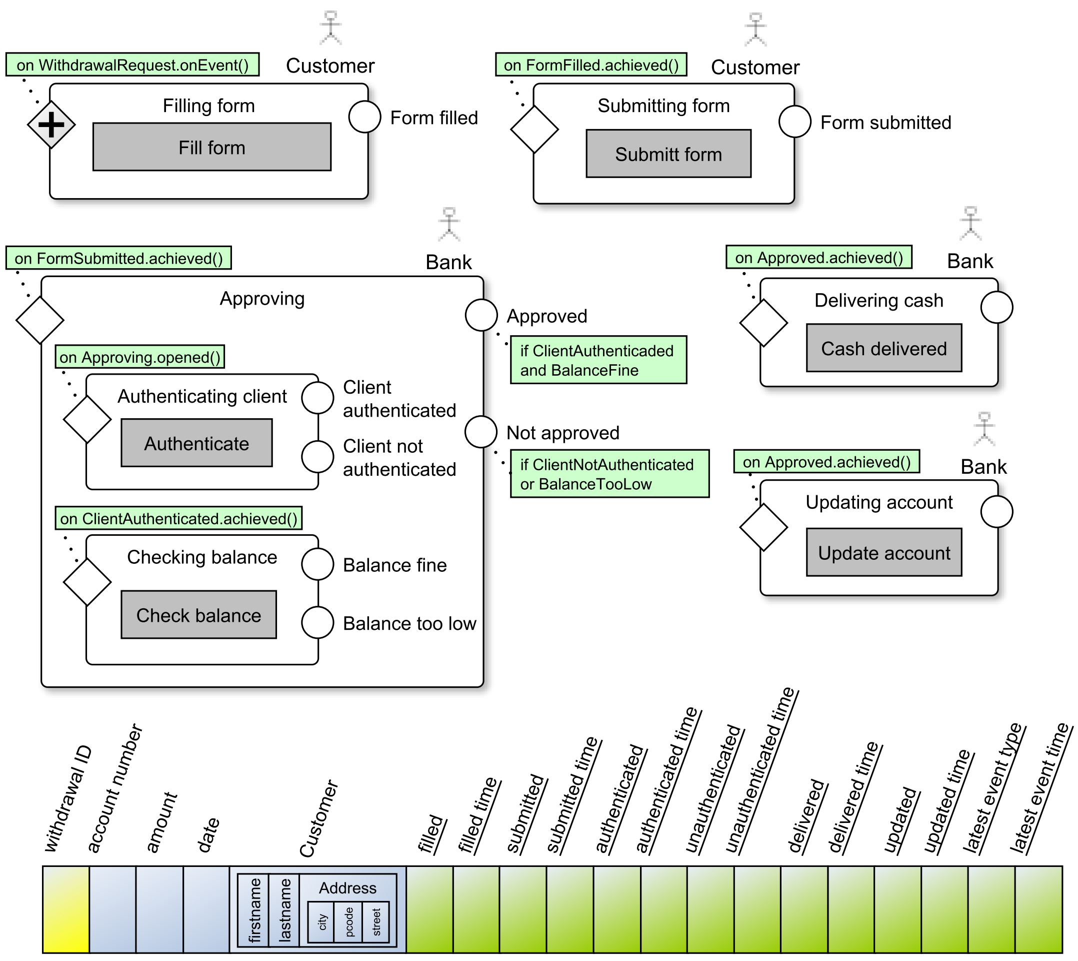


Figure 2.2: Withdrawal business process using GSM meta-model. Process has one artifact, *Withdrawal Request*. Top part shows artifact lifecycle model and bottom part shows artifact information model. Information model is further divided into two parts. Blue attributes correspond to actual data attributes, green attributes correspond to status attributes and record artifact progress.

2.2.1 Business artifacts

Business artifacts are core modeling constructs in the GSM meta-model. Every business artifact corresponds to a single business-relevant conceptual entity that is central point of some business operations and that evolves as it moves through these operations. Business artifact considers both data and functional aspects of business operations and as such it contains both information model and lifecycle model. Business artifact has an associated artifact type and unique identity.

2.2.2 Information model

Information model represents the data aspects of the business artifact. Information model contains all relevant information about an artifact instance that are necessary during the business operations and for their successful completion. This information can be further divided into two set of attributes.

Data attributes are pure business relevant data that represent information about business itself and how it is being affected by the artifact instance. Example of business data can be customer name, order price and so on. Data attributes can be simple values or complex values having the record or collection type. These record and collection attributes can be arbitrarily nested.

Status attributes contain information about which business relevant objectives assigned to the artifact are already achieved and what activities related to this artifact instance are currently in progress. These status attributes therefore record the progress in the artifact lifecycle.

2.2.3 Lifecycle model

Lifecycle model specifies how an artifact instance evolves as it moves through business operations. The core three components of lifecycle model are, as the meta-model name suggests, *stages*, *milestones* and *guards*. It specifies relationships between these components and uses ECA inspired rules to define how stages and milestones are changed over the time. Shortly speaking, stages represent activities, milestones are business objectives related to these activities and guards are condition on when stages become active. We will describe these three components in more detail.

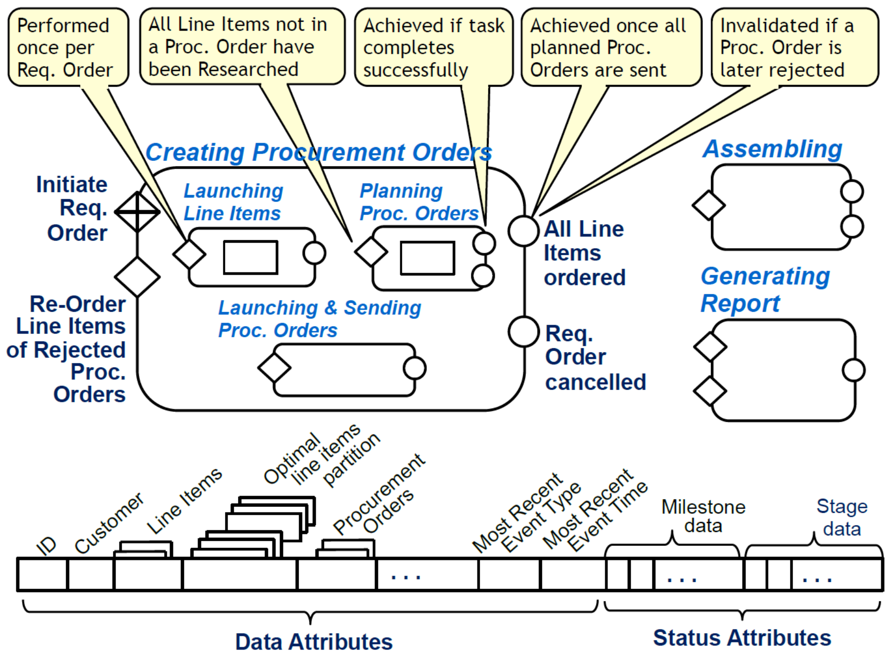

Performed once per Req. Order
All Line Items not in a Proc. Order have been Researched
Achieved if task completes successfully
Achieved once all planned Proc. Orders are sent
Invalidated if a Proc. Order is later rejected
Creating Procurement Orders
Assembling
Initiate Req. Order
Launching Line Items
Planning Proc. Orders
All Line Items ordered
Generating Report
Re-Order Line Items of Rejected Proc. Orders
Launching & Sending Proc. Orders
Req. Order cancelled
ID
Customer
Line Items
Optimal line items partition
Procurement Orders
Most Recent Event Type
Most Recent Event Time
Milestone data
Stage data
...
Data Attributes
Status Attributes

Figure 2.3: Example of information and lifecycle model (source [7])

An example of an artifact model is depicted in Figure 2.3. This figure shows both information model and lifecycle model. Information model is shown in the bottom of the figure and is divided into data attributes and status attributes. We can see that it has simple attributes and complex attributes as well. Lifecycle model is shown in the top of the figure. It contains stages, which are depicted as rounded-corner rectangles. As can be seen, stages can be nested. Milestones are shown as circles associated to single stage and guards are shown as diamonds also associated to single stage. Guard with diamond symbol with the cross inside is special kind of guard, bootstrapping guard and we will describe it later. Labels in the picture shows propositions for conditions on guards.

2.2.4 Milestones

Milestone corresponds to named business relevant objective, or goal, that can be achieved in respect to artifact instance. Milestone is always attached to exactly one stage, although the authors describe a variation in paper [13] where milestones are top level elements not associated to only single stage. Milestone can be either achieved or invalidated. This information is recorded in corresponded status attribute in the information model as a Boolean attribute that indicates if milestone is currently achieved. Information model also contains status attribute that holds the timestamp when the milestone last changed.

Milestone and stages are associated in their effect on each other. When milestone is achieved, the parent stage closes, because the purpose of the stage has been achieved. Similarly, when the stage opens, all associated milestones are invalidated, because if the activity is started, the goals are clearly not achieved. The expression that specifies the condition when milestone becomes achieved or invalidated are called sentries. Milestone conditions should be disjoint, meaning that only one can be true at any given time which mirrors the intuition that stage should achieve one of its objectives.

2.2.5 Stages

Stage corresponds to named activity that is related to business artifact instance. Stages support hierarchy and enable dividing the overall work into logically associated units. There are two types of stages, complex and atomic. Complex stages contain one or more children stages. Atomic stages cannot have children stages, but are placeholders for tasks, the units of business-relevant work, which we will describe later. Atomic stages can contain one or more tasks, depending on task types. Stages can be either active or inactive. This information is recorded in corresponded status attribute in the information model as a Boolean attribute that indicates if stage is currently active.

There is a relationship between stage and its children stage. When some stage is inactive, all children stages must be inactive as well. This can be memorized as simple rule *no activity in closed stages*. Similarly, when stage opens, all children stages can open as well if some of their guards become true. Each stage has at least one associated milestone that specifies under which conditions the stage goals are achieved and stage closes. Stage have also one or more associated guards.

When an atomic stage contains a task, the fact the stage closes does not necessarily mean that the reason for closing the stage is task termination [10]. Consider some atomic stage S with task T. Stage S opens and launches task T. Then some milestone of stage S that is not related to task T becomes achieved, so stage S closes and become inactive, although task T did not terminated. Closing the stage means that task T will be aborted, if possible, or alternatively its result will be ignored. This can happen for example if a manager decides that processing that contains this task should be canceled and sends event to the system that requests the cancellation [7]. Stage that can be canceled can have for example milestone *canceled*, that refers to cancellation event.

2.2.6 Guards

Guard specifies under which condition associated stage becomes active. When the guard becomes true, the associated stage is opened. Guard is always associated to single stage. Guards unlike milestones and stages do not have names, because as authors state in [7], "*business-level stakeholders do not typically refer to guards*". There is also one special kind of guard, called *bootstrapping guard*. This is a guard that specifies the condition when corresponding artifact instance is to be created. Each artifact must have exactly one bootstrapping guard that can be associated with some top level stage. The guard has a form of an expression, called sentry, or strictly speaking, guard *is* a sentry.

2.2.7 Sentries

Sentry is a condition expression that is used in guards and milestones. Sentry has a form $on\ \xi\ if\ \varphi$, where either 'on' part or 'if' part can be omitted. ξ is called event expression and refers to one external event or to one event representing change in artifact status attributes. φ is a condition that refers to artifact information model and does not involve any event occurrence expression. It is important to say, that sentry can refer not only to events and attributes of associated artifact instance, but also to any other artifact instance that is related to the associated artifact. Also, events can reference parameters of incoming events. The guards of a stages and milestones should be disjoint.

2.2.8 Tasks

Task corresponds to a unit of business-relevant work that is meaningful to the whole business process for two reasons. First, it represent measurable steps in the progress to achieve business relevant goals. Second, they enable the division of the business process into collection of identifiable services that can be be subject to administrative organization structures, IT infrastructures, customer-visible status, etc. [11, page 4]. Tasks are sometime called business services [11, page 4] to point out the close correspondence with the services used in Service Oriented Architectures and web services in general.

Tasks represent the actual work that is performed in connection to the GSM model. This work is performed by individual actors, which are either human or automated machines. The actual details of executing particular tasks are not covered by the model, because as authors state in paper [8], similarly to most BPM, case management and workflow systems, the GSM meta-model is intended to support the management of business-related activities, but not the details of their actual execution.

Tasks are invoked by artifact instances when the atomic stage containing the task opens. When a task is invoked, the artifact instance provides input data from its information model, and when the task terminates the task output data is written into the artifact instance information model.

Tasks may be essentially non-deterministic, but they may have associated post-conditions that captures a business policy. Human tasks can for example request user to fill necessary data or make approval and submit the information back to the system.

When tasks are executed they can make change to the data of one or more business artifacts. These changes should be transactional, so the task should have restricted exclusive access over involved artifacts for the duration of the service. [11, page 4]

The meta-model considers five categories of tasks: [8, 7]

- Assignment of attribute values
- Invocation of one-way or two-way external services
- Request to send response to an incoming two-way service call
- Request to send an event to one or more entity instances
- Request to create a new entity instance

Sending a message to another artifact instance is actually performed with the help of ASC that acts as an intermediary. The artifact sends the message to the ASC in a first step b1 and ASC then routes the message to the target artifact instance in a second step b2. In this case, it makes sense to have step b2 happen immediately after the step b2 [4, page 29].

Human tasks

We will take closer look to the human tasks. Tasks performed by humans are refered to as human services and are usually considered long-running, because they require human interaction. During their execution, the human service can have capability to receive other incoming events. These events may rise from different stages than task parent stage, because GSM meta-model allows parallelism between stages. Human services can also have an option to invoke save actions during the processing to save current progress for later continuation, but without closing the parent stage.

2.2.9 Events

Events are a used to trigger the sentries in guards and milestones. We consider two types of events:

External events External events are events sent from the environment to artifact instances. They are also called incoming events. These events can:

- Originate from outside of the system
- Originate from another artifact instance
- Correspond to the termination of a computational task

When some event arrives to the system, it is inserted to the queue of incoming events and later processed. External events are processed one at a time. Paper [13] studies possibility for parallel processing of these events, because serial processing can sometime be a bottleneck.

We can refer to incoming events within sentries in following ways:

- E.onEvent() is fired when incoming event E occurs.
- T.onComplete() is fired when currently active task T completes.

Internal events Internal events correspond to the changes in artifact status attributes. They are generated when milestones are achieved or invalidated and when stages are opened or closed. We refer to them within sentries in following ways:

- S.opened() is fired when a stage S opens and becomes active. This can happen if one of its guards become true and parent stage is active.
- S.closed() is fired when a stage S closes. This can happen when one of its milestones become achieved or when parent stage becomes inactive and closes.
- M.achieved() is fired when a milestone M becomes achieved and changes its value from false to true. This can happen when one of milestone sentries become true.
- M.invalidated() is fired when a milestone M becomes invalidated and changes its value from true to false. This can happen if milestone sentry is satisfied or when associated stage opens.

Events have a unique name and an associated artifact type. Incoming events also have a payload. Event occurrence can be relevant to more than one instance.

Outgoing events

Formal model also considers another type of events, called outgoing events. They correspond to task opening events that artifacts produce when some atomic stage opens and task inside that stage starts during process of incorporating one incoming event. Outgoing events are sent from artifact instances to the environment after the process of incorporating incoming event is completed. We can consider one outgoing event type for each task type, because outgoing event always signals that some task starts. They are also called generated events, because they are generated during incoming event processing.

2.2.10 Prerequisite-Antecedent-Consequent rules

Prerequisite-Antecedent-Consequent (PAC) rules are fundamental part of artifact lifecycle. They declaratively specify how artifact evolves over time. PAC rules are derived from explicitly defined sentries of guards and achieving and invalidating sentries of milestones, so the modeler does not write them explicitly and defines only individual sentries. These sentries specify the conditions that are used as a antecedent parts of corresponding PAC rules. The consequent and prerequisite parts of the rule are derived automatically from sentry type.

Guard rules Rules derived from guard sentries have action that causes opening of the associated stage. This corresponds to setting stage activity status attribute to true and set its timestamp of most recent change to current logical timestamp.

Milestone rules Rules derived from milestone achieving (resp. invalidating) sentries have action that causes achieving (resp. invalidating) of the associated milestone. This corresponds to setting milestone status attribute to true (resp. false) and set its timestamp of most recent change to current logical timestamp.

Rules that specify the lifecycle cannot be fired in an arbitrary order. They must be applied in an order based on a topological sort, which ensures that processing terminates and satisfies desirable principles. We require two principles to be followed:

Toggle once principle This states that through the incremental application of PAC rules, each status value attribute can change at most once during that construction. Note that if the incremental computation of a B-step did not satisfy Toggle once principle, then a given status attribute might change values inside the B-step, but those changes would not be visible from the starting and ending snapshots of the B-step [4].

Inertial principle This states that if a status attribute changes during a B-step, then there should be a justification for that change that is visible by examining only the starting and ending snapshots of the B-step [4].

We will present this in the example, borrowed from [10].

Example Consider figure 2.4. Suppose that stages S_1 and S_2 are open, m_1 and m_2 are false and $A = 20$. Now we start processing of a new incoming event e. Suppose that we go by the numbers in the figure:

1. Milestone m_1 becomes achieved, because incoming event is e.
2. Guard g_3 becomes satisfied, because m_1 is true and m_2 is false.
3. Stage S_3 becomes opened, because its guard became satisfied.
4. Milestone m_2 becomes achieved because incoming event is e and $A > 10$.

This results in incorrect state. We have m_1 and m_2 true, which means, that condition of g_3 should not be satisfied. But it is satisfied, which is a contradiction with what we would expected when looking only at the start and the end of the processing and considering only end values of attributes. This violates inertial principle.

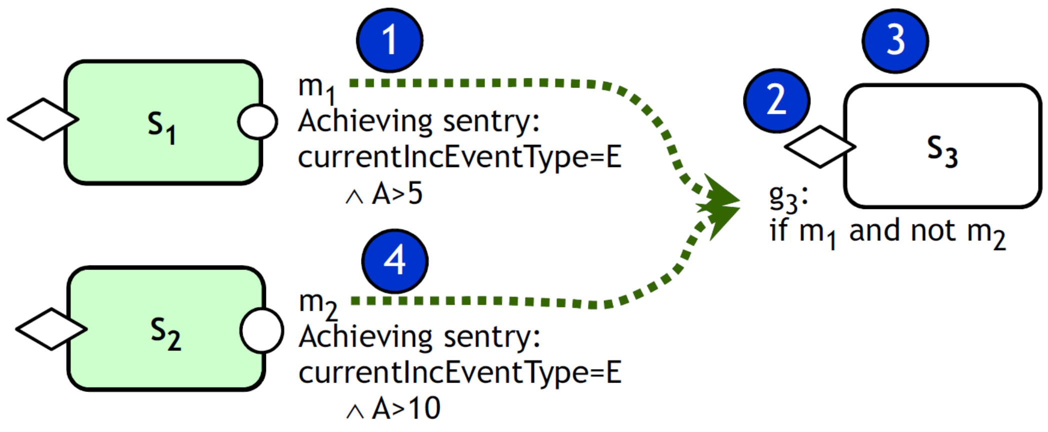


Figure 2.4: Example of information and lifecycle model (source [10])

2.2.11 Artifact Service Center

Artifact Service Center (ASC) is a container that manages related artifact types and their associated instances. In paper [7] they name this component *BEL Service Center* (BSC), where BEL stands for business entity with lifecycle. ASC has many important purposes.

Inter-artifact communication ASC enables communication between managed artifact instances. An artifact instance can send a message to another artifact instance by sending a message to the ASC, which then routes it to appropriate artifact instance.

External communication ASC ensures communication between managed artifact instances and external environment. This means primarily the ability to invoke two-way service calls and send one-way messages against the environment. This way ASC shields artifacts from concrete technologies used for implementation of service calls. ASC can also receive incoming events and route them to appropriate artifact instances for processing. Environment can also request ASC to create new artifact instances.

Human performers ASC supports interaction between human performers and managed artifact instances. This interaction can be provided by user interface or by exposing appropriate service operations. Human performers can send events to the system that are routed to target artifact instances, they can see their status of artifact instances and execute human services.

Ad-hoc queries ASC supports ad-hoc queries against the data store that holds the artifact instances. Human performers can access all information stored in artifact instances (within their privileges).

As described in [7], in practical implementations ASC acts as an SOA-service that interacts with external environment exclusively using service calls. These can be specified both using REST and WSDL.

2.3 Formal description

This section provides formal description of the Guard-Stage-Milestone meta-model. The main source is [4], because it provides detailed formal description from the authors of the GSM meta-model and presents the version of meta-model that supports multiple artifact types, multiple artifact instances and structured attribute, so it can focus on artifact interactions. Another detailed source for formal description of the GSM meta-model is paper [10], which defines restricted version of the GSM meta-model that considers only one artifact type, one artifact instance and simple attribute types. This restriction makes the presentation simpler to understand and provides good introduction to formal foundation. As authors state, this restriction does not fundamentally compromise the applicability of the results. Formal definitions in this section are always cited from [4].

2.3.1 Artifact types and GSM model

Before going further, we must first introduce supported data types and their domains. GSM meta-model supports primitive types, record types and collections of both primitive and record types. Record types are complex structures composed from primitive types and possibly from other records that can be arbitrarily nested. Primitive types include arbitrary scalar types, for example Boolean, integer, real and they also include two specific types: **EVENT** that ranges over all incoming event types and **TIMESTAMP** that ranges over logical timestamps. GSM meta-model also considers for each artifact type set $\mathbf{ID_R}$ of identifiers for artifact instances of this artifact type. All these domains are extended with the null value, denoted by symbol $\perp$. We will further use the set **TYPES** of all *permitted types* for artifact attributes, which contains all previously described types.

Definition An artifact type has the form $(R, x, Att, Typ, Stg, Mst, Lifecycle)$ where the following hold:

- R is the name of the artifact type and it is often used to refer to an artifact type (R, x, Att, Typ, Stg, Mst, $Lifecycle$) itself.
- x is the context variable of R that ranges over the IDs of instances of R. This variable is used in the logical formulas in $Lifecycle$.
- Att is the set of attributes of this type. This set is further spitted into the set of attributes Att_{data} and set of status attributes Att_{status}
- Typ is the function $Typ : Att \rightarrow \mathbf{TYPES}$ that assigns a type to each data attribute.
- Stg is the set of stage names.
- Mst is the set of milestone names.
- $Lifecycle$ is the artifact lifecycle model.

We use $\mathbf{ID_R}$ to denote the type of IDs of artifact instances of R. There are further restrictions for an artifact type:

- The sets *Att*, *Stg* and Mst are pairwise disjoint.
- The set *Att* must include special attribute ID that holds unique, immutable, not null identifier of the artifact instance.
- *Att* must include two attributes to hold information about most recent incoming event that affected this artifact instance.
  - Attribute *mostRecentEventType* of type EVENT holds the type of this event.
  - Attribute *mostRecentEventTime* of type TIMESTAMP holds the most recent logical timestamp in which the event processing occurred.
- The set Att_{status} must include two attributes for each milestone $m \in \mathbf{Mst}$ that hold information about status of the milestone m.
  - Attribute m of type Boolean indicates whether milestone m is currently true or false.
  - Attribute $m_{mostRecentUpdate}$ of type TIMESTAMP holds the most recent logical timestamp in which the value of milestone m changed.
- The set Att_{status} must include two attributes for each stage $s \in \mathbf{Stg}$ that hold information about activity of the stage s.
  - Attribute s of type Boolean indicates whether stage s is currently active or inactive
  - Attribute $m_{mostRecentUpdate}$ of type TIMESTAMP holds the most recent logical timestamp in which the activity of stage s changed.

We already described concepts like guards, milestones and stages in informal description. Now we describe a lifecycle model of an artifact.

Definition : Let $(R, x, Att, Typ, Stg, Mst, Lifecycle)$ be an artifact type. The lifecycle model L of R has structure $(Substages, Task, Owns, Guards, Ach, Inv)$ and satisfies the following properties.

- *Substages* is a function from *Stg* to finite subsets of *Stg*, where the relation $(S, S')|S' \in Substages(S)$ creates a forest. The roots of this forest are called top-level stages, and the leaves are called atomic stages. A non-leaf node is called a composite stage.
- $Task$ is a function from the atomic stages in *Stg* to tasks
- *Owns* is a function from *Stg* to finite, non-empty subsets of Mst, such that $Owns(S) \cap Owns(S') = \emptyset$ for $S \neq S'$. A stage S owns a milestone m if $m \in Owns(S)$.
- *Guards* is a function from *Stg* to finite, non-empty sets of sentries. For $S \in Stg$, an element of $Guards(S)$ is called a guard for S.

- Ach is a function from Mst to finite, non-empty sets of sentries. For milestone m, each element of $Ach(m)$ is called an achieving sentry of m.
- Inv is a function from Mst to finite sets of sentries. For milestone m, each element of $Inv(m)$ is called an invalidating sentry of m.

If $S \in Substages(S')$, then S is a child of S' and S' is the parent of S.

Intuitively, a GSM model is a set of all artifact types. We describe this formally.

Definition A GSM model is a set Γ of artifact types with form (R_i, x_i, Att_i, Typ_i, Stg_i, Mst_i, $Lifecycle_i$), i $\in [1..n]$, that satisfies the following:

- **Distinct type names**: The artifact type names R_i are pairwise distinct.
- **No dangling type references**: If an artifact type ID_R is used in the artifact type R_i for some $i \in [1..n]$, then $R = R_j$ for some (possibly distinct) $j \in [1..n]$.

2.3.2 Artifact snapshots and pre-snapshots

Artifact snapshots intuitively relate to concrete state of single artifact instance. These states can change as a result of incorporating some external event. Therefore the GSM meta-model defines the concept of snapshots to describe the states before and after the event processing. These states must satisfy some important invariants, that we will describe later. On the other hand, incorporating single incoming event is always composed from multiple internal steps where each step results in new artifact state. However, it is not guaranteed that these invariants are always satisfied in these intermediate states during the incoming event processing. This is why the concept of pre-snapshot is defined as well. We now formally define the pre-snapshot.

Definition Let Γ be a GSM model, and $(R, x, Att, Typ, Stg, Mst, Lifecycle)$ be an artifact type in Γ. In this context, an artifact instance pre-snapshot of type R is an assignment σ from Att to values, such that for each $A \in Att$, $\sigma(A)$ has type $Typ(A)$. Note that $\sigma(A)$ may be $\perp$ except for when $A = ID$.

We will use $\rho = \sigma(ID)$. Now, when A is an attribute of R, then $\rho.A$ refers to the value $\sigma(A)$. We call this the path expression and it can be chained, if A is of some artifact type as well.

Core aspects of relationship between stages and milestones are captured in the following three GSM Invariants, which apply to artifact instance pre-snapshots. Invariant GSM-3 is assumed to be maintained by properties of the milestones themselves.

GSM-1: Milestones are false for active stage If stage S owns milestone m and if $\rho.active_S = true$, then $\rho.m = false$.

GSM-2: No activity in closed stage If stage S has substage S' and $\rho.active_S = false$, then $\rho.active_{S'} = false$.

GSM-3: Disjoint milestones If stage S owns distinct milestones m and m' and $\rho.m = true$, then $\rho.m' = false$.

Definition An artifact instance snapshot of type R is an instance pre-snapshot σ of type R that satisfies the three GSM Invariants.

Artifact instance snapshots represent artifact state in particular time. Sequence of these snapshots compose an artifact instance. This instance corresponds to a single conceptual entity that evolves as it moves through some business processes.

Definition An artifact instance of R is a sequence δ_1, ... , δ_n of snapshots of type R such that $\delta_1(ID) = \delta_2(ID) = ... = \delta_n(ID)$.

We now define a pre-snapshot of an artifact model. Intuitively, this is a set of artifact pre-snapshots.

Definition A pre-snapshot of Γ is an assignment Σ that maps each type R of Γ to a set $\Sigma(R)$ of pre-snapshots of type R, and that satisfies the following structural properties:

- **Distinct ID's**: If δ and δ' are distinct artifact instance pre-snapshots occurring in the image of Σ, then $\delta(ID) \neq \delta'(ID)$.
- **No dangling references**: If an artifact instance ID ρ of type ID_R occurs in the value of a non-ID attribute of some pre-snapshot in $\Sigma(R')$ for some R' in Γ, then there is a pre-snapshot δ in $\Gamma(R)$ such that $\delta(ID) = \rho$.

Definition Snapshot of Γ is a pre-snapshot Σ of Γ such that each artifact instance pre-snapshot in the image of Σ is an instance snapshot.

2.3.3 Processing an incoming event

When an incoming event occurs, it must be incorporated into current snapshot. This is called GSM business step, or shortly B-step. B-step is denoted (Γ, e, t, $\Gamma\prime$, *Gen*), where Γ is a previous snapshot, e is an incoming event, t is a logical timestamp greater than all logical timestamps occurring in Γ, $\Gamma\prime$ is next snapshot and *Gen* is a set of generated outgoing event occurrences, which usually correspond to task opening events.

B-step is composed from two phases. In the first phase, event payload is incorporated into information model of affected (or created) instances and attributes *mostRecentEventType* and *mostRecentEventTime* are updated. This is called an immediate effect. Immediate effect does not change any status attribute and it does not fire sentries. It is denoted ImmEffect(Γ, e, t). In the second phase, event is incorporated into guards, milestone achieving sentries and milestone invalidating sentries. This is done by incremental application of PAC rules, changing in sequential steps the initial pre-snapshot Γ_1, that resulted from immediate effect, into final pre-snapshot $\Gamma_n = \Gamma\prime$, until no PAC rule can be applied to Γ_n. These computational steps are called micro-steps and their ordering is very important, as was described in section 2.2.10. At the termination of a B-step, a set of generated events *Gen* is sent to the environment. An entire B-step is always considered to happen in single logical time t.

Rule	Description	Prerequisite	Antecedent	Consequent
PAC1	**Guard**: if $on\ E(x)\ if\ \varphi(x)$ is a guard of S. Include term $x.active_{S\prime}$ if $S\prime$ is parent of S.	$\neg x.active_S$	$on\ E(x)\ if\ \varphi(x)\ \wedge\ x.active_{S\prime}$	$+x.active_S$
PAC2	**Milestone achieving sentry**: If S has milestone m and $on\ E(x)\ if\ \varphi(x)$ is an achieving sentry for m.	$x.active_S$	$on\ E(x)\ if\ \varphi(x)$	$+x.m$
PAC3	**Milestone invalidating sentry**: If S has milestone m and $on\ E(x)\ if\ \varphi(x)$ is an invalidating sentry for m.	$x.m$	$on\ E(x)\ if\ \varphi(x)$	$-x.m$
PAC4	**Guard invalidating milestone**: If S has milestone m and has guard $on\ E(x)\ if\ \varphi(x)$ of S, where $E(x)$ is not -$x.m$, and where $\neg x.m$ does not occur as a top-level conjunct in $\varphi(x)$. Include term $x.active_{S\prime}$ if $S\prime$ is parent of S.	$x.m$	$on\ E(x)\ if\ \varphi(x)\ \wedge\ x.active_S$	$-x.m$
PAC5	**Achieving milestone closes stage**: If S has milestone m.	$x.active_S$	$on\ +x.m$	$-x.active_S$
PAC6	**Closing stage closes child stage**: If S is child stage of $S\prime$.	$x.active_S$	$on\ -x.active_{S\prime}$	$-x.active_S$

Figure 2.5: Rules categories

2.3.4 Prerequisite-Antecedent-Consequent rules

We described PAC rules informally in section 2.2.10. Formally, each rule has prerequisite, antecedent, and consequent part. There are six rule categories. Three categories are explicit rules, derived from sentries from lifecycle models defined by a designer. Other three categories are invariant preserving rules, these focus on preserving the invariants GSM-1 and GSM-2. Rules evaluation must be done in precise topological order. Formal description for rules ordering algorithm is presented in [4].

3. Active XML framework

This section provides introduction and description of Active XML (AXML). First we introduce it and provide motivation for this technology. Then we describe its main aspects in principles. Lastly, we briefly describe its prototypical implementation.

3.1 Introduction and motivation

Active XML is a framework for distributed data and service integration [14]. It is centered around AXML documents, which are valid XML documents that contain not only static XML data, as ordinary XML documents, but also dynamic parts, which can incorporate new data into the document or provide updates for current one. Dynamic parts are realized using web service calls. They are specified declaratively using special XML elements that are embedded in the document. They can be activated and once this happen, the call to respective web service is invoked and its result is incorporated into the document. Using this approach the AXML document can evolve over time and integrated data from various sources.

We will now describe the motivation for Active XML. Nowadays, as more and more applications move on the web, data integration must deal with many problems, mostly interoperability and heterogeneity of different sources [14], because these sources may be often implemented in different programming languages and different platforms. Also, there is a problem with very large scale of the web. Fortunately, there are already technologies to address each of these problems, namely XML, standardized web services and peer to peer architectures.

XML XML addresses heterogeneity problem. It is a semi-structured, self describing data exchange format promoted by W3C [18] and is highly portable, extensible and widely used. XML is supported by many programming languages and there are many tools for its processing.

Web services Standardized web services based on SOAP [17] and WSDL [16] addresses interoperability problem. SOAP is a protocol specification for exchanging structured information that relies on XML . WSDL is an XML based language for describing network services as a set of endpoints operating on messages.

Peer to peer architecture Peer to peer architecture is a decentralized architecture that is based on independent and autonomous nodes. Each peer can acts both as a server and a client.

Authors believe that combining peer to peer architecture with XML and standardized web services in one framework provides a proper ground for data integration on the web [14].

3.2 AXML description and principles

We will now describe the main principles of AXML. AXML framework is based on AXML documents that can be distributed among several peers, called AXML peers. Figure 3.1 shows distribution and communication among peers. Every AXML peer has a repository of AXML documents and defines a set of AXML web services. Framework distinguishes true AXML peers, which are peers that contain AXML documents and AXML engine and classic, non AXML peers. AXML peer communicates with external world exclusively using web services.

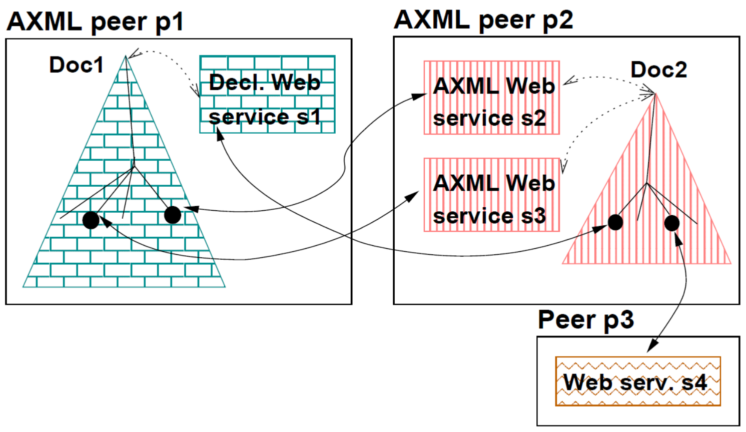


Figure 3.1: AXML data distribution (source [19])

AXML peer can act both as a client and a server. By invoking web service calls embedded in its documents and integrating results of these calls into the documents, peer acts as a client. Peers can also define own web services that query or update its documents. By defining and exposing these web services, accepting requests, executing them and sending responses, peer acts as a server.

Communication does not have to occur only among true AXML peers. As figure 3.1 suggests with peer p_3, AXML peer can invoke web services on any peer, not only on true AXML peer. Similarly, web services exposed by true AXML peer can be invoked from any peer, not only from true AXML peer.

3.2.1 AXML document

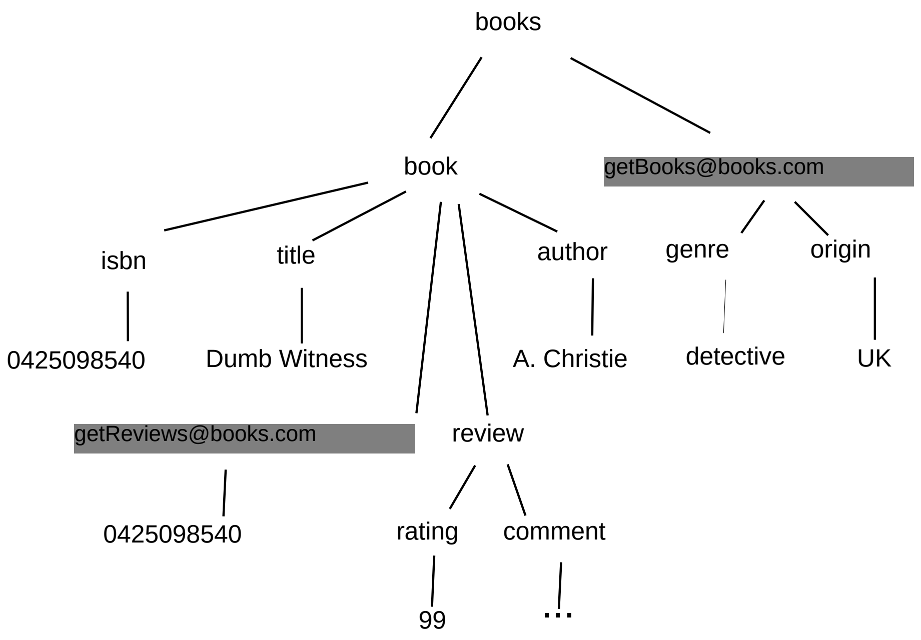
We said in the introduction, that AXML document is an ordinary XML document where some data are specified *explicitly*, exactly the same way as in classic XML documents and some data are given *intentionally*, using embedded service calls and denoted using special, but still classic valid XML elements. We will now present example of AXML document and describe embedded calls.

```
<books>
  <book>
    <isbn>25098540</isbn>
    <title>Dumb Witness</title>
    <sc service="getReviews@books.com">
      25098540
    </sc>
    <review>
      <rating>99</rating>
      <comment>...</comment>
    </review>
    <author>A.Christie</author>
  </book>
  <sc service="getBook@books.com">
    <genre>detective</genre>
      <origin>UK</origin>
  </sc>
</books>
```

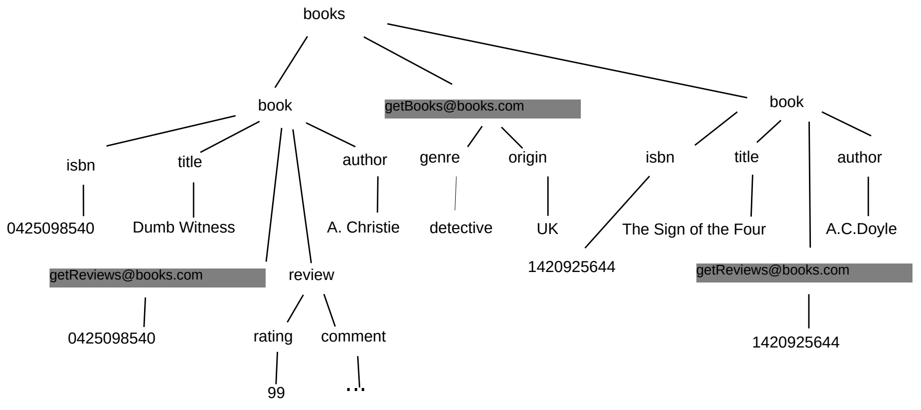
(a) Before activation, code form

```
<books>
  <book>
    <isbn>25098540</isbn>
    <title>Dumb Witness</title>
    <sc service="getReviews@books.com">
         1420925644
    </sc>
    <review>
      <rating>99</rating>
      <comment>...</comment>
    </review>
    <author>A.Christie</author>
  </book>
  <sc service="getBook@books.com">
    <genre>detective</genre>
      <origin>UK</origin>
  </sc>
  <book>
    <isbn>1420925644</isbn>
    <title>The Sign of the Four</title>
    <sc service="getReviews@books.com">
      1420925644
    </sc>
    <author>A.C.Doyle</author>
  </book>
</books>
```

(b) After activation, code form



(c) Before activation, tree form



(d) After activation, tree form

Figure 3.2: AXML document example both as a document and a tree

Figure 3.2 shows an example AXML document in its code and tree representations. Left side represents document before activation and right side represents the same document after activation of service call *getBook@books.com*. XML code contains *sc* elements to denote embedded service calls. Attribute *service* of *sc* element provides information necessary to invoke the service, namely its name and endpoint. Child nodes of *sc* element represent arguments for the service call. We note here that the code uses simplified syntax for expressing embedded service calls to make the demonstration easier.

Our example contains list of books, where each book contains unique ISBN, title, author and list of reader reviews. Some books are given explicitly, concretely book "Dumb Witness", while other books can be obtained from service *getBooks@books.com* that returns most popular books for current day, according to genre and origin as specified by service input parameters. See left side of our example. We declare that we want most popular book with genre "detective story" written by author from "United Kingdom". Right side shows result document after service call terminates. AXML document was enhanced with new book element for the book "The Sign of the Four". New element is inserted as a sibling node of the service call element. Note that original service call element is *not* removed. This is an important feature, since we can later re-activate the call and obtain up-to-date results.

We described what happens after activation of embedded service call. But we did not say when this activation actually happens. As we will see, embedded calls have many features including specification of when to activate the calls, how to integrate their results, how to deal with parameters and many others. We will describe them now in closer look.

3.2.2 Service call activation

Each embedded call can include specification on when the respective call should be invoked. This is called *frequency.* Frequency can be for example every day, only when needed, after some interval and so on. Embedded call can also specify for how long the response of the service call remains valid. This is called validity. Validity can be for example zero duration, some positive duration or infinity. As authors state in [14], combination of frequence and validity of calls allows to capture different styles of data integration, like warehousing, mediation and their combination. For example setting validity to zero means that client will use response data only to serve one specific request that needs the data and after request is served, it will automatically remove them. This is like mediator style, where data are not stored but obtained when needed [20]. On the other way, setting validity to infinity means that client will archive call responses forever and will not automatically remove them. Only application itself could control the data deletion.

We can quickly illustrate this in our example 3.2. We could specify frequency for call *getBook@books.com* to "every day" and validity to seven days, which means storing a history of most popular books for each of last seven days. Client caches these books and each day it removes the oldest book element.

3.2.3 Merging service results

We said that when client invokes embedded service call, its result are integrated into the document when the service call returns. There are many options how to integrate these results. This integration actually rises two questions: *where* to put the result and *how* to deal with previous service call results, if any. Simplest option is to put result data as a sibling of *sc* element that corresponds to the invoked embedded call. This can be either before calling *sc* element, or after it. Result can either be appended behind previous results, or replace the previous result, if any. Authors however investigate an option to integrate data using ID-based data fusion [15]. This option relies on fact that both DTD and XML Schema allow to promote some attributes to be unique identifiers for their elements. Then, if some service call result contains element with ID that exists in the document, then the result element can be merged with such element.

3.2.4 Continuous services

Sometime it is desirable when client does to receive only single answer, but entire stream of answers. This is common technique in subscription systems, where subscription system pushes new data to the subscriber. This is implemented in AXML as a *continuous service*. When client activates such continuous call, first a subscription is a made at a service provider. Service provider then subsequently returns individual results. These results are integrated into the documents of the client in an analogous way to normal service call.

3.2.5 XQuery parameters

We said that parameters are expressed as child elements of the service call. In our example in figure 3.2 we have embedded call to *getReviews@books.com*, that gets reviews for the book based on its parameter that corresponds to unique book ISBN. See that this ISBN value is the same as value of ISBN element of the parent book element. AXML allows us to specify input parameters as XQuery expressions. With this approach, when service call is activated, its parameters are first populated by evaluating the specified query. Since the query can return multiple results for each of the service call parameters, the service call is invoked for each combination of parameters evaluation. This can greatly reduce the amount of embedded *sc* elements, for example if we would like to invoke the same web service for each book element in the document.

In our example 3.2 we could use this XQuery parameters and write instead of static ISBN the query *getReviews@books.com*(*../isbn/text*()). This is very useful, because we do not have to explicitly write the ISBN parameter, which results in less typing, reuse and less errors, because change in ISBN value is automatically propagated to *getReviews* service call.

3.2.6 Intentional parameters and results

XQuery parameters are powerful way to define input parameters with respect to peers documents. But we can also use input parameters that contain embedded service calls. Moreover, embedded service calls can be included in service results as well. This enables a declarative service composition. These parameters and results are called *intentional*. Note that service composition can be recursive, because embedded service calls in parameters can return another service calls. The question around intensional parameters is *when* to invoke the service calls embedded in parameters. There are intuitively two alternatives.

First alternative is that client invokes them first, prior to invoking the parent service call, incorporates their results into the parameters and when input parameters contain no embedded service calls, client finally invokes the parent service call.

Second alternative is that server invokes these parameters. Client ignores them and invokes the parent service call without dealing with intentional parameters, and so server receives the service call with parameters containing embedded calls. It is now up to server to invoke these service calls and incorporate their results into the parameters. After that, server processes the call, which now corresponds to local computation. This variant allows to **push computation on the server**.

Similar case is with intentional results. When server processes service call and service result contains embedded service calls, it can either invoke them before answering to client or leave this up to client. When server leaves the invocation up to the client, the client, strictly speaking, does not get the actual information he requested, but he get the way how to obtain it which means that he can reuse it later to get more results or update his current results. This variant allows to **push computation on the client**. Authors illustrate this with a convenient quotation in paper [20]:

> *"Give a man a fish and you feed him for a day. Teach a man to fish and you feed him for a lifetime".*

We can see this scenario in our example 3.2. Client invoked the service *getBook@books.com* and he got in result not only static information about a book, but also an embedded service call that he can use to obtain reviews for this book. Clearly, list of all reviews for one book can be quite large, so it is up to the client whether he invokes the call to get reviews or not. This prevents sending potentially unnecessary data to the client and saves traffic. Client can also use the call to update its current reviews, because embedded call can be re-invoked many times.

This service composition brings potential problems with security, as authors state in many papers. For example, intentional parameters could contain embedded call to some malicious web service and this way client could push this malicious service call to the server. Similarly, server could return in intentional results a call to malicious server and push this malicious call to the client.

To conclude this section, we note that although AXML supports this service composition, authors state in [15, page 5] that AXML framework is not primarily a framework for service composition, but a framework for data integration using web services. The focus is on data, not on workflow and process-oriented techniques.

3.2.7 Lazy query evaluation

We described in section 3.2.2 how frequency can specify when an embedded service call should be called. The frequency however does not have to mean that the call should be invoked whenever the frequency says so. AXML actually allows to specify a mode of an embedded service call, that is either *immediate* or *lazy*. In immediate mode, the call is activated when it expires according to frequency. In lazy mode, the service call is invoked only when its result are really needed, this can happen for example if someone queries the AXML document and service call results should be contained in this document. This lazy mode prevents from invoking unnecessary service calls.

We can illustrate lazy mode in our example 3.2, picture d. Suppose there is a query $/books/book[author =' A.C.Doyle']/reviews$. This query surely does not involve first book element, so service call *getReviews@books.com* for this book would not need to be invoked, event if its frequency was already expired. On the other hand, second book element is included in result of this query, so its service call to get reviews would have to be invoked, if its frequency was expired.

Lazy query evaluation is researched in much closer look in paper [19]. Authors introduce methods based on query analysis and rewriting that find the sequence of service call invocations that are needed to evaluate given query for AXML data.

3.3 AXML prototypical implementation

Currently AXML has two major versions of prototypical implementation. First version used Apache Axis for web services, Apache Tomcat 4.0 and authors own, X-OQL query processor, because as authors state in [21], *"no XQuery processor existed when the project was started"*. More recent version 2.0 was released in November 2007. This version replaces previous one. We will now focus only on AXML 2, because in our thesis we use this most recent version.

3.3.1 Software components

Active XML distribution Apache Axis 2 for web services, Apache Tomcat 5.5 and eXist XML database [22]. This means that this time new version uses XQuery processor contained in eXist distribution.

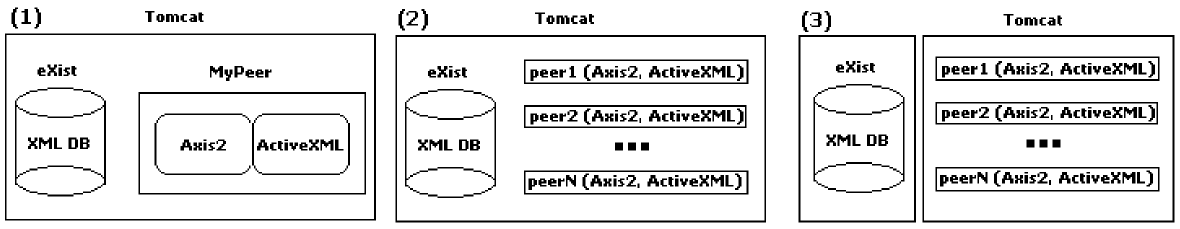


Figure 3.3: Software components in AXML (source [22])

Figure 3.3 shows software components in AXML and their possible configuration. Active XML engine and Axis 2 compose a web application and this application is deployed in the Tomcat web server. Web application uses eXist database and this database has for each peer one separate collection of documents that represents peers repository. The most expensive configuration is shown in part 1, where every peer has its own web application and own database.

3.3.2 AXML peer

Each peer has an AXML engine, which provides a database access and manages and evaluates AXML documents that are stored in the eXist database.

As we said in section 3.2, each peer acts as a client. This is assured by providing web interface for accessing, evaluating and optimizing AXML documents [22]. Peer also acts as a server by defining and exposing web services. It has these services:

Algebra web services Algebra web services enable distributed data management. These services are used by optimax, a module that optimizes and evaluates AXML documents. *ReceiveOperator* is responsible for preparing results from invoked web services, since service call responses are redirected to this service. *SendOperator* can send data to specified address by calling respective *ReceiveOperator*. *NewNodeOperator* allows to install new AXML documents into peers repository, or append new elements to specified address in the document. If new document is created, it is immediately evaluated.

MaterializationService Materialization service allows evaluation of AXML documents. Method *evaluate* evaluates entire document in a depth-first manner. So when a service call has intentional parameters, as described in subsection 3.2.6, these parameters are evaluated first. The ordering of execution can be explicitly overridden. Method *evaluateNode* is similar to *evaluate*, but evaluation starts in specified node. Method *activate* will activate a specified service call but will not automatically activate intentional parameters, unless their activation is explicitly requested using activation order constraints.

GenericQueryService Generic query service is used to execute input query over the database using method *executeGenericQuery*. Method *applyQuery* can be used to apply a query defined in another AXML document and provides a way to push data into another document [22]. Method *continuousQuery* gets a query and parameter streams and executes the input query over the database with each item from the stream.

DummyStreamService Dummy stream service only streams back a result of an input query.

ContinuousService Continuous web service can be used to call other web services in a continuous way. It receives a stream as a parameter and for each item from a stream, the service calls the specified web service and returns the result.

3.3.3 Known limitations

Current AXML distribution does not contain all the theoretical features described in the section 3.2. Most importantly, the activation frequency and validity is not implemented, so there is only one option, which is call the web service whenever needed. Also lazy query evaluation for AXML document is not implemented, which is confirmed from Active XML mail conference archive [23]. Also, more complex merging strategy based on ID-fusion is not implemented.

3.4 AXML Artifact model

There already exists an approach that connects business artifacts and Active XML. It is called AXML Artifact Model and is described in paper [32]. Authors propose an approach, where *"artifacts are XML documents that evolve in time due to interactions with their environment, it means human users or web services"* [32].

AXML Artifact model is based rather on earlier business artifact-centric modeling approaches where lifecycles are modeled using finite state machines and it does not seem to be very convenient for GSM model that supports stage hierarchy and parallelism within single artifact instance, as authors of GSM meta-model state in [4]: *"The AXML Artifact model supports a declarative form of artifacts using Active XML as a basis. The approach takes advantage of the hierarchical nature of the XML data representation used in Active XML. In contrast, GSM uses milestones and hierarchical stages that are guided by business considerations."*

```
<plant artID="plant02">
...
<webOrder artID="wo3">
   <client>
      <name>Sue Leroux</name>
      <address> ... </address>
   </client>
   <order> ... </order>
   <order> ... </order>
   <creditApproval artID="wo3-ca">
   ...
   </creditApproval>
   <fun funID="?warehouseOrder" />
   <fun funID="?comm" />
</webOrder >
...
</plant>
```

Figure 3.4: AXML Artifact example

In AXML Artifact model, the state of an artifact is an AXML document. Figure 3.4 show an example of such document, it contains a *webOrder* artifact. This artifact has a subartifact *creditApproval*. Artifact has function nodes *?warehouseOrder* and *?comm*, which will be later activated in order to create the warehouseOrder and communication sub-artifacts that will then work concurrently. AXML Artifact model uses function guards to control the call. There are four types of guards:

- Call guard that controls call activation
- Argument query that computes call arguments
- Return guard that controls call return
- Result query that computes call result

These guards are implemented using *continousCall* and *continousQuery* [33]. We can ilustrate usage of call guard using a continuousCall embedded in a document. ContinuousCall specifies a service address that will be called and a query that corresponds to call condition. This condition query can return method name with parameters for every data that satisfy the condition. Then, when continuousCall is activated, it evaluates the query condition and calls returned method with returned data on specified service for each result of the condition query.

```
<items>
   <item>
      <name>Monitor</name>
      <ordered />
   </item>
   <item>
      <name>Keyboard</name>
   </item>
   <item>
      <name>Mouse</name>
   </item>
</items>
<sc method="continuousCall" id="MakeOrderCall">
   <endpoint>OrderService</endpoint>
   <data>
      for $item in /items/item where not($item/ordered)
      return
      <MakeOrder>{$item}</MakeOrder>
   </data>
</sc>
```

Figure 3.5: AXML Artifact continuous call

Figure 3.5 shows an example of continuous call. When sc node with id “MakeOrderCall” is activated, it evaluates the query and for each *item* element that does not have *ordered* child element, it calls method *MakeOrder* at service *OrderService*. In our example, it will call the method *MakeOrder* twice, once for item *Mouse* and once for item *Keyboard*.

4. Conceptual model for XML

Conceptual model for XML, so-called XSem, is an approach to modeling XML schemas based on principles of Model-Driven-Architecture (MDA). It was introduced by Martin Necasky in [27] and later in [28]. XSem connection with MDA is its characteristic feature that distinguishes it from other approaches to XML data modeling and that implicates many useful qualities. Before we describe it further, we will briefly describe the MDA in general.

4.1 Model-Driven-Architecture

Model-Driven-Architecture is a software design approach to developing software systems defined by the Object Management Group [25]. It was introduced to increase the efficiency of software development. It is a based on an idea that system under construction should be developed in a *top-down* approach, which means that the system is firstly described in an abstract high-level representation that enables to concentrate on conceptual aspects of the application domain and relationships between them. This representation is called *Platform-Independent-Model* (PIM for short), because it is independent on target platform and implementation language which means that it can be understood by several stakeholders, most importantly domain experts that can analyze and design the model using their knowledge of the problem domain and analysis of user requirements. This is very important, because domain experts can have strong knowledge of the problem domain, but do not need to know other areas of software construction, like programming, testing etc and can focus on integrating user requirements into the model. Since the model is rather abstract, also business level stakeholders can intentionally read and use to the model because model provides much better understanding then lower level technologies. This helps to clarify the understanding between stakeholders involved. Also changes in the requirements can be represented by changes in the model. Importantly, models can be validated before the system implementation starts, or during the construction when changes are involved.

Model is further used during system implementation, where implemented system respects the concepts from the model or it can be even, at least partially, generated from the model. Earlier approaches, that first appeared in 1980s under name Model-Driven-Development were based on this idea [24]. Simply put, they expected that in a first step we draw a diagram that represents an aspect of the system under construction and in a second step we use that diagram directly to help generate or implement that system. This is also called *forward engineering.* Back then, there were great expectation, but the success was unfortunately not so great. The models were not a particularly pleasant or convenient expression of the problem at hand. Also there was a lot of generated code, and mistakes in the system would tend to be corrected by fixing the generated code, which however breaks the link between model and solution and heavily decrease the gain of the approach. This leads to following observations [24].

- Model must be readily apparent to people familiar with the domain. The model language must be designed to fit the intended purpose.

- Generation process must be efficient, and it must be straightforward to remedy errors in it and to customize it.
- Developers must see that model will help them to get their work done and adopt it.

MDA exploits three concrete models in the software development process. A top level model is a computation independent model (CIM), that focuses on the environment of a system and on requirements. It can be transformed to the PIM. In this thesis, we do not work with CIM. We already described the PIM. PIM captures conceptual aspects of the application domain. This model however does not say anything about how the system will be represented in target platform. This is where *Platform-Specific-Model* (PSM) comes into play. PSM is derived from PIM and defines how the data is represented in a target data model. Because it is platform specific and it must capture concrete implementation details, there must be a special PSM for each target data model. We can see a PSM diagram as a mapping between conceptual diagram and target data model schema. This derivation can be either automatic, semi-automatic or manual. Unlike PIM, PSM will only make sense to a developer who has knowledge about the specific platform.

4.2 Other approaches to XML data modeling

The strong emphasis on MDA distinguishes XSem from other approaches to XML data modeling. Authors state in [28] that current approaches are not satisfactory for several reasons.

ERbased Approaches ER-based approaches are inspired by ER diagrams used in conceptual modeling for relation databases. There are entity types that model real world concepts and associations that model relationships between them. One approach is called EER, and it is extended ER with constructs that adds constructs specific for DTD. Similarly, there is an approach called XER that add constructs for XML schema. These aproaches hovewer do not apply MDA, because since XML schema constructs are incorporated in conceptual schema, designers must think how data will be represented in XML schema during definition of conceptual model. But in conceptual level the designer should model independently of the target platform. Also, there is no relation between XML schemas and for two different XML formats designer must create two conceptual diagrams.

UML UML-based approaches use UML diagrams for conceptual model. There are classes that model real world concepts and associations that model relationships between them. This approach is used in Enterprise Architect. Because UML class diagrams itself cannot express the XML format, they are extended with profiles that guide the translation to XML format. Problem is that PSM derivation is only automatic. There can be many PSM schemas based on the same PIM schema that have different representation, so it mus be the designer who decides how the PSM schema should look like and tool should only assist him. Automatic derivation cannot know how the target structure should look like, so if designer needs more PSM schemas for one PIM schema, he must model more PIM schemas [28].

XML Schema Visualization XML Schema visualization focuses on visual representation of XML Schema constructs. It does not consider MDA at conceptual model at all, so it faces mentioned problems like dealing with change since create schemas do not have correspondence to conceptual entities.

4.3 XSem

We already said that Xsem is based on principles of MDA and focuses on modeling XML schemas. It uses UML class diagrams extended with additional features. XSem considers PIM and PSM levels, where PIM models real-world concepts from the application domain and PSM models these concepts in the platform specific formats. This means that PSM schemas are derived from PIM schemas. Because we described that PSM is always specific to target platform in which we develop the system and since Xsem focuses on modeling XML schemas, it makes an intuitive sense that PSM schemas model concrete XML formats. The main feature of XSem is that PIM and components are formally related with PSM components. There can be multiple PSM schemas that are all derived from the same PIM schema, each for one concrete XML format that is used in the system. Concepts from PIM schema have their interpretation in PSM schemas and similarly concepts in PSM schema are mapped to the concepts in PIM schema. This is useful when we need to model the same concept from PIM schema in more XML formats that are used in different situations. This reflects the real life scenario where for example two messages use the same concept from PIM schema, but their XML representation is different.

This connection between model concepts and multiple interpretations has many advantages. The fact that PSM schemas are linked to the same PIM schema has a great benefit when dealing with change, because we can see where the change needs to be propagated and most importantly, in some cases this propagation can be automatic. So when user changes associations, classes, attributes and others, these changes can be automatically propagated to interrelated components. This connection also helps to understand the semantics behind XML documents.

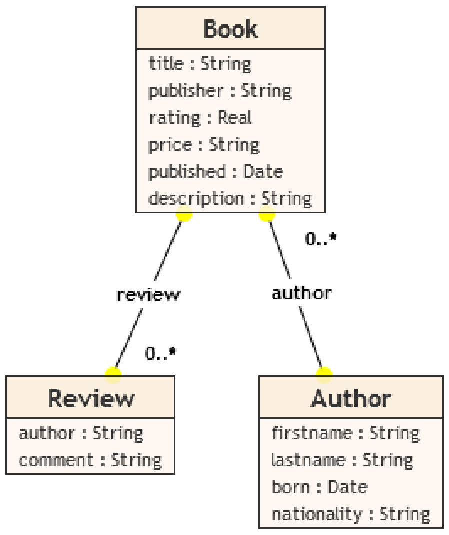
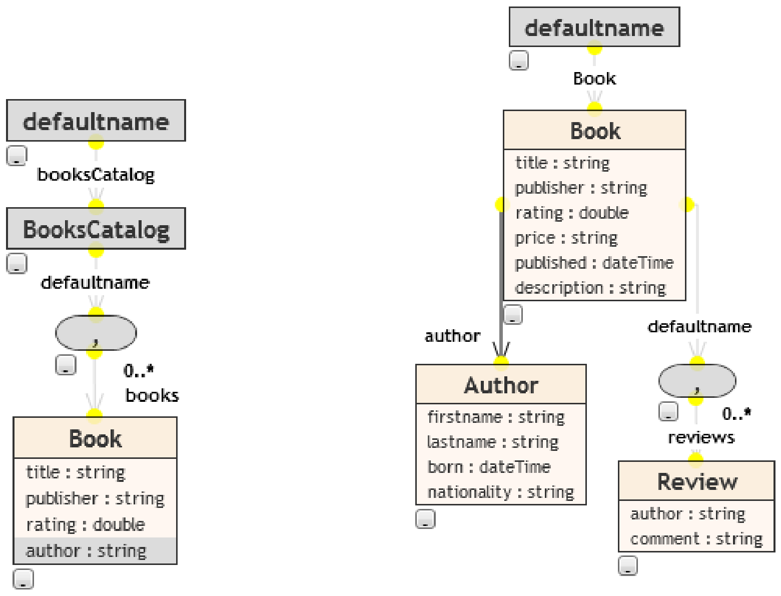


Figure 4.1: PIM schema for book



(a) PSM schema for message with book catalog (b) PSM schema for message with book details

Figure 4.2: PIM schema for book domain and related PSM schemas

Example Consider figure 4.1. We have PIM concept Book that has properties title, associated concept Author, publisher, price, rating and description. When considering the message send from the server to the client for the purpose of single book presentation, the message could contain PSM concept for Book that contains all the properties. On the other hand, when sending the book catalog with list of books, it is sufficient to provide for example only book title and author. Both messages use the same PIM concept, but their XML representations are different.

This propagation can actually occur at many levels. XSem considers evolution on five levels [26], as shown in figure 4.3. The *extensional level* contains actual XML documents. The *logical level* contains XML schemas that describe the XML formats. The *operation level* contains queries over XML documents in respect to the XML format. The *platform-independent* level contains conceptual schemas that we have already described. The platform-specific level contains also already described PSM schemas. Components from one level are always connected with components from upper level and from lower level, which allows the propagation of the change to all affected places. The platform-independent and platform-specific levels are called conceptual levels. The remaining three levels, the schema, operational and extensional, containing the actual files representation in the system, are called logical levels.

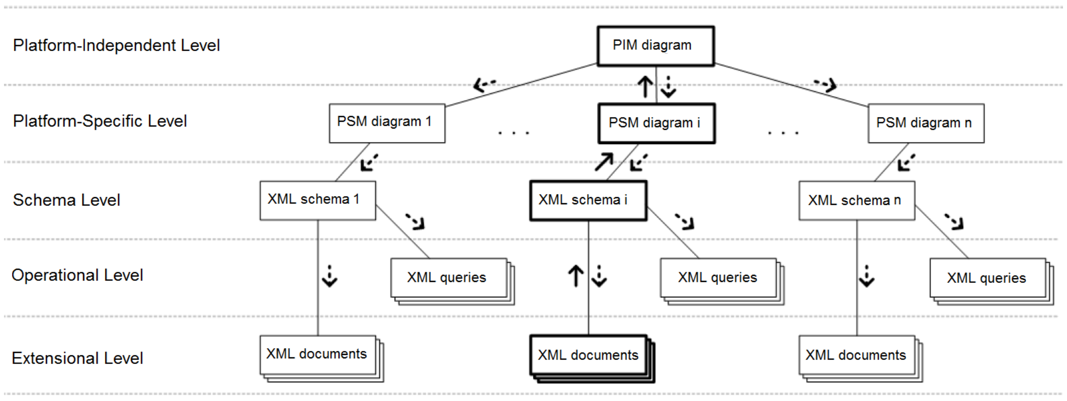


Figure 4.3: Five-level evolution architecture (source [26])

4.3.1 PIM schema

PIM schemas are visualized using UML class diagrams. These diagrams are simplified, because some features are unnecessary for XML data modeling, for example class operations. Most important constructs are classes, class attributes and associations. A PIM schema is shown in a non-hierarchical layout.

PIM Classes

PIM classes are core constructs of the PIM. They represent real-world conceptual entities. Classes have a name and can have attributes. Classes can be connected using associations. See figure 4.1, where classes are shown as a rectangles. Book is an example of a class.

PIM Attributes

PIM attribute models property of some real-world conceptual entity. Therefore PIM attribute is attached to concrete class. Attribute has a name, data type, optional default value and cardinality. See figure 4.1, where attributes are shown inside classes. Book property title is an example of an attribute.

PIM Association

PIM association models some relationship between real-world conceptual entities. Therefore PSM association connects PIM classes. Association have optional name and has cardinality in both ends. See figure 4.1, where associations are shown as links between classes. For example association between classes Book and Review specifies that each book can have zero or many reviews and each review has exactly one book. PIM associations can be self-referenced. Each association end can contain only one PIM class.

4.3.2 PSM Schema

PSM schemas are also visualized using UML class diagrams. In this case UML diagrams are extended with additional constructs that provide better support for XML modeling. These constructs have their own visualized form for better readability. Because PSM schema represents XML, it used hierarchical structure and has the form of forest. PSM constructs are derived from PIM components and this connection is remembered. Thanks to this changes can be automatically propagated to affected components, although not all changes are propagated. PSM schema can be directly translated into XML Schema. This process is automatic and unambiguous, as described in [28].

4.3.3 PSM Class

PSM class models representation of some PIM class in XML format. Some PSM classes can hovewer exist without any related PIM class and be used just as a container. PSM class has a name, optional attributes and optional associations. Class name can be different from interpreted PIM class name. Class can have at most one parent association. Classes are only constructs that can be both top roots or leafs. One special class must always exist in each PSM schema, called PSM schema class. When PSM schema is translated to XML, an instance of a PSM class is modeled as a sequence of elements representing its content. This sequence is enclosed in an XML element with the name from the parent association.

4.3.4 PSM Association

PSM association connects PSM classes and PSM content models. It have optional name and cardinality in both ends. Each association end can contain only one PSM class or PSM content model.

4.3.5 PSM Attributes

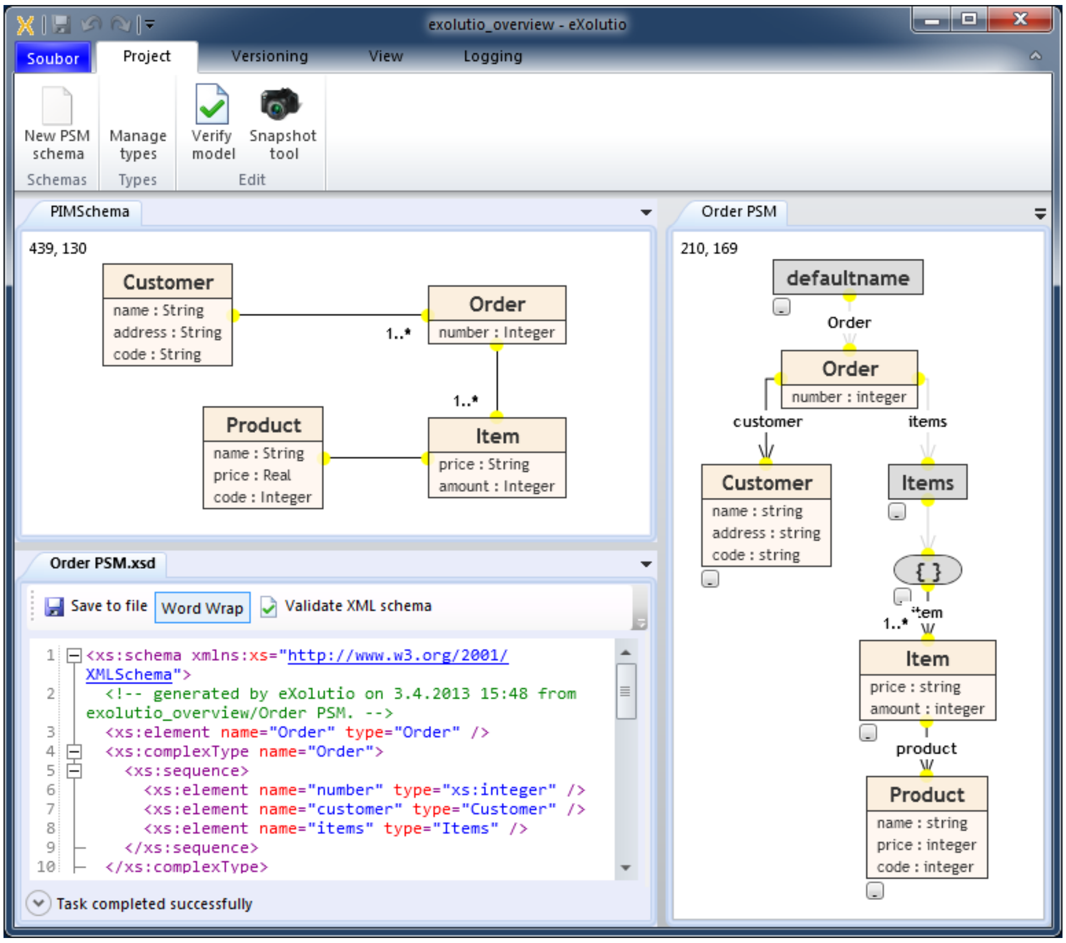
PSM attributes are attached to concrete PSM classes. They can be either derived from PIM attributes of represented PIM class or they can stand alone, which means that they have no counter part in PIM model. PSM attribute has a name, data type, cardinality and optional default value. Attribute name can be different from represented PIM attribute. PSM attribute has also an XML form that specifies whether in XML format it will be translated as XML element or XML attribute.

4.3.6 PSM Content model

PSM content model enables to model XML schema constructs sequence, choice and set. Content models are visualised using rounded rectangles. They have always parent and child associations. Hence, they cannot be top level or leaf nodes.

4.4 eXolutio

We will now briefly describe eXolutio, an experimental tool that implements conceptual modeling for XML. We can divide eXolutio modeling into multiple steps. These steps need not to be sequential, they can overlap when designer decides to change previously defined concepts.



```
<xs:schema xmlns:xs="http://www.w3.org/2001/
XMLSchema">
  <!-- generated by eXolutio on 3.4.2013 15:48 from
exolutio_overview/Order PSM. -->
  <xs:element name="Order" type="Order" />
  <xs:complexType name="Order">
    <xs:sequence>
      <xs:element name="number" type="xs:integer" />
      <xs:element name="customer" type="Customer" />
      <xs:element name="items" type="Items" />
    </xs:sequence>
  </xs:complexType>
```

Task completed successfully

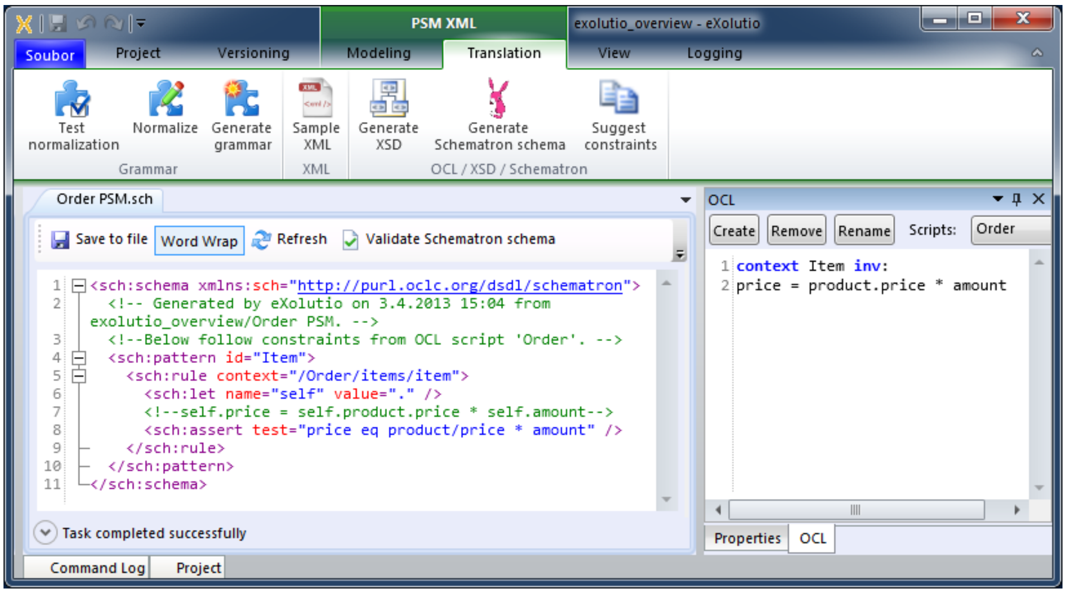
Figure 4.4: Exolutio PIM and PSM schema modeling. From left we can see PIM schema, then derived PSM schema. Below PIM schema, we can see generated XML Schema.

1. **PIM Schema** At the very first step, designer must define the PIM schema, where he defines real-world conceptual entities of the problem domain using classes, associations, class attributes, content models and other constructs that we have already described.

2. **PSM Schema** From this PIM schema, designer can later derive multiple PSM schemas, where he defines concrete XML representation of the concepts from the PIM schema. Constructs from PSM schema can represent an interpretation of another construct from PIM schema, in this case eXolutio maintains a connection between these constructs. This connection enables propagation of changes between associated concepts.

3. **XML Schema generation** Finally, designer can generate XML Schemas from PSM schemas. Generated XML schema is specified by PSM schema unambiguously as described in [28]. Designer can also generate sample XML documents based on this PSM schema and he can also validate these documents against the schema.

4. **OCL constraints** Another significant feature is support for defining OCL constraints against the PIM and PSM schemas. OCL constraints against PSM schema can be transformed into Schematron schema, which provides more detailed control over XML document than the XML Schema alone. This particular feature will be very important for us later.



```
<sch:schema xmlns:sch="http://purl.oclc.org/dsdl/schematron">
  <!-- Generated by eXolutio on 3.4.2013 15:04 from
exolutio_overview/Order PSM. -->
  <!--Below follow constraints from OCL script 'Order'. -->
  <sch:pattern id="Item">
    <sch:rule context="/Order/items/item">
      <sch:let name="self" value="." />
      <!--self.price = self.product.price * self.amount-->
      <sch:assert test="price eq product/price * amount" />
    </sch:rule>
  </sch:pattern>
</sch:schema>
```

Task completed successfully

```
context Item inv:
price = product.price * amount
```

Figure 4.5: Exolutio OCL constraints and generated Schematron schema. In right side we can see OCL constraint against the PSM schema from figure 4.4 which states that for each item holds that item price is equal to price of corresponding product multiplied by its amount. In left side we can see Schematron schema generated from OCL constraints.

Figures 4.4 and 4.5 shows described features of eXolutio.

5. Executable model analysis and design

We will now analyze how to represent business artifacts in an executable model, by which we mean concrete Active XML representation that can be executed and demonstrated. After that, when we have a suitable representation, we will analyze how to extend eXolutio to support definition of business artifacts and generation of these artifacts into proposed representation.

Proposed representation will be composed from one or multiple Active XML documents along with necessary XQuery code designated for operations over these documents. But such active documents alone are still merely static, unless they are evaluated in Active XML framework using Active XML engine. So, after creating concrete representation, we must execute it inside Active XML framework.

In addition to this, there must be also another system that corresponds to the functionality of *Artifact Service Center*, which was described in more detail in section 2.2.11. We will develop this system as a web application on top of Active XML and use it to execute generated Active XML representations. These representations will be internally managed using Active XML framework underneath. As described in section 2.2.11, this system will be also responsible for communication with external world, communication with human performers, receiving incoming events and processing these events. Such event processing will be performed using Active XML framework. Many operations of this web application will be performed with respect to generated Active XML representation. For example, exposed web service for receiving incoming events must depend on concrete model definition. Actual implementation of this system will also depend on proposed Active XML representation.

5.1 Artifact evolution

In this section, we will analyze artifact evolution in an executable model. We must analyze what exactly we need from an executable model, how it should behave and what problems we need to solve. We are talking about artifact evolution, but what does this evolution exactly mean in our context? Generally, it means making changes and updates to an artifact information model. These changes can happen as a consequence of incorporating single incoming event into the system and by execution of artifact tasks. Both incorporation of incoming events and invocation of artifact tasks are driven by predefined lifecycle, specified using guards, milestones, stages and declarative rules. These concepts together specify the evolution of the system. We can imagine the evolution as a repetition of following steps:

1. Incoming event arrives to the ASC system and system eventually starts its processing.
2. Processing incorporates an immediate effect of an incoming event and updates information model.

3. Processing evaluates applicability of individual rules in predefined order and if some rule becomes applicable, it is fired. Firing a rule can either open a stage, close a stage, achieve a milestone or invalidate a milestone. Opening a stage can generate a request for task invocation, if opened stage is atomic. These requests are collected during event processing. Then evaluation continues. Processing terminates when there are no applicable rules left.

4. When the processing terminates, system invokes tasks for generated requests. Invoked tasks can make changes to information model during their execution and as a result of their termination, which corresponds to another external event.

Figure 5.1 illustrates these steps.

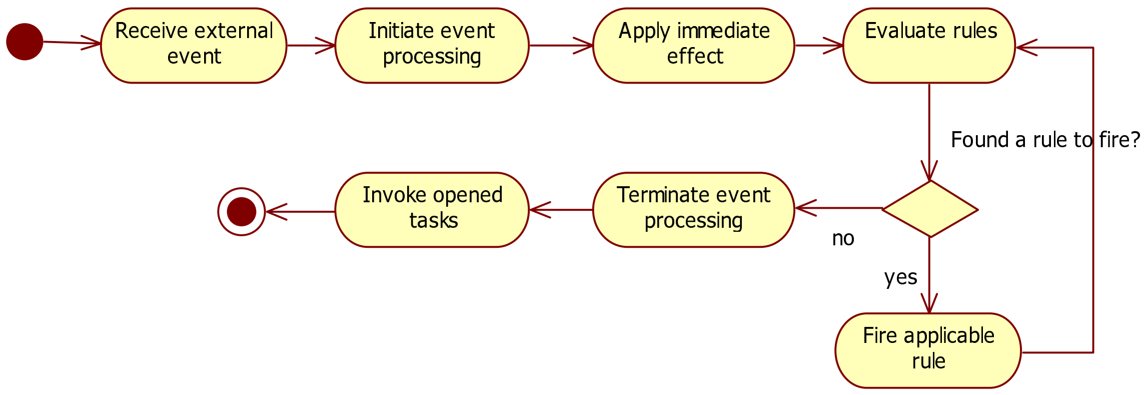


Figure 5.1: Evolution of business artifacts in an executable model

This means that we have to implement particular mechanisms that will be responsible for realization of previously described steps. Our idea is that we have a system that correspond to **Artifact Service Center** described in chapter 2. This system manages artifact instances and enables communication with external world. So, our executable model must be able to store business artifact instances, create new business artifact instances, receive external events, queue them for processing in correct order, initiate event processing, update information model of business artifacts, evaluate PAC rules and invoke services that correspond to tasks. We will now analyze these requests.

5.1.1 Processing b-step

We will start with b-step processing, concretely rules evaluation, because it is one of the most important part of an executable model, since it realizes the lifecycle model in practice. We will discuss it together with information model XML representation because as we will see soon, these two concepts are highly connected.

Rules evaluation is the second part of an external event processing, right after application of an immediate effect. Since we implement information model of business artifacts with XML, an intuitive approach is to implement rules evaluation with XQuery. This has a great advantage, because with XQuery we can implement all three parts of PAC rules, which are preconditions, conditions and actions. It is because XQuery is suitable for both read-only and update queries over the XML document, where the first is useful for evaluation of conditions and preconditions and the second is useful for realization of actions. Another very important reason to choose XQuery is that eXolutio already contains support for OCL constraints over PSM schemas and capability to generate Schematron schema based on these constraints. Schematron uses XQuery expressions, so eXolutio can internally generate XQuery expressions from OCL constraints, we only need to adjust it for our needs.

Therefore, our goal is to have a XQuery code that corresponds to rules conditions, actions and evaluation. This code traverses PAC rules in topological order that must satisfy order described in chapter 2, evaluates their conditions and once it finds an applicable rule, it immediately fires it. Firing a rule means updating information model, concretely changing values of corresponding status attributes. Note that firing rule really changes only status attributes, while data attributes are consistent during entire rules evaluation. Which status attributes are changed depends on rule category (guard, milestone achieving sentry, milestone invalidating sentry). Firing rule can also generate a request to start a task, if firing rule opens an atomic stage. So, our rules evaluation will evaluate individual rules and fire applicable ones until no rule is applicable, and then return set of generated requests to start a task.

Consider we have a following sentry, related to the *customerOrder* artifact.

guard*(customerOrder.sendToManufacturer):*
on *self.productCodeSet.achieved()* ***if*** *manufacturerID = 5*

We can see that it has only the condition part specified, there is no precondition and action part. This is because these parts are automatically derived, since they are based only on rule category and target status attribute (stage or milestone). This rule defines under which condition stage *sendToManufacturer* of artifact *customerOrder* opens. The rule is relevant to every artifact instance of *customerOrder* type. So, how to translate the rule into XQuery evaluation? To see this, we must first shortly discuss information model XML representation.

5.1.2 XML representation

We said that we want to represent business artifacts with XML. Because we use Active XML and Active XML uses native XML database, our business artifacts will be eventually stored in this database. We only need to think about concrete representation. Representation for data attributes will highly depend on model definition created by the designer. But information model must also contain status attributes to record artifact progress through its lifecycle. Status attributes, unlike data attributes, should not be explicitly defined in the information model by the designer, because they can be derived from the lifecycle model. This means that we have to find a way how to represent status attributes in target XML realization. As we will see later, this issue is closely connected with PAC rules evaluation and for this reason, we will discuss this issue together with PAC rules.

We can now discuss how to realize status attributes effectively with respect to rules evaluation. Using XQuery to evaluate rule conditions means that we should take care when defining artifacts XML representation with respect to performance. Rules evaluation happens quite often, almost every time when external event is incorporated and it can process large amount of data, so it is suitable to use such XML representation of the information model that supports higher performance when testing rules condition. Of course, a great part of information model representation is in designers hands, because it is only up to him to specify this realization and associations between artifacts in eXolutio. This however applies only for data attributes, which designer explicitly defines on business artifacts. On the other way, status attributes are derived from lifecycle model, so it depends on concrete executable model implementation how to realize them in XML representation, meaning how will they look like and where to put them in artifact XML realization.

```
<customerOrder>
   <customerOrderID>1</customerOrderID>
   <productCode>2</productCode>
   <customer>
      ...
   </customer>
   <manufacturerID>3</manufacturerID>

   <milestone name="ProductCodeSet"
      status="true" time="1" />

   <stage name="SendToManufacturer"
      active="true" time="2" />
</customerOrder>
```

(a) Stage and milestone names are specified using attribute value

```
<customerOrder>
   <customerOrderID>1</customerOrderID>
   <productCode>2</productCode>
   <customer>
      ...
   </customer>
   <manufacturerID>3</manufacturerID>

   <ProductCodeSet
      status="true" time="1" />

   <SendToManufacturer
      active="true" time="2" />
</customerOrder>
```

(b) Stage and milestone names are specified using element names

Figure 5.2: Two among many possibilites of status attributes realization

We illustrate this in figure 5.2, which shows only two of many possibilities. First approach uses attributes to distinguish individual status attributes, while second uses element names for the same thing. This distinction is very important, because rules evaluation will very often search status attributes by name and inspect their values. So, we have to find out which alternative is more friendly for this. Because we use eXist native XML database, we can follow optimization guidelines in [31] and see that eXist indexes elements by default, so searching by element name will be potentially much faster then searching by attribute value.

5.1.3 Rule representation

Now, when we know how the information model XML realization will roughly look like, we can discuss rule realization. So, if we suppose that we have all *customerOrder* artifact instances in file artifacts.xml, the XQuery realization could be like this:

```
for $artifact in doc('artifacts.xml')/artifacts/customerOrder
where
  $artifact/productCode/@status eq 'true'
  and $artifact/productCode/@time eq $now
  and $artifact/manufacturerID eq '5'
return (
  update value $artifact/sendToManufacturer/@status with 'true',
  update value $artifact/sendToManufacturer/@update with $now,
  generate-task-request('sendToManufacturer', $now)
)
```

Figure 5.3: Rule realization

This XQuery representation corresponds to the behavior of evaluating and applying one concrete rule. It iterates over *customerOrder* artifacts, evaluates conditions and if some artifact satisfies the conditions, the update part of the code changes status attributes of the artifact and generates request to start associated task. We can also see how condition part tests status attributes and finds them by name, which will be very frequent scenario. This realization is rather simplified and in later implementation, much more code will be necessary, but this should give a good initial overview of the realization look. Also, concrete realization will highly depend on the XML representation of artifacts.

Each such rule realization could be encapsulated within unique XQuery function with name that is derived from rule essence and that clearly associates the function and the rule, in this case for example *CustomerOrder-SendToManufacturer-Guard*. Having functions for each rule, we could then evaluate rules in correct order from some function.

```
declare function local:evaluate-rules($now as xs:integer) as empty() {
  evaluate-CustomerOrder-SetProductCode-Guard($now),
  evaluate-CustomerOrder-ProductCodeSet-Achiever($now),
  evaluate-CustomerOrder-SendToManufacturer-Guard($now),
  (: here follow functions for remaining rules :)
}
```

Figure 5.4: Rules evaluation realization

Again, the realization is simplified. We will later analyze rules evaluation in more detail.

5.1.4 Rules precondition

We know from section 2.3 that every rules has a precondition which must be evaluated with respect to initial snapshot of B-step processing. However as individual rules are fired, information model is changed and initial snapshot is transformed into different pre-snapshot. This means that preconditions of the rules cannot be evaluated during rules evaluation, but they must all be evaluated before we change the model by applicable firing rules. On the other hand, preconditions are dependent only on one status attribute, depending on rule category. Therefore to achieve the same result without iterating over artifacts twice, we will add new attribute to milestones and stages, called *stableStatus*. This *stableStatus* attribute will have the same value during entire B-step processing and only at the end of B-step processing its value will be updated to match actual status value. Now, rules preconditions will not evaluate against real status attributes, but against these *stableStatus* attributes in order to evaluate against the initial snapshot.

5.1.5 Immediate effect

Rules evaluation is only one step of external event processing. Another important step is an application of immediate effect. Immediate effect makes changes to the information model, so it will be implemented using XQuery as well. As we know from chapter 2, immediate effect must incorporate event payload into information model and set *mostRecentEventType* and *mostRecentEventType* to all affected artifact instances. Setting *mostRecentEventType* and *mostRecentEventType* is rather straightforward, but incorporating event payload is more intricate. We will analyze it in following section.

5.1.6 Tasks

We will now discuss how to realize task processing in our executable model. Because tasks usually correspond to external web service invocations, we will use web services to implement task processing. Some tasks however do not correspond to web services, for example an assignment task. Nevertheless, such tasks can be also implemented as web services, only with difference that such web service calls will be always call to a local web service. This will simplify our analysis and as we will see, it is also suitable, because Active XML provides web services to execute generic query over our documents. Such generic queries can easily implement any custom assignment task.

We have chosen Active XML for artifacts implementation, so we must find a proper representation in this framework. Active XML is centered around XML documents. These documents can contain embedded service calls. Activation of embedded service calls can enrich the document with the service call result. We can see that this is analogous to artifact tasks, which also perform some actual business relevant work and enrich artifact information model. Hence an intuitive approach is to put tasks directly into XML documents in a form of embedded service calls.

Let A be an artifact type, $Tasks_A$ be a set of all tasks owned by artifact A. Let $ArtInstances_A$ be a set of all artifact instances of type A. Then, create an embedded service call for each item in $ArtInstances_A \times Tasks_A$ and place it in the XML document.

```
<customerOrder>
   <customerOrderID>1</customerOrderID>
   <productCode>2</productCode>
   <customer>
      <firstname>Alice</firstname>
      <lastname>Carter</lastname>
         <address />
   </customer>
   <manufacturerID>3</manufacturerID>

   <!-- status attributes ommited here -->

   <sc method="customerOrderSetProductCode@workflowService" />
   <sc method="customerOrderSendToManufacturer@workflowService" />
</customerOrder>
```

Figure 5.5: Embedded service calls for artifact tasks

An immediate problem is where to put these embedded calls and when to activate them?

5.1.7 Service calls location

We know that when embedded call returns, its result is inserted into the document either as a sibling of the *sc* node or somewhere else in the document if more complex merging strategy is specified. So in case of sibling insertion, we should place *sc* nodes into the location where the call result should be inserted. This is straightforward if result contains for example only the product code, where we just place *sc* node as a sibling of the *productCode* node. When the service call terminates, its result updates or replaces current *productCode*. Unfortunately, task can generally provide a result that breaks this simple merging strategy. There are at least following problematic scenarios:

- Result is composed from nodes that are not located together in one place in the document, but are spread across the document. This means that sc node cannot be a sibling of all these nodes together.
- Result contain nodes that have no representation in the document. This means that inserting it into the document would make the document invalid against the schema.
- Result contain nodes that have similar representation in the document, but contain only restricted set of child nodes. This means that inserting it into the document could make the document invalid against the schema, or at least unintentionally delete some child elements.

```
<customerOrder>
   <productCode>5</productCode>
   <manufacturerID>6</manufacturerID>
</customerOrder>
```

Figure 5.6: Nodes *productCode* and *manufacturerID* are not siblings in artifact XML document, so it is not possible to put sc node for this task as a sibling of both nodes. Moreover, node *customerOrder* has no representation in artifact XML document, so inserting it into the document would make the document invalid against the schema.

```
<customer>
   <address>
      <city>Prague</city>
      <postcode>11000</postcode>
      <street>Malostranske namesti 25</street>
   </address>
</customer>
```

Figure 5.7: Node *customer* has similar representation in artifact XML document, but both representation have different set of child nodes. Result contains node *customer* that has only address child, so directly inserting it into the document could replace customer firstname and lastname.

We could potentially solve these problems by designing artifact XML schema in order to fit all task results or design task results to fit artifact XML schema, but generally, it would not be possible. Also, it would be very hard to incorporate any changes, because artifact representation and task result representation would be very tightly coupled. Finally, many tasks can be implemented as external services and we might have no control of their interface.

For this reason, we must use more complex merging strategy. Active XML theoretically provides also merging strategy based on ID-fusion, but this is not applicable as well, because we probably do not want each element to have its ID and even if it had, it would still not solve the problem that individual parts of the call result can be spread across the document. Therefore, we must have an individual merging strategy for each task result. This strategy would be some predefined procedure that would take the task result and merge it with the document. Naturally, this merging strategy will be implemented using XQuery. We will further call this an **event binding**, because it binds task output parameters to information model representation.

We will enable the designer to define task output binding for each artifact task. This binding will be realized in the form of XQuery function, that will accept task result and artifact data and it will update this artifact data with data from the task result. Figure 5.8 illustrates this.

```
declare function binding:CustomerOrder_SetProductCode(
  $artifact as node(), $message as node()) as empty() {
  update value $artifact/produtCode with $message/customerOrder/productCode/text()
};
```

Figure 5.8: Example of event binding

However, Active XML does not directly provide such merging strategy for service call returns. On the other way, Active XML provides two mechanisms that could help us - declarative service composition and generic query service. Combining these two concepts, we can have one *sc* node for real service call and parent *sc* node calling generic query service for merging process. Figure 5.9 illustrates this.

```
<sc method="executeGenericQuery">
   <query>
      let $result := <sc method="customerOrderSetProductCode" />
      let $artifact := doc("artifacts.xml")//customerOrder[id=1]
      return binding:customerOrderSetProductCode($result)
   </query>
</sc>
```

Figure 5.9: AXML Query handler node

It works as follows. Framework requests evaluation of *sc* node for *executeGenericQuery*. Because this *sc* node depends on *sc* node for customerOrderSetProductCode, first *sc* node for *customerOrderSetProductCode* is evaluated. Its result is then used for *sc* node for *executeGenericQuery*, which does the merging. We will call this parent node a **handler node**, because it handles the processing of child task termination.

Input parameters We described how service result must be merged into the document. Similarly, when task starts and corresponding embedded service call is activated, we must provide input parameters for the call, if it requires any. These input parameters bring along the same issues that we described in result merging, because input parameters can contain data that need to be obtained from different parts of the document. This can be again achieved using service composition, where child sc node is a call to generic query service that obtains input data from the document and parent sc node is a call to requested service.

We were now talking about call activation and termination, but when does this happens and how?

5.1.8 Service calls activation and termination

We will now analyze when to activate embedded service calls and how to control their termination. A direct approach would be to activate service call when corresponding task should be started as a result of opening an atomic stage during b-step. However, we said that tasks are started after the b-step finally terminates, so we cannot activate these calls immediately, we have to wait for b-step termination. When b-step terminates, we can activate at once all embedded calls that correspond to tasks that should be started. We must activate these calls at once, because we know that GSM stages and their tasks can run in parallel. But this brings a problem, because Active XML framework does not provide an option to activate set of embedded calls - it can either activate all calls in the document or only a single call in the document. We can solve this by first disabling calls that should not be activated (mark them as disabled) and then activating entire document. This solves the activation problem, but what about termination?

Embedded service call terminates when the service call returns a response, but this can happen virtually anytime. But we know that every task termination is considered by the system as an external event that must be incorporated in concrete B-step. So it should not happen, that one task terminates, corresponding b-step starts and another task terminates during the B-step and starts another B-step. Only one B-step can be processed at time. Similarly, some request event could arrive into the system during the B-step. Therefore, we must have some mechanism that queues external events, including task termination events and that allows to start corresponding B-step only when no other B-step is currently being processed.

There is also another problem with task termination. Imagine following scenario:

1. External event is received and b-step processing starts
2. Processing B-step opens two atomic stages, so two tasks t_1 and t_2 should be later started
3. Processing B-step ends, so system activates two embedded calls for tasks t_1 and t_2.
4. First task t_1 terminates and B-step processing starts
5. Processing B-step opens another atomic stage, so another task t_3 should be later started

6. Processing B-step ends, so system **should** activate another embedded call for task t_3.

However, service call for task t_2 is still running, so Active XML document is still being evaluated. This is because when Active XML document evaluation starts, the evaluation terminates after all activated calls terminate as well. Before that we cannot activate any embedded call or evaluate the document. We must wait for previous evaluation termination. But this very limits the parallelism that GSM model supports, especially if we start some long running service. We can see that we have two problems with this approach:

1. B-steps interleaving. **Problem:** When one task terminates during ongoing b-step processing, it immediately starts another B-step processing. **Consequence:** We need to guarantee that when one task terminates during ongoing B-step processing, it does not initiate another interleaving B-step processing.

2. Tasks synchronization. **Problem:** When one b-step processing starts multiple tasks, no other B-step processing can be initiated before all started tasks terminate. **Consequence:** We need to enable processing of another B-step without waiting for termination of all started tasks.

To solve the problem with B-steps interleaving, we have two options:

1. Invoke embedded service calls asynchronously

2. Use handler node that delays task termination

Invoke embedded service calls asynchronously

We could invoke embedded service calls asynchronously to solve the interleaving problem. Unfortunately, Active XML supports only synchronous calls. However, we can create a workaround. From embedded service call we do not invoke the real requested service, but our custom wrapper service, which asynchronously invokes the requested service and immediately returns, which terminates embedded service call, but does not terminate the task. Task is still running within asynchronous call. When asynchronous service call returns as well, our wrapper service routes this information to the system as an external event. Only then the task really terminates. This way, we can always control the termination and external event processing. This means that we do not need the handler nodes, because we do not require that Active XML automatically starts handler service when child embedded call terminates. We queue external events and start event processing ourselves. Figure 5.10 illustrates this.

```
<customerOrder>
   <customerOrderID>1</customerOrderID>
   <productCode>2</productCode>
   <customer>
      <firstname>Alice</firstname>
      <lastname>Carter</lastname>
         <address> ... </address>
   </customer>
   <manufacturerID>3</manufacturerID>

   <!-- status attributes -->

   <sc method="startTwoWayService@utilityService">
      <request>customerOrderSetProductCode</request>
      <service>workflowService</service>
   </sc>

   <sc method="startTwoWayService@utilityService">
      <request>customerOrderSendToManufacturer</request>
      <service>workflowService</service>
   </sc>
</customerOrder>
```

Figure 5.10: Asynchronous invocation

We must note here that this asynchronous invocation of embedded service calls is not the same as realization of synchronous and asynchronous task, so asynchronous invocation does not mean that we support only asynchronous tasks. When synchronous task is invoked, it is considered to be terminated when the corresponding service call terminates. When asynchronous task is invoked, it is considered terminated as soon as corresponding service call is invoked. We can easily simulate this asynchronous embedded calls, because all that matters is when we return external event to the system. The fact that embedded service call has already terminated does not mean anything, what is important is external return event.

Use handler node to signal task termination

We can use handler node that is very similar to query handler node. But in this case, handler node does not immediately incorporate task termination event into information model, but it only signals task termination to the system using determined web service. System then creates external event corresponding to task termination event, queues this event for later processing and starts corresponding b-step processing when it is appropriate.

```
<customerOrder>
   <customerOrderID>1</customerOrderID>
   <productCode>2</productCode>
   <customer>
      <firstname>Alice</firstname>
      <lastname>Carter</lastname>
         <address />
   </customer>
   <manufacturerID>3</manufacturerID>

   <!-- status attributes -->

   <sc method="signalTwoWayServiceTermination@utilityService">
      <sc method="customerOrderSetProductCode@workflowService" />
   </sc>

   <sc method="signalTwoWayServiceTermination@utilityService">
      <sc method="customerOrderSendToManufacturer@workflowService" />
   </sc>
</customerOrder>
```

Figure 5.11: Using handler nodes to signal task termination

We can see that these two approaches are somewhat opposite. The handler approach is more consistent with Active XML, because it calls real web services and solves the problem using service composition, but it requires lot of additional XML code in the document, which makes it less understandable. Unfortunately, it does not solve the synchronization problem, because activated embedded calls are active until web service call really terminates. On the other hand, with asynchronous invocation approach, the embedded service call immediately terminates and so it does not block any subsequent b-step processing. Therefore, it seems to be more suitable.

Another problem with this approach is that because service calls are part of the document, they must be defined in document XML Schema. This needs to be taken into account when generating XML Schema from eXolutio. However, since service results are merged using binding query anyway, it is not necessary to put these calls into the same document with business artifacts. We could put them in different document so that we do not violate the document schema.

We have seen that using embedded service calls directly in the same document with artifacts may pose a problem. We could put calls into another file, but the problem would not come away, because the another file would have the same problem with parallelism and document evaluation. Strictly speaking, to avoid this, we would have to put each task call into its own XML file, but managing task calls spread across many documents would be harder and impractical and it would also lack the meaning of embedded service calls.

5.1.9 Comparison with AXML Artifact model

We will now look at AXML Artifact model that we described in section 3.4. This model represents artifacts in XML documents and uses embedded service calls to implement evolution of business artifacts. Embedded calls are guarded using queries that control under which conditions can be the guarded service call invoked. If we imagine an embedded service call as a task, we can say that these call guards are similar to our GSM stage guards, because similarly like our GSM stage guards they control task, respectively service call activation. However, semantics of these guards is different then ours and it may bring problems that we have described earlier.

Suppose we have a document with more embedded calls, implementing AXML Artifact model. Suppose we consider call guards to be an implementation of GSM stage guards. Call guards are evaluated when corresponding continuousCall is activated, which can be done explicitly (activate this call) or implicitly when evaluating entire document or when evaluating parent call. So, in our case, to perform complete rules evaluation, we would have to evaluate entire document, which would evaluate individual embedded calls and therefore evaluate call guards.

1. When call guard is satisfied, then corresponding service call is immediately invoked. But we can never start some task as soon as associated stage guard is satisfied, we must always wait until current b-step terminates and only then start generated tasks. This could be solved by not calling real service, but calling wrapper service instead and this wrapper service would call the real service when appropriate.

2. When embedded service call returns, its result is immediately incorporated into the document. But we need to handle task termination as an external event that must be eventually sent to the system and processed when appropriate, because it must not interleave with current b-step processing. This could be solved by termination handler, that we have previously described.

3. When we evaluate an entire document, the order of embedded call activations is not determined, so evaluation of individual call guards is not determined as well. But we need to strictly follow topological order of guards evaluation. This could be solved by using ordering constraints on individual calls, which is a feature that AXML supports.

4. When some call guard is satisfied, it only invokes corresponding service call. But we need to also update an information model, concretely record change of stage activity into corresponding status attributes. This could be solved by calling wrapper service that would first call generic query service to update an information model and then invoke real service. This generic query service would not interleave with another one, because in paragraph 3 we have described that individual embedded calls are ordered and processed one by one.

5. Call guards actually correspond only to such GSM stage guards that protect atomic stages with task inside, because call guards watch only embedded service calls. But we need also guards for composite stages. This can be solved by flatting the model into atomic stages only, which might not be a problem actually, because it is done only at an implementation level. So, at design level, designer still thinks about composite stages, but at implementation level, the composition is flatten into atomic stages. Of course, it is not straightforward, for example we need to preserve invariant rules (PAC_6 for example).

   Example For example, suppose that we have stage S_1 with child stages S_2 and S_3 and milestone m_1, all S_1, S_2 and S_3 are active, and we have a rule:

$achieve(m1):\ on\ AbortEvent.onEvent()$

This means that on event *AbortEvent*, m_1 becomes achieved, so S_1 becomes closed, so by PAC_6, S_2 and S_3 becomes closed as well. So, if we flatten the model, we have to preserve such behaviour and *AbortEvent* must still close both S_1 and S_2.

6. Call guards do not correspond to milestone achieving sentries and invalidating sentries. But we need to evaluate these rules as well in order to update milestone status attributes. This could be solved by adding additional embedded service calls, which call guards would correspond to milestone achieving sentries and invalidating sentries. Satisfying such call guards would invoke only generic query service to update milestone status attributes.

We can see that we could adjust AXML Artifact model to fit our needs. Only problem with this approach is that it could be less efficient. Topological order of rules evaluation would be implemented using dependencies between activations of embedded service calls, so any time the document would be evaluated, Active XML framework would have to create entire dependency graph. In previous approach, where entire rules evaluation was expressed using XQuery, the evaluation order was strongly determined. Also, each status attributes update would be implemented as individual call to generic query service, while in previous approach all updates were performed inside the same query.

5.2 Selected approach

We have analyzed possible representation of business artifacts. Based on this analysis, we will implement concrete approaches. We have already described a naive approach with embedded service calls. We will also implement following two approaches.

5.2.1 Continuous calls approach

We can take an inspiration from AXML Artifact model and exploit its idea of the call guard implemented as continuous call. But instead of using the call guard as an evaluation of corresponding rule, we separate complete rules evaluation into a standalone query, similarly as we did in the first approach. Call guards of these continuous calls would only check if corresponding task should be started, which would be already recorded as a result of rules evaluation.

First we introduce concept of *task activation.* Task activation consist of task name, artifact name, artifact ID and filled parameters. It is created during rules evaluation to signal task opening event. Its XML representation is shown in figure 5.12. This structure can contain additional data, for example logical timestamp when the task was opened. There is analogous representation in the ASC system.

```
<taskActivation>
   <taskName>ShipOrder</taskName>
   <artifactID>1</artifactID>
   <artifactClass>CustomerOrder</artifactClass>
   <inputParameters>
      <date>15.1.2013</date>
      <customer>Alice</customer>
   </input>
</inputParameters>
```

Figure 5.12: Task activation represented in XML

In following text, by task category we mean one of starting two-way service, starting human service and sending one-way message. We now describe individual concepts in the approach. This approach uses one continuous call for each task category. Possible alternative is to have one continuous call for each possible task from the model. The behavior will be the same.

Documents

- We have one document with business artifacts, called BA document. This document contains information models for all existing business artifact instances.
- We have one document with generated task opening events, called TASK document. This document contains all task opening events that were generated in most recent B-step processing.

Service calls

1. We have one web service in the ASC, called *DelegationService*, which has one operation for each task category. This operation gets task activation as a parameter and it asynchronously invokes corresponding web service operation with corresponding input parameters (both read from task activation parameter). These three operations are *startTwoWayService*, *startHumanService* and *sendOneWayMessage*.
2. We have one embedded call to *executeGenericQuery* operation of *GenericQueryService* web service in the BA document. This call invokes the query for immediate effect and rules evaluation.
3. We have one embedded continuous call in the BA document for each task category. All these calls depend on the embedded call from bullet 2, so they can be activated only after that embedded call terminates. Each continuous call has as the first parameter an endpoint of the ASC system web service that realizes corresponding task. As the second parameter, continuous call has a query that returns sequence of wrapped task activations for each opened task corresponding category. Task activations are wrapped in parent element that corresponds to one of the ASC system operations from bullet 1, as shown in figure 5.13. As a consequence, continuous call invokes the web service at specified endpoint for each item of this sequence.

```
for $event in doc("/Peer/events.xml")/taskActivations/twoWayServiceCalls
return (
  <startTwoWayService>
    {$event}
  </startTwoWayService>
)
```

Figure 5.13: Task activation from continuous call

B-step processing

1. Incoming event is received by the ASC system and eventually selected for processing.
2. ASC system requests evaluation of business artifacts document using operation *evaluate* of *MaterializationService* provided by Active XML.
3. Active XML loads embedded calls in business artifacts document and creates activation tree based on embedded calls dependencies.
4. Because in this case all continuous calls depend on call for rules evaluation, Active XML calls *executeGenericQuery* operation of *GenericQueryService* query that performs immediate effect and evaluate rules.
5. Rules evaluation can open several tasks. Evaluation query records task activations for each opened task into the TASK document.
6. When rules evaluation terminates, operation *executeGenericQuery* terminates and Active XML invokes **in parallel** all three continuous calls. Every continuous call executes described query to get sequence of task activations and it invokes corresponding call for each task activation.
7. ASC system asynchronously invokes corresponding web service from given task activation and immediately terminates.
8. When invoked web service terminates, ASC system creates new incoming event corresponding to task termination adds it into the processing queue.

5.2.2 Pure query approach

We said that all opened tasks can be started only after b-step finally terminates. Therefore, another approach can be to not use embedded service calls at all and invoke service calls directly from the ASC system, using Java code.

Documents

- We have one document with business artifacts, called BA document. This document contains information models for all existing business artifact instances.
- We have one document with generated task opening events, called TASK document. This document contains all task opening events that were generated in most recent B-step processing.

Service calls

In this approach, there are no embedded service calls and no delegation web service for signaling from Active XML.

B-step processing

1. Incoming event is received by the ASC system and eventually selected for processing.

2. ASC system starts a query that performs immediate effect and evaluate rules. Started query gets incoming event as a parameter. This query is called using *executeGenericQuery* operation of *GenericQueryService* provided by Active XML.

3. Rules evaluation can open several tasks, or in other words, generate several task opening events. Evaluation collects these task opening events and returns them when it terminates.

4. When rules evaluation terminates, operation *executeGenericQuery* terminates and Active XML returns result of the query to the ASC system.

5. ASC system asynchronously invokes corresponding web service for each task opening event and waits for service termination without blocking other processing.

6. When invoked web service terminates, ASC system creates new incoming event corresponding to task termination adds it into processing queue.

5.3 User interface

The main purpose of the user interface in our prototype is to demonstrate that the implemented core system works. This means that it must show how the artifacts progress through their lifecycle in response to new incoming events so that the user can check that this progress correctly matches the expectations according to the model specification.

We note here that the core system could actually exist without any user interface, because it is able to communicate with outside world exclusively via web services which provide operations to submit new workflow events and to query the status of the artifacts. Knowing this we could test the system with some web service testing tool, for example SoapUI, by first invoking the operation to submit new workflow event and then invoking the operation to query the artifacts status and check the result. However the user interface is easier to work with on the fly, is more readable and provides better demonstration.

We work on the assumption that an incoming event usually updates some artifact data, fires some rules and produces new outgoing events, like task opening events. For these reasons the user interface is primarily designed to have three main functionalists:

- **Display the current artifacts status**. Show the list of artifact instances and for each show artifact name, data attributes and their values, achieved milestones, active stages, enabled human tasks and enabled incoming one-way messages.
- **Display the history of events and fired rules in the system**. Show the list of events that occurred in the system and for event each show the rules that were fired during the processing of this event.
- **Post new incoming events to the system**. Enable the user to submit new events corresponding to the human task response, create call or incoming one-way message. For each possible event create a form that has a structure matching the event payload.

5.3.1 How to create forms

The problem with forms for incoming events is caused by the fact that the system is generated from the model specification so the forms depend on the model. Suppose that we have in model one PSM schema for each possible incoming event. This means that we can either:

1. Create all forms by hand to suit the model specification. This is very hard to maintain, because when the model changes we must not forget to propagate every single change from the model to the forms.
2. Generate the HTML forms directly from model specification together with the rest of the system. This is easy to maintain, because when the model changes, the forms will be regenerated.
3. Generate the forms on the fly by the user interface from xml schemas. This is easy to maintain, because when the model changes, the xml schemas will be regenerated.

We have chosen the option 2, to generate forms directly from the model, because it is more efficient than option 3 and there is no need to generate forms on the fly. The form generator creates only simple forms composed from inputs with type ‘text’, because this was sufficient for the examples.

5.3.2 How to extract data from forms

The user interface communicates with the core system via web services, so when the user submits the form for an event, then before calling the web service we must transform the data from the submitted form to the XML data that represents the message for the web service. This XML data must also match corresponding XML schema.

When the user submits the form the data are stored in the request parameter map. We can create the XML data from this map if we use a suitable naming convention for the parameters, which allows us to determine the original hierarchy of the nodes.

The convention is: The name of the input element is composed from the values separated by dot and a number at the end. Each value from the left to the right represents name of associations on the path from the root to the text node that this element represents. For example:

```
<data>
   <person>
      <firstname>...</firstname>
      <lastname>...</lastname>
   </person>
   <address>
      <city>...</city>
   </address>
</data>
```

(a) XML representation that needs to be created from the form

```
data.person.firstname.1
data.person.lastname.2
data.address.city.3
```

(b) Input names representation to enable later XML contruction

The order number is very important, because without it we could not reconstruct the XML to match XML schema in case where ordering is important between siblings.

6. Exolutio modeling analysis and design

We now know how to represent GSM model using XML and Active XML framework. We must now analyze how to represent the GSM model in eXolutio and how to generate it into our executable model. First we show on a concrete example how we can define information models of business artifacts using conceptual XML modeling, which is currently supported in eXolutio. During this example, we will point out the GSM modeling concepts that eXolutio cannot express nowadays and that we will need to add later. Mostly, these are concepts related to lifecycle models of business artifacts, since eXolutio is currently merely data-oriented. So by extending eXolutio with additional features that would support these concepts, we would allow the user to model business artifacts and use XML formats for their representation, while taking advantage of conceptual XML modeling.

6.1 GSM concrete example

We now show a concrete GSM example borrowed from [30] and slightly modified. This example models process of purchasing a product. The process starts when customer sends an order to the manufacturer. This order contains information about one concrete product that the customer wants to buy. When manufacturer receives the order, he researches individual items that compose the product, creates a work order artifact and assigns all researched items to the work order. Because each item has its supplier, manufacturer creates multiple manufacturer orders, where each one groups all items that belong to the same supplier. Then he sends the material orders to the suppliers. Suppliers then ship the items and send material orders back to manufacturer. When all material orders are delivered, manufacturer can assemble the product and ship it to the customer along with an invoice. During entire processing before delivering the product, the customer can cancel his order. If he cancels after the delivery, the product is delivered and customer must pay full price. Also supplier can reject a material order before delivering the items. Then manufacturer must process rejected items again, create new material order for alternative supplier and send this order to him.

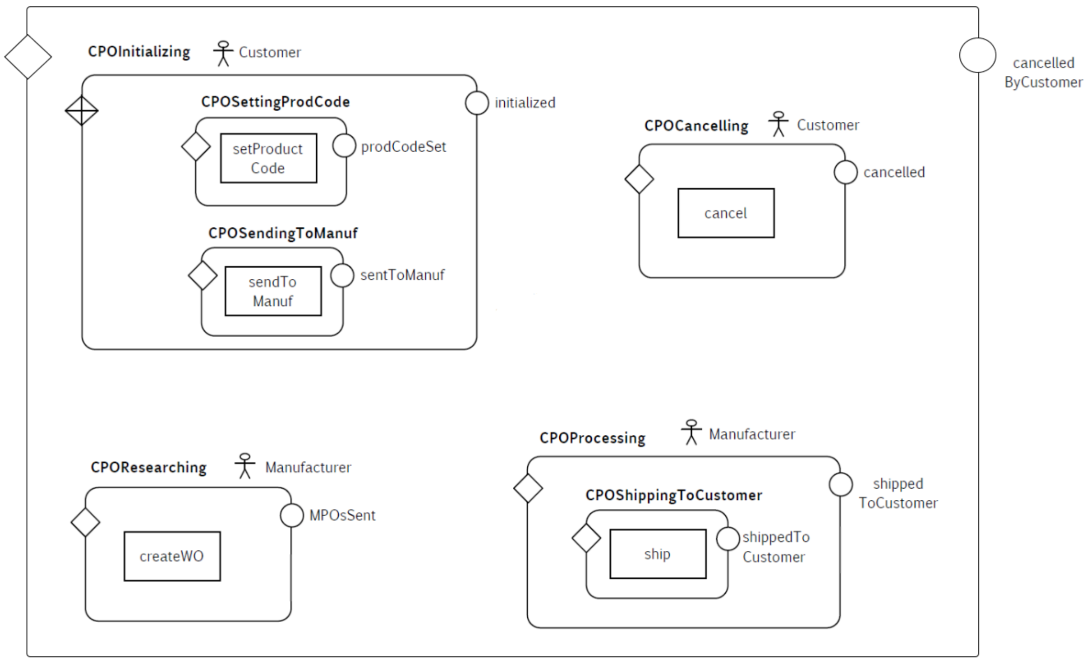


Figure 6.1: Customer order lifecycle model (source [30]

1. ***guard****(CPOInitializing):* ***on*** *CreateCall.onEvent()*
2. ***guard****(CPOSettingProductCode):* ***if*** *not(initialized) and not(productCodeSet)*
3. ***guard****(CPOSendingToManufacturer):* ***on*** *productCodeSet.achieved()*
4. ***guard****(CPOCancelling):* ***on*** *EventCancel.onEvent()* ***if*** *not(shippedToCustomer) and initialized and not(cancelled)*
5. ***guard****(CPOResearching):* ***on*** *initialized.achieved()*
6. ***guard****(CPOProcessing):* ***on*** *workOrder.assembled.achieved()*
7. ***guard****(CPOShippingToCustomer):* ***on*** *EventShip.onEvent()* ***if*** *workOrder.assembled*
8. ***achieve****(sentToManufacturer):* ***on*** *SendToManufacturer.completed()*
9. ***achieve****(productCodeSet):* ***on*** *SetProductCode.completed()*
10. ***achieve****(initialized):* ***on*** *sentToManufacturer.achieved()*
11. ***achieve****(MPOsSent):* ***on*** *Ship.completed()*
12. ***achieve****(shippedToCustomer):* ***on*** *shipped.achieved()*
13. ***achieve****(cancelled):* ***on*** *Cancel.completed()*
14. ***achieve****(customerConfirmed):* ***on*** *SendConfirmation.completed()*
15. ***achieve****(initializingCancelled):* ***on*** *Cancel.completed()*
16. ***achieve****(researchingCancelled):* ***on*** *Cancel.completed()*
17. ***achieve****(shippingCancelled):* ***on*** *Cancel.completed()*

```
customerOrderID: int
productCode: string
manufacturerID: int
customer: record(
      firstname: string
      lastname: string
      shippingAddress: record(
            city: string
            postcode: string
            street: string
            number: string
      )
)
```

Figure 6.2: Customer order information model (data attributes only)

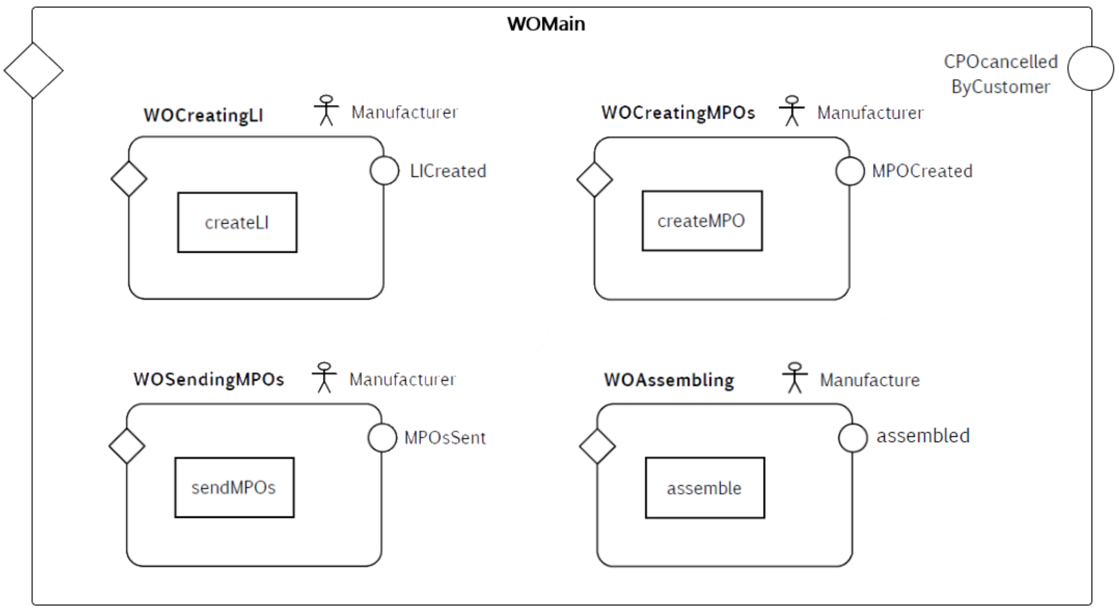


Figure 6.3: Work order lifecycle model (source [30]

1. ***guard***(*WOAddingLI*): ***on*** *CreateCall.onEvent()*
2. ***guard***(*WOCreatingLI*): ***on*** *EventCreateLI.onEvent()* ***if*** *not(WOFilled)*
3. ***guard***(*WOCreatingMPOs*): ***on*** *EventCreateMPO.onEvent()* ***if*** *LICreated*
4. ***guard***(*WOSendingMPOs*): ***on*** *MPOsCreated.achieved()*
5. ***guard***(*WOAssembling*): ***if*** *materialOrder->forAll(m | m.shippedToManufacturer) and materialOrder->size() > 0*
6. ***achieve***(*LICreated*): ***on*** *CreateLI.completed()*
7. ***achieve***(*WOFilled*): ***on*** *CreateMPO.completed()*
8. ***achieve***(*MPOsCreated*): ***on*** *CreateMPO.completed()*
9. ***achieve***(*MPOsSent*): ***on*** *SendMPOs.completed()*
10. ***achieve***(*assembled*): ***on*** *Assemble.completed()*

11. ***invalidate***$(MPOsCreated)$: ***if*** *lineItem->exists(l : LineItem | l.MPORejected)*

12. ***invalidate***$(MPOsSent)$: ***if*** *lineItem->exists(l : LineItem | l.MPORejected)*

```
workOrderID : int
customerOrderID : int
manufacturerID: int
lineItems: collection(int)
materialOrders: collection(int)
```

Figure 6.4: Work order information model (data attributes only)

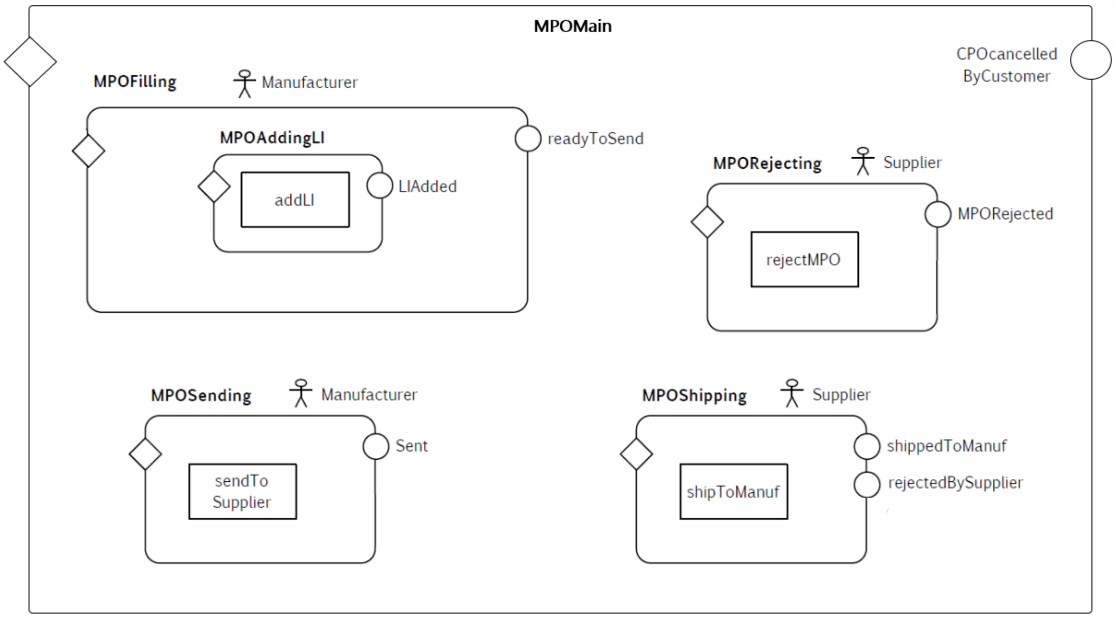


Figure 6.5: Material order lifecycle model (source [30]

1. ***guard****(MPOFilling)*: ***on*** *CreateCall.onEvent()*
2. ***guard****(MPOAddingLI)*: ***on*** *EventAddLI.onEvent()* ***if*** *not(readyToSend)*
3. ***guard****(MPOSending)*: ***on*** *readyToSend.achieved()*
4. ***guard****(MPOShipping)*: ***on*** *sent.achieved()*
5. ***guard****(MPORejecting)*: ***on*** *EventRejectMPO.onEvent()*
   if *sent and not(shippedToManufacturer)*
6. ***achieve****(LIAdded)*: ***on*** *AddLI.completed()*
7. ***achieve****(readyToSend)*: ***on*** *parent.SendMPOs.completed()*
8. ***achieve****(sent)*: ***if*** *lineItem->forAll(l | l.toLineItem().sentToSupplier)*
9. ***achieve****(shippedToManufacturer)*:
   if *lineItem->forAll(l | l.shippedToManufacturer)*
10. ***achieve****(MPORejected)*: ***on*** *RejectMPO.completed()*
11. ***achieve****(rejectedBySupplier)*: ***on*** *MPORejected.achieved()*

```
materialOrderID : int
workOrderID : int
supplierID: int
lineItems: collection(int)
```

Figure 6.6: Material order information model (data attributes only)

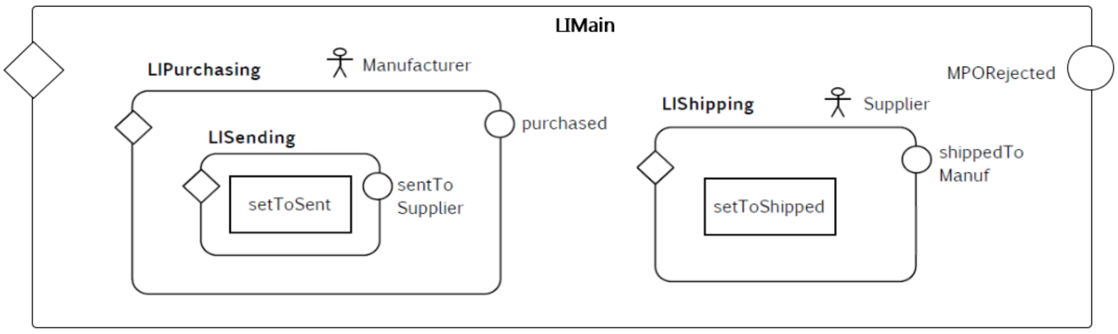


Figure 6.7: Line item lifecycle model (source [30]

1. ***guard**(LIPurchasing):* ***on*** *CreateCall.onEvent()*
2. ***guard**(LISending):* ***on*** *materialOrder.readyToSend.achieved()*
   if *not(purchased)*
3. ***guard**(LIShipping):* ***if*** *purchased and not(shippedToManufacturer)*
4. ***achieve**(sentToSupplier):* ***on*** *SetToSent.completed()*
5. ***achieve**(purchased):* ***on*** *sentToSupplier.achieved()*
6. ***achieve**(MPORejected):* ***if*** *materialOrder.MPORejected*
7. ***achieve**(shippedToManufacturer):* ***on*** *SetToShipped.completed()*

```
lineItemID : int
workOrderID : int
materialOrderID: int
componentCode: string
```

Figure 6.8: Line item information model (data attributes only)

Now when we have a concrete representation in the GSM model, we can create analogous representation in eXolutio. We start with the information model, because eXolutio is data-oriented, which means that transforming an information model into it will be naturally simpler than lifecycle model. The overall modeling process that we want to achieve is depicted in figure 6.9.

6.2 Artifact information model to PIM schema

We can follow steps described in section 4.4. So, we first define a PIM schema based on our GSM model example. It can look for example like schema in figure

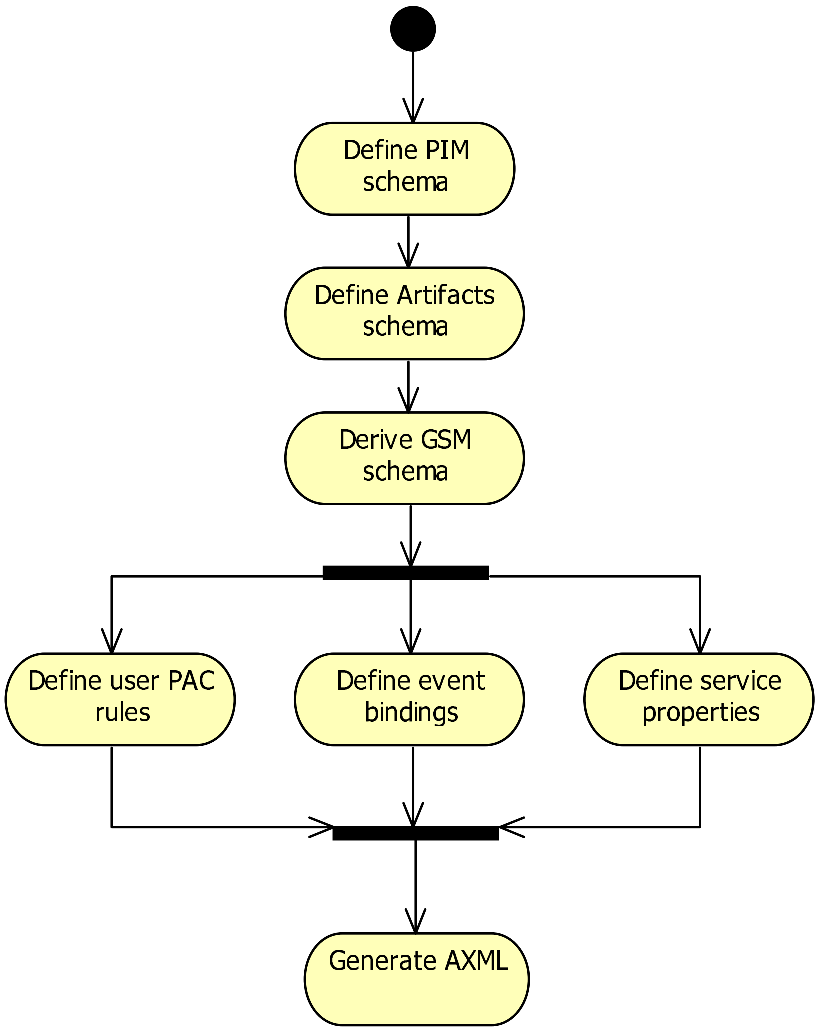


Figure 6.9: Creating Guard-Stage-Milestone model process

6.10. Notice that our conceptual schema contains not only classes for artifacts, like customer order or work order, but also classes for record attributes of these artifacts, like customer and address. This is because we define real-world concepts in the PIM schema and both customer and address are real-world concepts. These concepts are completely independent on a target platform and actual implementation.

Algorithm *Artifact schema to PIM schema.* Let S be an artifact schema.

- For each artifact type R in schema S, create new PIM class $Cpim_R$ in PIM schema.
- For each relation between two artifact types R_1 and R_2, create new association between corresponding PIM classes $Cpim_{R1}$ and $Cpim_{R2}$.
- For every primitive **data** attribute in artifact type R, create new PIM attribute $Apim_A$ in corresponding PIM class $Cpim_R$.

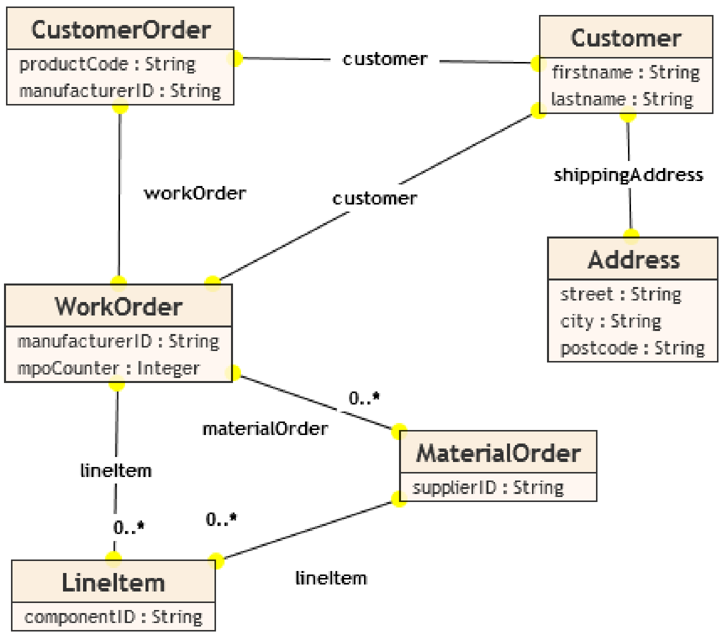


Figure 6.10: PIM schema for the example GSM information model

- For every structured **data** attribute type A that occurs in artifact types R_1, R_2, ..., R_n, create one PIM class $Cpim_A$ and associate it with corresponding PIM classes $Cpim_{R1}$, $Cpim_{R2}$, ..., $Cpim_{Rn}$. For every primitive type in this structured attribute, create new PIM attribute in class $Cpim_A$. For every structured attribute, follow the same procedure, but append new PIM class as a child of class $Cpim_A$.

6.3 Artifact information model to PSM schema

In a next step, we have to define how our artifacts will be represented in a target platform, which in our case is XML. Therefore we must define one or more PSM schemas to represent concrete XML formats for our artifacts. But this is where some questions arise.

Problem *PSM schema partition* Should we have a separate PSM schema for every artifact type, so that every artifact type would have its own XML format ? Or should we have a single PSM schema for an entire artifact schema, so that all artifacts would share only a single XML format?

An answer to this question is related to a correspondence with lifecycle modeling, so we will reason about it later. For now, let us suppose that we choose a single PSM schema for an entire artifact schema. We call this special PSM schema an *APSM schema*. This schema defines how conceptual entities from PIM schema are structured into PSM classes and what are the associations between them. In this schema we also declare which PSM classes represent artifacts and which classes are plain PSM classes.

Figure 6.11 shows one of many alternatives how to define the PSM schema based on the PIM schema. We say 'one of many alternatives', because it is completely up to the designer how the concrete XML format should look like, since the derivation process is not automatic for the reasons that we have already described in chapter 4. Designer defines the PSM schema manually and eXolutio only assists him. We will call this PSM schema for artifacts information model an *APSM schema*.

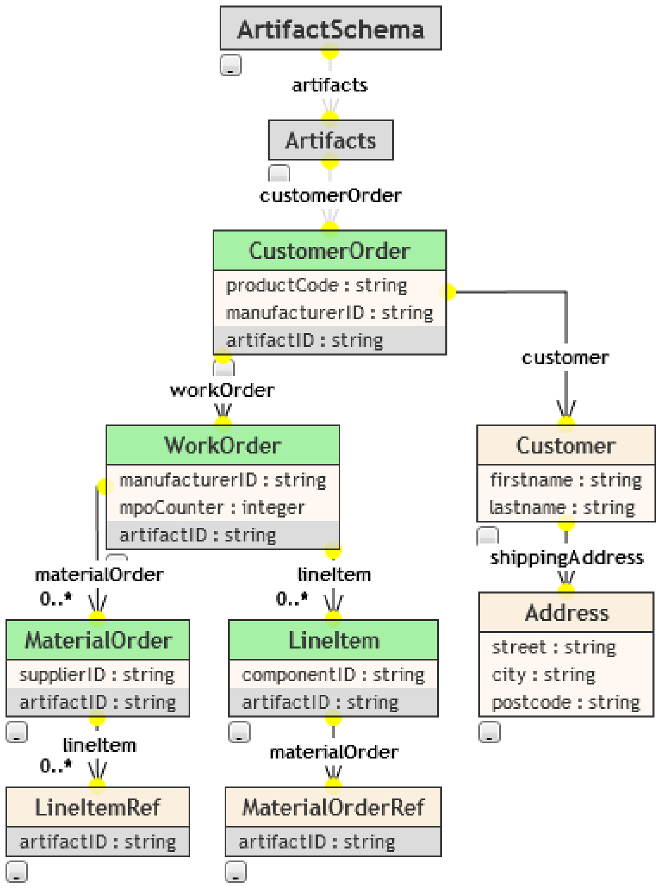


Figure 6.11: PSM schema for the example GSM information model, based on PIM schema from figure 6.10. This schema exploits XML data hierarchy to express parent child relationships.

PSM classes with green header represent artifacts, while other classes represent 'merely' artifacts data. We can see that every artifact type has one concrete PSM class that represents artifact starting boundary in the document tree. We will call this an *artifact PSM class*, or APSM class for short. This APSM class unambiguously marks where the artifact is located in the document. Secondly, this APSM class can have many other child PSM classes that represent artifact structured data. For example, customer order artifact has a single PSM class with green header and one child PSM class for customer record attribute and another grandchild PSM class for address record attribute. Along with PSM classes for data attributes, an artifact class can have another child APSM classes for child artifacts.

We will declare some PSM class to be an APSM class by assigning new lifecycle model to this PSM class. From that moment this PSM class will be an APSM class and will represent an artifact in the final system. A structure of an artifact related to this APSM class is composed from this PSM class and all descendants of this

PSM class except those that are themselves APSM classes and their descendants. If some APSM class A has another APSM class B as an descendant, then class B is represents a child artifact of the artifact A and all its descendants are part of child artifact B.

We can now think about another question when looking in the PSM schema in figure 6.11. The example GSM model contains two artifacts that both have an association with the third artifact. Speaking concretely, both the *work order* and *material order* artifacts contain an association with the *line item* artifact. This is not a problem in the GSM model, since both parent artifacts contain only ID of child *line item* artifact. On the other hand, XML representation supports artifact hierarchy, so one artifact can entirely contain another child artifact. The difference is that in the GSM model, both *work order* and *material order* artifacts have shared association with *line item* artifact (**aggregation**), while in our XML representation, they have exclusive association with the *line item* artifact (**composition**). Clearly, the *line item* artifact can be completely contained in at most one parent artifact only. We have following options:

1. Duplicate child artifact data in every parent artifact. We note here that we would have to duplicate not only artifact real business data, but also status attributes for milestones and stages. This is very inefficient both in space and artifact evolution, because every change would have to be done multiple times. Also, it would break an attempt to efficiently maintain unique identity for each artifact. For example, if we wanted to use xsd:id type for an artifact identifier, we would immediately face the problem, because values of this type must be unique in an entire XML document. So, if we had multiple representations for the same artifact instance in the document, all with the same ID, we would have to use a different type. An advantage of xsd:id type however, at least in eXist database, is that values of this type are automatically indexed, which provides very fast way to look-up corresponding element [31]. Since we evaluate PAC rules using XQuery over the document, fast lockup is surely important.

2. Do not use child artifacts at all, so that every artifact would stand alone and other artifacts would only have its ID, like in the GSM model. This approach however omits an advantage of XML data hierarchy and tends to be rather a XML representation of a relational model. This approach is shown in figure 6.12.

3. Permit only one child relationship for every artifact type. So, if one artifact instance needs to be shared between multiple parent artifacts, only one parent artifact would contain child artifact representation and another parent artifacts would only contain the child artifact ID to reference the child artifact. This way we keep an advantage of XML data hierarchy while still being able to represent child artifacts. This approach is shown in figure 6.11.

   We can see that there is only one APSM class for the *line item* artifact that has a complete representation. This is the APSM class `LineItem`. On the other hand, the APSM class `MaterialOrder` has only reference representation in the child PSM class `LineItemRef`. This PSM class has the same PIM interpretation as the PSM class `LineItem` and it contains an identifier for the artifact instance that it refer to. Using this representation means that the *material order* artifact has the child artifact *line item* that is defined in different place in the document with the same identifier.

In OCL expressions, we can then refer from the PSM class `MaterialOrder` to the PSM class `LineItem` using following expression:

context *MaterialOrder* ***inv:***
self.lineItem.toLineItem().componentID = '5'.

This feature is already implemented in eXolutio. It requires that both `LineItemRef` and `LineItem` PSM classes must have the same not null PIM interpretation, which can be easily satisfied, because there is no reason for `LineItemRef` to have different PIM interpretation. Referencing is represented in generated XQuery using join on the PSM attributes that both PSM classes have the same. Therefore there must be at least one such PSM attribute. In our case, it is straightforward to use for this `artifactID`.

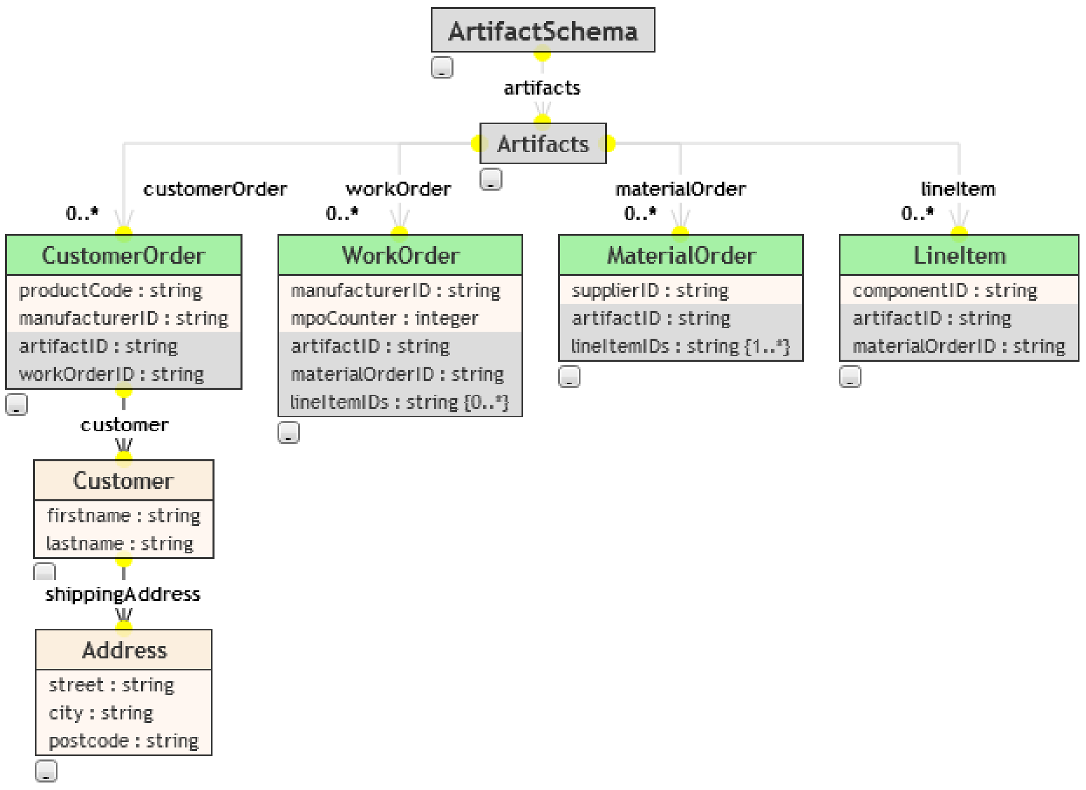


Figure 6.12: PSM schema for the example GSM information model, based on PIM schema from figure 6.10. This schema does not exploit XML data hierarchy and individual artifacts stand alone.

As we said, concrete XML representation of artifacts information model depends on designer consideration. Therefore, there is no direct algorithm to derive the PSM schema for artifact information models. We must only ensure that all PIM classes, associations and attributes will be represented in the APSM schema as well, or in other words, that we do not lose any information. We do not strictly define how individual primitive attributes and structured attributes should be represented. We only require that every APSM class has the PSM attribute `artifactID` and that every artifact has only one complete representation.

We successfully represented the artifact information models in the PIM and PSM schema and therefore also defined the XML format for the information models. We can now use eXolutio generation feature to *automatically* generate XML Schema from the PSM schema, as well as sample XML documents. But what we need next is realization of GSM lifecycle model. Lifecycle model contains stages, tasks, milestones, guards and sentries, but none of these concepts currently exists in eXolutio, so we have to add new features to support them.

6.4 Stages, milestones, tasks and events

In this section we will analyze how to represent stages, tasks, guards and milestones. Stages and milestones require two representations: first they must be specified within lifecycle model that shows artifact stages, stages hierarchy, milestones and association of milestones to stages. But stages and milestones must be also represented in the information model, using status attributes that record for each stage whether it is active and when this activity last changed and for each milestone whether it is true and when its value last changed. In contrast, guards do not need to be represented in information model, because there are no status attributes that correspond to them.

For stages and milestones, there are several options how to add them in the PSM schema.

Explicit definition We can let designer define status attributes on the PSM classes explicitly. This means that designer itself would add the status attributes on the PSM classes. This has a major drawback - anytime designer changes the lifecycle model, he must manually propagate the changes into the information model and change status attributes as well. The same applies for the opposite direction - when designer changes stage or milestone in information model, he must manually propagate the changes into lifecycle model. This results in needless work and possible errors.

Design time propagation We can let designer define stages and milestones only in the lifecycle model and propagate respective status attributes to the information model. So anytime user changes milestone or stage in the lifecycle model, this change will be automatically and immediately propagated to the information model. This solves problems with *explicit definition* approach in one direction. But there is still the problem with opposite direction, when designer changes status attribute in the information model. We could prevent this by making these status attributes to be read-only, so that designer could not change them and therefore no changes would need to be propagated from the information model. This approach needs to make changes and additions to current PSM modeling implementation, to support read-only attributes and propagation between lifecycle and information model. The propagation itself is partially implemented - it is similar to relation between PIM and PSM concepts, where for example a PIM class can have a PSM class interpretation in a PSM schema and changes in the PIM class are propagated to the PSM class.

Generation time propagation Previous approach propagated changes from the lifecycle model immediately. Similar approach is to propagate changes in generation time. This means that change in information model is not propagated immediately once it occurs, but it is dynamically added later during generation process by inspecting defined lifecycle model. Therefore status attributes will be contained in the result, but will not be visible in the information model during design time. This might not be a problem however, since status attributes are still contained in the lifecycle model, so it is clear what milestones and stages an artifact has. An advantage of this approach is that there is no need to implement immediate propagation between the lifecycle and information model and there is no need to implement concept of read-only attributes. On the other way, designer cannot reason about status attributes by looking only at the information model and must inspect the lifecycle model as well.

Lifecycle model must include specification of guards, milestones, stages, tasks, milestones and sentries. Since guards are only a sentries without name, we can represent them only using rules. We must now think if it is possible to represent the lifecycle model using PSM concepts. Lifecycle representation must satisfy following requirements:

1. Specify milestones, stages, tasks and guards.
2. Specify for each stage which stage is its parent, if any.
3. Specify for each milestone which stage owns it.
4. Specify for each task which stage owns it.
5. Specify for each guard which stage owns it.

Stage hierarchy can be seen as a forest of stages. Similarly, a PSM schema is a forest of PSM classes. This can suggest a naive approach to use a PSM schema to represent artifact lifecycle. Stage is represented as a PSM class. All milestones and tasks that this stage owns are represented as attributes of the class. Child stages are represented as child PSM classes. Guards need not to be represented, because they are represented within rules. This way we get an ability to express all mentioned requirements, because PSM classes naturally support hierarchy and attributes ownership. Also, we do not have to implement new visualization.

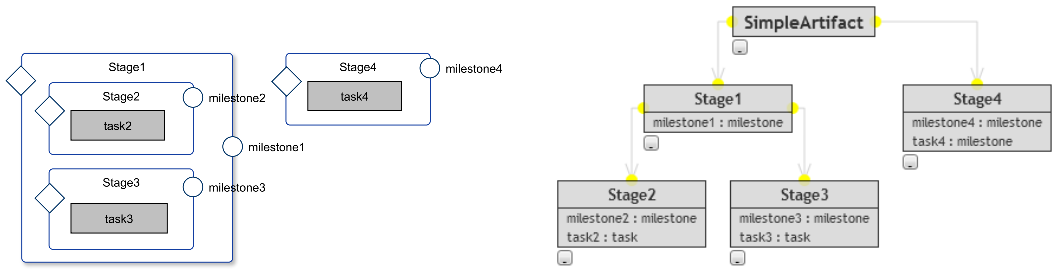


Figure 6.13: A naive approach to model GSM lifecycle

This naive approach has several disadvantages. Many constructs in PSM schema have no sense in GSM schema, for example content models, generalization, association cardinality and many others. Also, a stage although modeled as PSM class is naturally not an interpretation of some PIM class. The same holds for milestones and tasks. Therefore, several commands and features related to PSM modeling are needless for GSM modeling. Moreover, this approach is not easily extensible, because adding new features would require changing PSM schema implementation. Therefore, it is better to provide new modeling capabilities to model lifecycles.

6.4.1 Adding GSM schemas

We have decided to add new modeling capabilities to model GSM lifecycles. Since reusing PSM schemas has some advantages, we could create modeling that would reuse similar concepts - class hierarchy, associations between classes and visualization of these concepts. This means that the visualization will not be exactly the same as in original GSM meta-model, but it would be able to express the same semantics.

Illustration of proposed GSM schema is shown in figure 6.14. Rectangles with rounded corners are stage classes that represent stages and plain rectangles are task classes that represent tasks. Associations between two stage classes and between stage classes and task classes correspond to stage hiearchy. Stages can be either composite and have one or more child stages, or atomic and have only a child task. Tasks must always have exactly one parent stage. Milestones are represented as stage class attributes. Only guards are missing, because they will be specified using sentries. This representation along with sentries fully implement lifecycle specification.

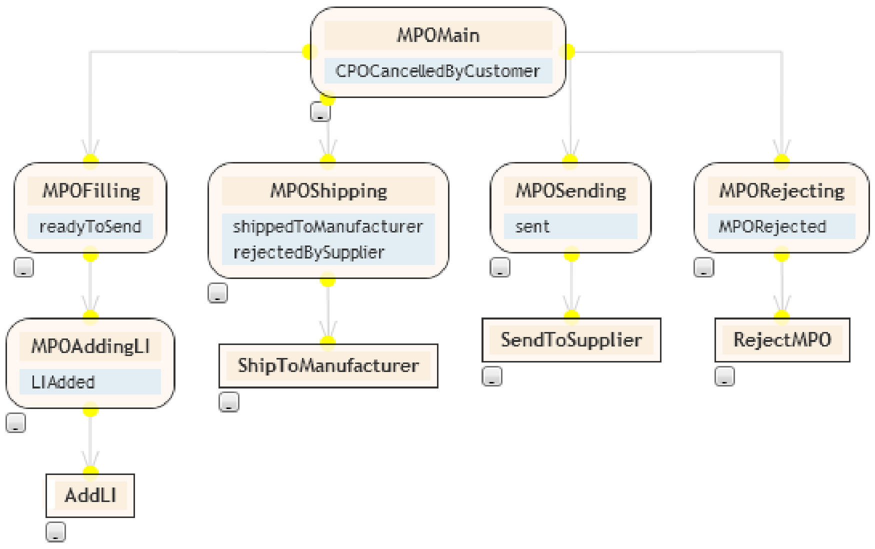


Figure 6.14: GSM schema for material order from the example GSM information model

Algorithm *Lifecycle model to GSM schema.* Let S be an artifact schema.

- For each artifact type R from schema S, create new GSM schema $Sgsm_R$.
- For each artifact stage S from artifact type R in schema S, create new GSM stage class $Cgsm_S$ in the GSM schema $Sgsm_R$.
- For each child stage S_{child} of stage S_{parent}, represented by the GSM stage classes $Cgsm_{Sparrent}$ and $Cgsm_{Schild}$, create new GSM association from the GSM stage class $Cgsm_{Sparrent}$ to GSM stage class $Cgsm_{Schild}$ in the GSM schema $Sgsm_R$.
- For each milestone M of stage S represented by GSM stage class $Cgsm_S$, create an GSM attribute $Agsm_M$ in the GSM class $Cgsm_S$ in the GSM schema $Sgsm_R$.
- For each artifact task T of stage S from artifact type R in schema S, create new GSM task class $CTgsm_T$ in the GSM schema $Sgsm_R$.
- For the child task T of stage S, represented by the GSM stage classes $CTgsm_T$ and $Cgsm_S$, create new GSM association from the GSM class $Cgsm_S$ to $CTgsm_T$ in the GSM schema $Sgsm_R$.

6.5 Sentries

We will start with sentries and associated PAC rules. When designing PAC rules modeling, the goal is to preserve as much already implemented functionality from eXolutio as possible while also enable designer to write sentries in the language very close to OCL variation from original GSM materials. We must note here, that we want to allow the designer to write **only sentries, not entire PAC rules**, which means that designer would not write prerequisites and actions, but only conditions for sentries.

We can immediately notice that sentries look very similar to sentry expressions used in eXolutio to define constraints against PSM schemas. Unfortunately, deeper look reveals that these OCL expressions could resemble only one part of the sentry, an if-condition expression, but not an on-condition expression. The if-condition expression is very close to eXolutio OCL expressions, but the on-condition expression has no direct equivalent. Also, there is no **on** keyword and semantics of **if** keyword is different - in OCL if keyword is used as *if A then B else C*. Unfortunately, this means that in current implementation we are unable to represent both on-condition and if-condition together in one OCL expression. Therefore, we have to find a way how to extend eXolutio to support the on-condition expressions and full sentry definition.

At first, we have to decide against which schema the sentries will be defined? Because sentries specify constraints over an information model, it makes an intuitive sense to define them against the APSM schema. On the other hand, sentries also specify constraints on status change events and external events and these are defined in lifecycle model. Because eXolutio currently supports constraint expressions against PSM schemas and we want to build on current implementation, we will specify sentries against the APSM schema and find a way how to reference lifecycle constructs from sentries as well.

We can now finally explain our decision to use only a single APSM schema to define information model of all related artifacts, instead of using individual PSM schema for every single artifact. This is because we always need to specify PAC rules, implemented using OCL constraints over single PSM schema. Since every PAC rule is not generally restricted only to a single artifact, but it can reference all related artifacts as well, we must define all related artifacts in one APSM schema together. When using multiple schemas, we would need to heavily re-implement current OCL expressions capabilities.

6.5.1 Lower abstraction approach

We said that current expressions can specify only if-condition. A direct approach how to solve this problem is to taking advantage of on-condition semantics and define event expressions using separated form. Remember that in section 2.3 we described how an event is incorporated:

External event If sentry has a form *on E.onEvent()*, where E is an external event type, then it is applicable in time t if $mostRecentEventType = E\ and\ mostRecentEventTime = t$

Status change event If sentry has a form *on S.opened()*, where S is a stage name, then it is applicable in time t if $S_{status} = true\ and\ S_{time} = t$

Therefore, we could substitute an external event expression *on E.onEvent()* with *mostRecentEventTime = 'E' and mostRecentEventTime = now()*. Similarly for status change event, we could substitute an expression *on S.achieved()* with $S_{status} = true\ and\ S_{time} = now()$

The problem is, however, where to get the value $now()$. We know that rules are evaluated when incorporating an effect of single incoming event and it always happens in concrete logical time. Therefore, $now()$ is not a static value, but it corresponds to system logical time. To solve this, we could extend modeling and translation of eXolutio expressions to support new construct for $now()$ expression. This actually means only automatically adding the *now()* function to every APSM class prior the OCL parsing and defining XQuery translation for this function. This function would take no argument and return an integer. During XQuery generation, we would need to generate `$now` variable instead. This variable would correspondence to `$now` variable introduced in section 5.1.3.

This approach would walk around the need for *on* and *if* keyword to distinguish between on-condition and if-condition. However forcing designer to use this long syntax every time he wants to express event expression is unnecessarily verbose, error prone and provides lower level of abstraction than original sentries. For example, designer would need to ensure that he always defines this walk-around expression in one piece, which cannot be assured automatically, since it contains two sub-expressions connected with *and* keyword. Also, designer would need to ensure that he does not define more than one walk around expression, because only one on-condition is permitted.

Therefore, it would be better to directly implement support for on-condition expressions and allow user to define on-condition expressions with higher level of abstraction. This would also clearly divide the on-condition expression and if-condition expression of the rule, because on-condition expression starts with *on* keyword and if-condition expression starts with *if* keyword, so unlike in previous approach, there is strict distinction. The goal is to support more abstract on-condition expressions defined in section 2.2.9. On the other hand, lower abstraction approach requires smaller changes in current implementation.

6.5.2 Higher abstraction approach

Current eXolutio implementation uses a parser generator ANTLR to define grammar for reading and parsing OCL expressions. Exolutio specifies lexer that constructs an *Abstract Syntax Tree* (AST for short) and parser that reads it and creates an internal eXolutio representation of OCL expressions. To support our event expressions, we must extend lexer and parser and add new eXolutio classes for internal representation. An advantage of using full event expression form, instead of rewriting, is that we do not have to modify current grammar rules, but only provide new ones, because we do not need to modify functionality of current OCL expressions. We will now illustrate this.

6.5.3 Status change events

Suppose that we have a sentry *on S.opened() if* $\varphi(x)$. We could similarly illustrate this on any other status change event, without loss of generality. We can see that *on* and *if* keywords unambiguously divide the on-condition expression and if-condition expression. The if-condition expression does not need new functionality, since it defines constraints over the information model exactly the same way as plain OCL expression, without any additional constructs. Sure, it also needs to define constraints for milestone and stage status attributes, but if we add these status attributes into the information model, then this would represent no problem. Since both expressions are strictly divided, we can extend existing grammar with new rule that corresponds to this sentry, representing the forms *on S.opened() if* $\varphi(x)$, *on S.opened()* and *if* $\varphi(x)$. Similarly for other status change events.

New grammar rule will reuse existing grammar rule for general OCL expression to specify the if-condition expression. To illustrate this, let *oclExpression* be this existing rule, then new grammar rule can be *sentry: on eventExpression when oclExpression*. We will not use *if* word to specify if-condition, because *if* word is already used in OCL and using it in sentry would require non trivial changes in existing grammar rules. For this reason, it is much easier and convenient to use *when* word instead. This does not create any confusion or impractical notation, because *when* word is often used in this context as well, for example in JBoss Drolls rule based engine [34].

We can now turn to on-condition expression. The on-condition expression is composed from status attribute specification and operation call on this attribute. Operation call represents concrete event type, so it is one of achieved, invalidated, opened, or closed. Status attribute specification refers to a status attribute in information model. Because status attributes are represented in APSM schema as well, the status attribute specification in on-condition is actually a property call OCL expression, which already exists in current OCL implementation, so we can reuse it. We note here that on-condition expressions could be related to status change events of another artifacts, for example child artifacts, so it is not sufficient to consider only artifact within context. Fortunately, if we have a corresponding association in APSM schema between these artifacts, then property call expression can handle this as well.

6.5.4 External events

We have described a solution for status change event expressions. But what about external event expressions? Unfortunately, these events are slightly different from status change events. This is because status change events always correspond to some status attribute and this attribute is always represented in the information model and therefore in the APSM schema as well, so we can use property call expressions to reference these status attributes from sentries. Existence of status attributes in the information model is implied directly from GSM meta-model. On the other hand, external events have no such representation in information model, so we cannot reference them this way. We could solve this by adding these event attributes to the APSM schema as well. Fortunately, if we look in figure 6.15 from paper [7], we can see that authors of the GSM meta-model consider also an alternative variantions of GSM meta-model where event attributes are represented in the information model as well, which can be useful for recording information related to corresponding external events. So, we will include event attributes for external events into information model, into PSM schema. Thanks to this we will be able to reference them from sentries the same way as we reference status change events.

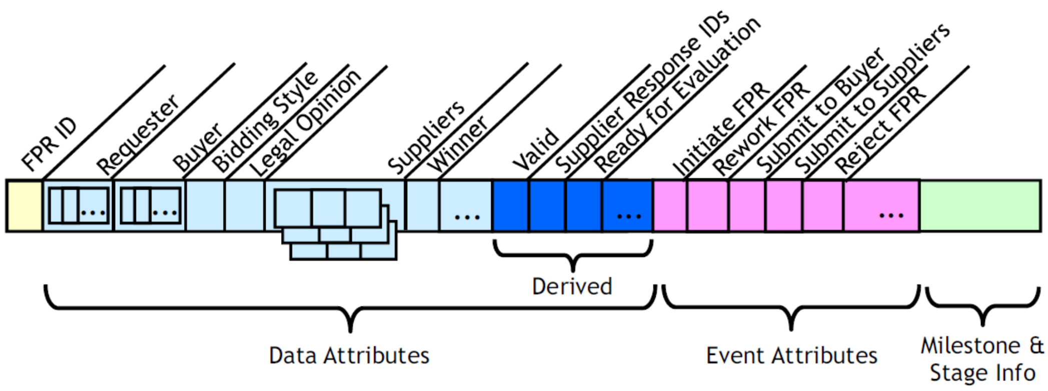


Figure 6.15: Another example of GSM information model (source [7])

Another option how to represent on-conditions for external events is to use an alternative sentries notation instead, used in materials from authors in [13]. There, for an event called $Event_1$, instead of writing $Event_1.onEvent()$, the on-condition is writen as $Event_1()$. Fortunately, this is just a plain operation call OCL expression. With this approach, we would not need to represent event attributes in APSM schema, but similarly as with now() function, only automatically add function for every required event into appropriate APSM class prior to OCL parsing. We have chosen first approach, because it is used in more materials and we do not mind with having event attributes in target XML representation, it is even useful for simpler insight into the artifact evolution, which is an advantage in a prototypical implementation.

6.5.5 Mapping sentries to lifecycle

Described solution allows designer to define conditions for sentries, but we must also associate sentries to individual stages and milestones, or in other words, specifying whether sentry is a guard sentry, milestone achieving sentry or milestone invalidating sentry. We will use a term **gsm rule** to denote triple (sentry, sentry mapping, sentry type). Note that this gsm rule is still not actual PAC rule, because it has neither prerequisite nor consequent part.

We could write sentries directly into the lifecycle model visualization, as comments associated to milestones and guards, as shown in figure 2.2 and 2.3. We call this a visualized form of a sentry. Alternatively, we could define them separately and text only, similarly like current OCL constraints are defined. We call this a stand-alone form of a sentry.

From our experience, visualized form is better with smaller models, because sentries are visually closer to the associated constructs, so it is easy to see concrete conditions for guards and milestones. This approach is however inconvenient for larger models, because greater amount of visualized sentries makes the model less clear. Stand-alone form is in contrast better for larger models, because all related sentries are in one place and in a well arranged way. We conclude that it would be best if a modeling tool supported both approaches and user would decide himself which one to use for each individual guard and milestone. For example, the stand-alone sentries would be used for most of the guards and milestones and user could use the visualized sentries to point out the important ones, or the high-level ones. Therefore, we see this stand-alone sentries as a standard way and visualized sentries as rather a modeling facilitation. For this reason, since we are implementing only a prototype, we will use only stand-alone form for gsm rules.

To associate sentries with guards and milestones, we will use status attributes and we will introduce new special keywords to specify concrete sentry type. We will use keyword *guard* for guard sentries, *achieve* for milestone achieving sentries and *invalidate* for milestone invalidating sentries. This keyword will be followed by the status attribute name to specify concrete guard or milestone. We use stage name for guards here instead of guard name, because as we have said earlier, guards do not have names and since every guard is associated with exactly one stage, it is safe to use the name of the stage instead. This is true even in case when there are multiple guards for one stage, because we do not need to distinguish between individual guards in modeling phase (we will however need to distinguish between them during generation phase when sorting gsm rules).

- **Guard sentry** for stage S of artifact A:
  context A guard(S): on E(x) when $\varphi(x)$
- **Achieving sentry** for milestone m of artifact A:
  context A achieve(m): on E(x) when $\varphi(x)$
- **Invalidating sentry** for milestone m of artifact A:
  context A invalidate(m): on E(x) when $\varphi(x)$

We use a keyword *context* to specify concrete artifact under which the sentry is defined. The *context* keyword defines the base for references within the OCL constraints, which in our case is concrete **APSM** class that corresponds to some artifact. Context can be relevant for multiple OCL constraints, in our case sentries, so using this we provide very similar notation to the notation from original papers, where sentries are also specified within concrete context artifact.

It is important to remind here the difference between an APSM class and plain PSM class. An artifact can be composed from many PSM classes, but there is always one APSM class for each artifact, which is the predecessor to descendant classes that only represent artifact data. This distinction is important, because only APSM class can be used as the context PSM class for sentries related to this artifact, that means that associated sentries have initial constraint context in this class. For sentries, the context class must always be the APSM class.

We will need a new grammar rule to represent gsm rules. This grammar rule will exploit new keywords *guard*, *achieve* and *invalidate*. To specify concrete milestone or stage for which the sentry is defined, we will again reuse an existing property call OCL expression. This expression will reference associated status attribute in the APSM class for context artifact from the information model. Finally, we will need a new grammar rule to represent a constraint context block, which is a single context with multiple associated sentries. Thanks to this, existing OCL grammar rules will not be in a conflict with new grammar rules, since new grammar rules will only reuse existing OCL grammar rules and simultaneously introducing new necessary constructs.

Exolutio sentries example

We can now show an example how to model sentries in eXolutio. Figure 6.16 shows fragment of sentries for customer order artifact.

```
context CustomerOrder

guard(CPOInitializing): on CreateCall.onEvent()
guard(CPOSettingProductCode): when initialized = false and product-
CodeSet = false
guard(CPOSendingToManufacturer): on productCodeSet.achieved()
guard(CPOCancelling): on EventCancel.onEvent() when shippedToCus-
tomer = false and initialized = true and cancelled = false
guard(CPOResearching): on initialized.achieved()
guard(CPOProcessing): on workOrder.assembled.achieved()
guard(CPOShippingToCustomer): on EventShip.onEvent() when workO-
rder.assembled = true
```

Figure 6.16: Sentry definitions in eXolutio for customer order from the example

6.5.6 Event parameters

At this moment we are able to define sentries against an artifact schema, including on-conditions that can refer to status change and incoming event occurrence.

However, an incoming event occurrence can also have associated parameters and what we need is a possibility to refer to these parameters as well. These event parameters constitute an event message and format of this message should be defined in a PSM schema, designed for this purpose. This is exactly in agreement with conceptual modeling, where we have one PIM schema and multiple PSM schemas for associated messages. So, together with one APSM schema for artifact information models, we will define for each incoming event its own PSM schema describing the event message format.

For example, suppose we have an event type *ApproveOrder* with parameters *vote* of type Boolean and *person* of type string. We can imagine format of corresponding input message as a PSM class `ApproveOrder` with two attributes `vote` and `person`. Now, we would like to have a sentry where we would refer to vote and person parameters as in figure 6.17.

context *CustomerOrder* ***guard****(ApprovalStage):*
on *ApproveOrder.onEvent()* ***when*** *ApproveOrderMessage.vote = true and ApproveOrderMessage.person = 'Manager'*

Figure 6.17: Example sentry for event parameters

As we know from our previous analysis, this sentry is an extended OCL expression attached to the APSM schema. The problem is, that this APSM schema knows nothing about the type *ApproveOrderMessage*, because this type is defined in different PSM schema for event message. Strictly speaking, because constraint context starts in a class `CustomerOrder`, the `ApproveOrderMessage` class should have been attached as a child of the `CustomerOrder` class by an association `ApproveOrderMessage`. We can think of two possible ways.

6.5.7 Definition in the APSM schema

We can define the PSM class for *ApproveOrderMessage* directly in the APSM schema, under the PSM class for *CustomerOrder*. Actually, the PSM class would not have to be directly under the PSM class for *CustomerOrder*, since we can adjust the path during parsing, but the idea is the same, having the message format in the APSM schema. We can think of two possible ways how to achieve this:

1. Designer creates an appropriate PSM schema for *ApproveOrderMessage* and then manually adds exactly the same *ApproveOrderMessage* class under *CustomerOrder* class in the APSM schema. In this case however, it makes no sense to have a separate PSM schema for *ApproveOrderMessage*, because designer would have to maintain the message event format in two places. So, designer creates PSM class for ApproveOrderMessage only in APSM schema. The problem is, that this approach significantly increases the size of APSM schema, since every possible event type would have to be defined in APSM schema, so it would be harder to focus on the primary thing that APSM schema should represent, the artifacts information model. Also, this approach slightly breaks the idea of conceptual modeling, where we have

multiple PSM schemas associated to the same PIM schema, since in this case there would be only one PSM schema, our APSM schema.

2. Designer creates an appropriate PSM schema for *ApproveOrderMessage*, but does not add corresponding PSM class to the APSM schema. This class will be added automatically during generation process, prior parsing sentries and will be automatically derived from corresponding PSM schema for event message. Unfortunately, such process is far from being trivial. In case of single PSM class this is easy, it would mean to only clone the PSM class from message PSM schema and add it to APSM schema, but message PSM schema can be generally much more complex. It can contain multiple classes, associations, generalizations, structural representatives and all these together makes the cloning process significantly more difficult. Moreover, some PSM classes from message PSM schema can be already defined in the APSM schema, but have different structure, or in other words, there can be a name collision. We could solve this by cloning the message PSM schema with different namespace. Another option how to automatically add the *ApproveOrderMessage* class to the APSM schema would be to implement new feature that would allow to import one PSM schema into another PSM schema.

We must note here, that this approach also implicates that generated XML document for business artifacts would contain XML data for these event messages as well and similarly generated XML schema would contain definition for these event messages as well. This is inappropriate for two reasons. First, it makes the generated XML Schema more complex and therefore harder to understand and secondly and most importantly, we would not have individual XML Schemas for individual event messages.

6.5.8 Context switch extension

We can define the PSM class for *ApproveOrder* only in the message PSM schema and extend sentry parsing to support references to different PSM schemas. This means switching constraint context inside the expression. This way, we could let the APSM schema to focus only on artifact information models and specify event messages separately, clearly divided from the APSM schema. This would also be more consistent with the idea of conceptual modeling.

For presented reasons, we choose an alternative with parsing extension. Again, our goal is to extend parsing in a way that does minimal changes to current implementation. We would like to have a mechanism to switch current constraint context inside one sentry expression to different PSM class in different PSM schema. We have three important requests for this context switch:

1. The context switch must have clear *start* and *end* boundaries. We must unambiguously specify where the context switch starts and where it terminates, so that we know to what constraint context we map the expressions in any given time.

2. We want to support not only simple equality expressions, like in figure 6.17, but generally all OCL expressions that are otherwise available and make it

possible to use them in the switched constraint context as well. These are for example expressions like *forAll*, *exists* and many others.

3. We want to support comparison between properties from the switched PSM schema and properties from the APSM schema, which means in other words, to support comparison between event parameters and data from schema information model. This is very useful, because we will often need to compare event parameters to current status of schema information model, not only to literal values or other message parameters.

We start with request 1. We can add new keywords to declare switch boundaries, for example *switchStart* and *switchEnd*, but the problem is that this context switch can generally appear anywhere in the OCL expression, so it would require to add this keywords to existing grammar rule for general OCL expression, which is a kind of change that we want to avoid.

Alternatively, we can solve this by using **operation call OCL expression**. Operation call OCL expression accepts list of general OCL expressions as parameters, so we can use these expressions in parameters to define constraints with respect to switched schema and use operation call parentheses as a signal to start and terminate context change. To distinguish whether an operation call represents context change or is an ordinary operation call, we use operation call name. If it corresponds to event name, we switch the context to the PSM schema with the same name and set constraint context to the first child of PSM schema class, which corresponds to event message root class.

Figure 6.18 illustrates this. An entire expression inside the operation call is evaluated with respect to switched context. This operation call returns Boolean type, so it is possible to use it anywhere in the if-condition where ordinary operation call with this return type is permitted. Figure 6.18 illustrates this, where we first define constraints in switched context and the result of this constraint is used as a left operand of an equality expression.

```
context CustomerOrder guard(ApprovalStage):
when ApproveOrderMessage(vote = true and person = 'Manager')
and filled = true
```

Figure 6.18: Example sentry for event parameters with context switch

This alternative does not require a change in existing grammar rules, but only adding extensions to parsing. Unlike changing grammar rules, such change can be easily incorporated and clearly separated, since it can be implemented in different method and called only when explicitly requested (for example, by setting the flag for mode to the parser). It also immediately satisfies request 2 as well.

Request 3 is more intricate. We want to compare properties from the message PSM schema against the APSM schema, so operation call must return concrete type of referenced property from switched context. For example, suppose that we have a data attribute *requiredPerson* in the *CustomerOrder* artifact and we want to specify that value of the person parameter is equal to this *requiredPerson* attribute from information model. Figure 6.19 illustrates this.

```
context CustomerOrder guard(ApprovalStage):
when ApproveOrderMessage(person) = requiredPerson
```

Figure 6.19: Example sentry for event parameters with context switch returning property from switched context

However, returning an arbitrary type from this operation call is not possible, because if returned type was unknown for the APSM schema, then returned property could not be compared with data attribute from the APSM schema. To solve this, we must map returned property type to corresponding type from the APSM schema only if we know for sure that this type exists in the APSM schema. This can be guaranteed only for standard XSD primitive types, because these types will be always the same in both PSM schemas. Moreover, it can be guaranteed for collections of these types. For any other type, we must map returned property type to **xsd:any**.

6.5.9 Invariant preserving rules

Designer defines sentries for stages and milestones and these are used to generate corresponding rules from categories PAC_1, PAC_2 and PAC_3. However, as we know from section 2.3.4, to ensure correct rules evaluation, we must also generate invariant preserving rules from categories PAC_4, PAC_5 and PAC_6. Fortunately, invariant preserving rules can be derived and generated automatically from GSM schemas, because all the necessary information to derive these rules is already contained in associations between stages and milestones. So, designer does not have to write these rules by hand.

We will create OCL representation of these rules directly, because it would be inefficient to parse them from text representation without bringing any advantage. During generation process, we will merge these invariant preserving rules with user defined rules and use this merged set as an input for further processing.

Apart from generating invariant rules, we must also generate preconditions for each rule. These preconditions are necessary as well to ensure correct rules evaluation. Similarly as with invariant preserving rules, these preconditions can be automatically derived and generated from GSM schemas.

6.5.10 PAC rules ordering

We now have the specification for sentries, but these represent only the partial fragment of an artifact evolution. Complete evolution semantics is specified using PAC rules, which can be automatically derived from the sentries in the lifecycle model. Sentries already specify the condition part and the mapping to concrete milestones and stages. When we have this information, we can unambiguously assign the sentry to corresponding PAC rule, this is defined in the rules table in section 2.3.4. So, we can use sentry condition as the condition part of the rule and derive the precondition and action parts of the rule from the lifecycle model and sentry mapping.

These rules must be sorted in the topological order to ensure correct semantics as described in section 2.3.4. As we have described in the executable model analysis, we will later translate these rules into XQuery and also generate the XQuery code that will evaluate these rules, so we must ensure that this code will evaluate individual rules in the correct topological order. We could first translate the rules into XQuery and sort these translated rules later, but it would be unnecessarily harder. Sorting of the rules depends on the properties used in the rules and associations between them, so it is best to do it when we have the rules parsed in OCL representation, so that we can easily traverse through the rules and inspect them. If topological sort of the rules does not exist, then the generation process cannot continue because this means that designer must first correct the rules.

6.6 Incoming event binding

As we said in formal analysis for the GSM meta-model, when an incoming event is incorporated into the artifact schema, the first step is to process an **immediate effect**. This process changes *mostReventEventType* and *mostRecentEventTime* attributes and changes the values of data attributes of directly affected artifact instances. Changing *mostRecentEventType* and *mostRecentEventTime* is straightforward and it always depends only on the event type and current logical time. On the other hand, changing data attributes can require individual strategy for every event type, because event parameters do not generally have to correspond directly to some attributes from the information model. In other words, an event message can have absolutely different structure than any data attribute in the information model.

We described this problem in section 5.1.7 in the context of an executable model. From a modeling point of view, this also corresponds to division between definition of event messages and schema information model, where the schema information model is specified with the APSM schema, while individual event messages are specified with their own PSM schemas. All these PSM schemas and APSM schema are based on the same PIM schema, but they can specify different realization of PIM concepts. Naturally, if we want to generate an executable model from eXolutio, we need to explicitly specify mapping between event parameters and artifact information model, because it must be clear how an incoming event changes artifact data attributes.

In section 5.1.7 we said that in an executable model this binding will be XQuery function, like in figure 5.8. It is only the question how to define this function in the modeling process. We could let the designer to write explicitly this function explicitly for every event type, but it would be very error prone to write binding function for every event by hand. It would be easier to use if designer would merely specify the merging strategy, assign it to the event and eXolutio would then generate appropriate function. Figure 6.20 illustrates an example for incoming binding.

```
<binding event="{event type}" artifact="{artifact type}">
   <definition>
      <replaceValue node="customer/firstname"
         with="CustomerInfo/firstname/text()" />
      <replaceValue node="customer/lastname"
         with="CustomerInfo/lastname/text()" />
   </definition>
</binding>
```

Figure 6.20: Definition for an incoming event binding

The easiest way would be to write the definition directly in XQuery, because eXolutio would only take this definition and use it as a function body. Unfortunately, the version of eXist database used in Active XML distribution does not support standard XQuery Update Facility and uses its own update syntax that is different from the standard one. Although we could use this eXist syntax in the binding definition, we want to shield the designer from actual implementation technologies. For this reason, we will enable the designer to provide update definition in XML that will mirror syntax of standard XQuery Update Facility. This definition will be parsed and transformed into eXist XQuery code. This is much easier task then parsing standard XQuery code and transforming it into eXist code.

User-friendliness We note here that this approach is only for the purpose of our prototype, so that we are able to easily define the binding. In real application, it would be more appropriate to assist the designer with some GUI editor that would automatically obey the PSM schema for message event. This is however outside the scope of this thesis.

6.7 Outgoing binding

Outgoing binding is similar to incoming binding, except that the direction is different - from artifact data to event message. This binding is used when the system generates some outgoing event, for example when task opens that contains two way service call. This service call may contain parameters that need to be set by data from the source artifact.

The situation is little different from incoming binding. In incoming binding we have data in the event message and data in target artifact and we need to merge data from event message into the artifact. But in case of outgoing binding, we have only artifact data, because event message is empty. For this reason we do not use XQuery update for outgoing binding, but rather message definition, where we write the event message with the structure it should have. Data that needs to be inserted into the message are written as XPath expressions that refer to the artifact data. Naturally, defined structure for an event message must conform to the XML Schema generated from corresponding PSM schema for this event message. Figure 6.21 illustrates an example for outgoing binding.

```
<binding event="{event type}" artifact="{artifact type}">
   <WorkOrder>
      <manufacturerID>$artifact/manufacturerID/text()</manufacturerID>
      <customerOrderID>$artifact/artifactID/text()</customerOrderID>
   </WorkOrder>
</binding>
```

Figure 6.21: Definition for an outgoing event binding

Again, the same note with user-friendliness as in previous section applies here. Although in this case, we could at least assist the designer with the binding definition writing, because this binding definition must obey associated PSM schema and therefore associated XML Schema as well, which can be used for binding validation. So, we could first generate all XML Schemas for all outgoing event types and then allow the designer to put binding definitions into separate files, one for each event type and validate these files against corresponding XML Schemas. Moreover, if the designer used some XML editor with on the fly schema validation, it would automatically give him even better guidance with the writing. We will not use this feature in our prototype, but it could be easily incorporated.

6.8 Definition of incoming events

When we want to refer to incoming events occurrences inside sentry expressions, we must first know what is the set of all possible incoming events. This set must be defined somewhere. So far, it is neither in the APSM schema nor in the GSM schemas. However, we said that we will create one PSM schema for every event message type. So, we could use all PSM schemas to define the set of all incoming events. Another possibility would be to use events defined in bindings file. No alternative has clear advantages, so we choose second alternative.

6.9 Specification of service calls

When there is a task that contains call to two-way service, we must have the specification of this service call, otherwise we would have no clue how to execute the task in generated executable model. We need specification of endpoint, namespace and operation for the web service call. This is not a part of the GSM model, but it must be specified if we want to generate the system from the model. We will define these properties along with event bindings and load them prior generation process.

6.9.1 Updating APSM schema

We can now continue with analysis from section 6.4. We said that we need to propagate lifecycle information from GSM schemas to APSM schema. We will use generation time propagation approach, so prior the generation process, we will add new PSM constructs for stages and milestones to relevant APSM classes. We will also add new PSM constructs for event attributes. In case of stages and milestones, we will not add only new PSM attributes for status and timestamp, but we will add entire PSM class that will contain these two attributes. Also, we will not add new classes directly as children classes of corresponding APSM classes, but we will first create wrapper parent classes for them, in order to clearly distinguish between data attributes, status attributes and event attributes.

Algorithm *Updating APSM schema.* For each artifact type R_i, let $Capsm_{Ri}$ be corresponding APSM class from the APSM schema.

1. **Adding status attributes class**. For each artifact type R_i, create PSM class $Capsm - status_{Ri}$ as a child of the APSM class $Capsm_{Ri}$ with association named `statusAttributes`. This class will contain status attributes for stages and milestones associated to artifact type R_i.

2. **Adding stage attributes**. For each stage S_j of artifact type R_i, add new PSM class as a child of the PSM class $Capsm - status_{Ri}$ with association named as the name of the stage S_j and add to this class three PSM attributes named `status` of type Boolean, `update` of type Integer and `stableStatus` of type Boolean.

3. **Adding milestone attributes**. For each milestone M_j of artifact type R_i, add new PSM class as a child of the PSM class $Capsm - status_{Ri}$ with association named as the name of the milestone M_j and add to this class two PSM attributes named `status` of type Boolean, `update` of type Integer and `stableStatus` of type Boolean.

4. **Adding event attributes class**. For each artifact type R_i, create PSM class $Capsm - events_{Ri}$ as a child of the APSM class $Capsm_{Ri}$ with association named `eventAttributes`. This class will contain event attributes associated to artifact type R_i.

5. **Adding event attributes**. For each incoming event type E_j of artifact type R_i, add new PSM class as a child of the PSM class $Capsm - events_{Ri}$ with association named as the name of the event type E_j and add to this class one PSM attribute named `update` of type Integer.

Because every milestone and stage status attribute is contained in the wrapper `statusAttributes` class, we need to take this into account when parsing sentries. This is not a problem actually. Ordinarily, during OCL parsing, when a property path item is encountered, new property call expression pointing to that property is returned. For example, when there is a path:

$$customerOrder.customer.name$$

and parser encounters property path item *name*, it returns new property call expression pointing to property `name` with source expression pointing to its predecessor, in this case `customer` property, which can be again a property call expression. So, in case of milestone and stage related properties, we can return property call expression with one additional path item, pointing to `statusAttributes` class, if we know that parsed property is actually some stage or milestone. Fortunately, this information is already known from GSM schemas. For example, when there is a path:

customerOrder.shipped

and parser encounters property path item *shipped*, which will be property for milestone, it first creates new property call expression pointing to property `statusAttributes` with source expression pointing property `customerOrder` and then returns new property call expression pointing to property `shipped` with source expression pointing to property `statusAttributes`.

The same idea can be used for event attributes. We use this extension for purpose, to demonstrate that concrete milestone and stages XML realization could be independent of sentry definition, if necessary. It is also useful to clearly distinguish between data, event and status attributes in generated XML representation.

We will also add PSM attribute *type* to every status attribute, to add information whether it is attribute for stage and milestone. This is rather for testing purposes and it is not necessary.

6.10 Generating XQuery from PAC rules

When all rules are sorted, we can translate them to XQuery. For each rule we generate one XQuery function that contains the condition evaluation for the rule and also associated actions that must be performed if corresponding rule is fired. Also, we generate the wrapper function that calls these functions in the topological order that was obtained in previous section.

Before we can process to actual translation, we must transform rule expressions from higher abstraction syntax into standard OCL syntax, so that we can use existing implementation for translation OCL expressions into XQuery. This is because rule itself is not standard OCL expression, but it is a composition of individual OCL expressions for rule property mapping, precondition, on-condition and when-condition. This implicates that each rule translation is divided into multiple sub-translations, where each part is translated individually. We must divide rule to these parts:

On-condition On-condition is optional for rule. If rule has no on-condition, then we generate only value $true()$ instead, because empty on-expression is always true. If on-condition is not empty, we must first transform it to standard OCL expression. The form of this expression depends on event kind. If it is status change expression, then we must translate it to two OCL expressions. First expression will compare status of the attribute and second will compare the update time of the attribute. For example, the condition *ShipStage.achieved()* will be replaced with *ShipStage.status = true and ShipStage.update = now()* and then translated.

When condition When condition is standard OCL expression, but still it cannot be immediately translated to XQuery. We must extend milestone and stage expressions to refer to status attribute. For example, consider that we have following expression:

ShipStage = true

This condition states that stage named *ShipStage* must be active. This means that we are actually referring to status sub-property of this stage status property. So we need to generate this in target XQuery code:

xs:boolean(ShipStage/@status) = true()

In order to use standard generation mechanisms from OCL to XQuery, we first extend the original expression to:

ShipStage.status = true

This OCL expression can be directly translated into presented XQuery code. We do this for every sub-expression of the original if-condition where the property is the status property that represents some milestone or stage. We extend each such property to the status sub-property. When we replace all occurrences, we translate the updated expression to XQuery.

Precondition Every rule has a precondition. Precondition is standard OCL expression that could be directly translated. It always refers to either a milestone or a stage attribute of the artifact and has the form *property = value*. This property refers to wrapper status attribute, so it must be extended the same way as described in paragraph for *when condition*.

Action Rule action specifies operations that are performed if rule condition is satisfied, that means, when rule is fired. The set of operations depends on rule category. In every case, the milestone or stage attribute is updated and new event is added to the set of generated events. If rule category is a guard, then also related stage opens and if this stage is atomic, it starts a task that is inside the stage, so operation to start this task is also generated. Starting task means calling function that sets task as active. Similarly, rules PAC_5 and PAC_6 closes related stage and this ends task that is inside the stage, if there is some that is active.

Event guard Event guard is an optimization in rules evaluation. It comes from the observation that we do not need to evaluate each rule for each artifact if we know for sure that the condition will evaluate to false. One case where we can be sure about this is when rule has an event condition that refers to some concrete event and we evaluate the rules with context that does not contain this event. For example, suppose we have a rule:

context Order guard(ShipStage): on EventShip.onEvent()

Rule evaluation is implemented as a for loop over possibly affected artifact instance where on-condition constitue a predicate:

```
for $artifact in doc('/Peer/artifacts.xml')//Order
[xs:boolean(lifecycle/mostRecentEvent/@type) eq 'EventShip' and
[xs:integer(lifecycle/mostRecentEvent/@time) eq $now]
```

Consider that we start rules evaluation due to an incoming event *EventOrder*. Since current event occurrence is *EventOrder*, we know for sure that the condition in the loop evaluates to false for each Order artifact. This is why we can add guard expression that allows evaluating the loop expression only if the event guard condition is true.

6.11 Generation overview

In this section we summarize the generation of AXML system from eXolutio. Figure 6.22 illustrates the overall process.

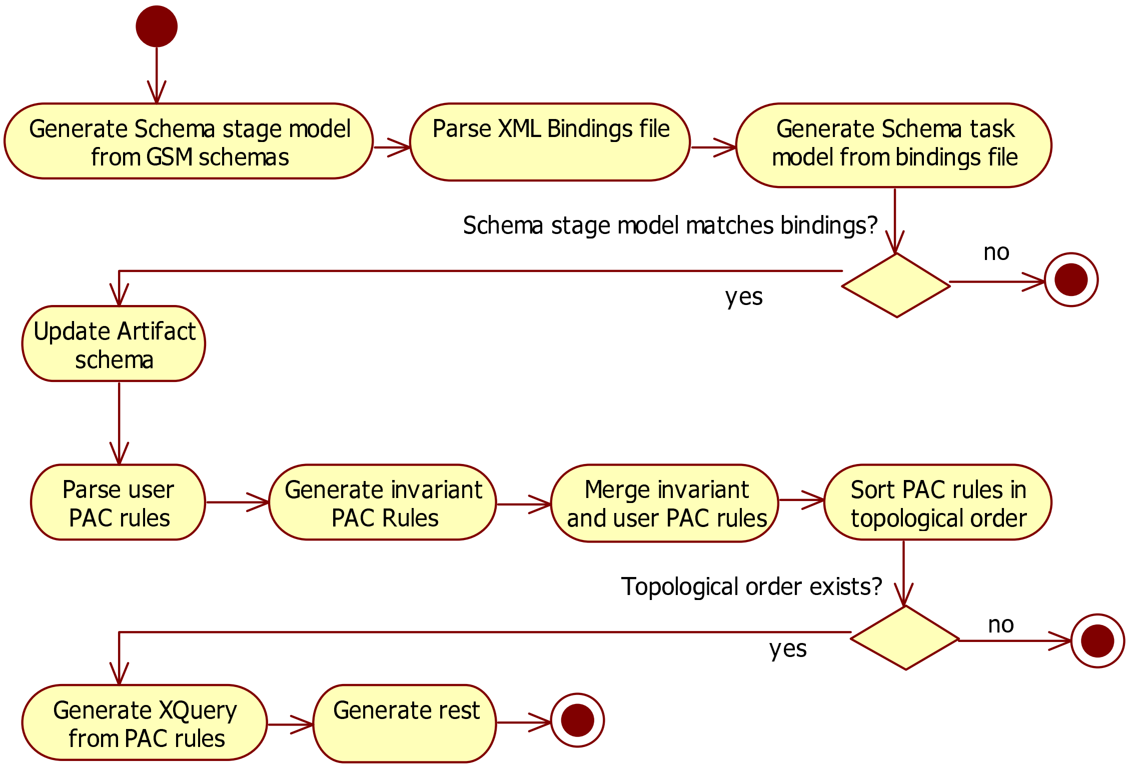

Generate Schema stage model from GSM schemas
Parse XML Bindings file
Generate Schema task model from bindings file
Schema stage model matches bindings?
no
yes
Update Artifact schema
Parse user PAC rules
Generate invariant PAC Rules
Merge invariant and user PAC rules
Sort PAC rules in topological order
Topological order exists?
no
yes
Generate XQuery from PAC rules
Generate rest

Figure 6.22: Generating AXML system from eXolutio

1. **Generate schema stage model**. Processing starts by creating *schema stage model*, an abstract representation that holds all the necessary information about stages and milestones and their associations. This model is derived from GSM schemas and it is used during an entire generation process.

2. **Parse event bindings**. We load and validate the binding file that specifies incoming and outgoing bindings for all event types in the system. It also specifies service properties for tasks that correspond to invocation of two-way service or sending one-way message.

3. **Generate schema task model**. Parsed binding file is used to create *schema task model*, an abstract representation that holds all the necessary information about event bindings and service properties. This schema is a part of schema stage model, because these two are frequently used together. Schema task model must be validated against schema stage model to verify that it contains the same set of artifacts and the same set of tasks. Otherwise this is considered to be an error and generation cannot continue.

4. **Extend artifact schema**. When schema task model and schema stage model are created and successfully validated, we need to extend the APSM schema so that it contains all the necessary information about stages, milestones and incoming events. This is very important because we want to refer to stages, milestones and incoming events from sentries and invariant preserving rules. We update the APSM schema for each stage, milestone and incoming event type. This update cannot be done with eXolutio commands, because each command execution causes view of user interface to repaint and with too many commands the update would be too slow. Instead, we add new items directly to appropriate collections and remove them after the generation.

5. **Generate user defined rules** With updated APSM schema we can parse sentries and create corresponding user defined PAC rules. This parsing will frequently consult schema stage model and schema task model in order to check if currently parsed property refers to some milestone, stage or event attribute or if it is only an ordinary data property.

6. **Generate invariant preserving rules** We generate invariant preserving rules automatically from GSM schemas.

7. **Merge PAC rules** We merge together generated invariant preserving PAC rules and user define PAC rules for further processing.

8. **Sort PAC rules in topological order** We sort PAC rules in topological order to ensure correct evaluation. If such topological order does not exist, we must terminate the generation process.

9. **Generate XQuery from PAC rules** When rules are sorted, we can generate XQuery code for their evaluation.

10. **Generate other files and code** We also generate many other necessary files and code:

    - XML Schemas for event messages from PSM schemas. These schemas are used for validation of incoming event messages in the ASC.

- XML document for initial artifact templates. These templates are necessary when new artifact instance needs to be created and inserted into the business artifacts document. Template file contains initial XML representations of artifact information models, one for each artifact in the model. Initial representations are generated from the APSM schema. For each artifact, generator translates corresponding APSM class into XML, taking into account also all children PSM classes, except those that already belong to some child artifact.
- HTML forms for event messages. These forms are used in the ASC user interface to submit incoming events into the ASC by human performers. Forms are generated from corresponding PSM schemas for event messages.
- Workflow web service and operations for incoming events. These operations are used to receive incoming events in the ASC. For each incoming event type, generator creates one web service operation that accepts corresponding event message as a parameter.
- XQuery utility functions. These functions are used to work with artifacts, for example for creating new artifact, finding concrete artifact, obtaining artifact information and so on. These functions are generated to provide better query performance, because generated functions can take into account concrete element names in target XML representation and element names are automatically indexed.

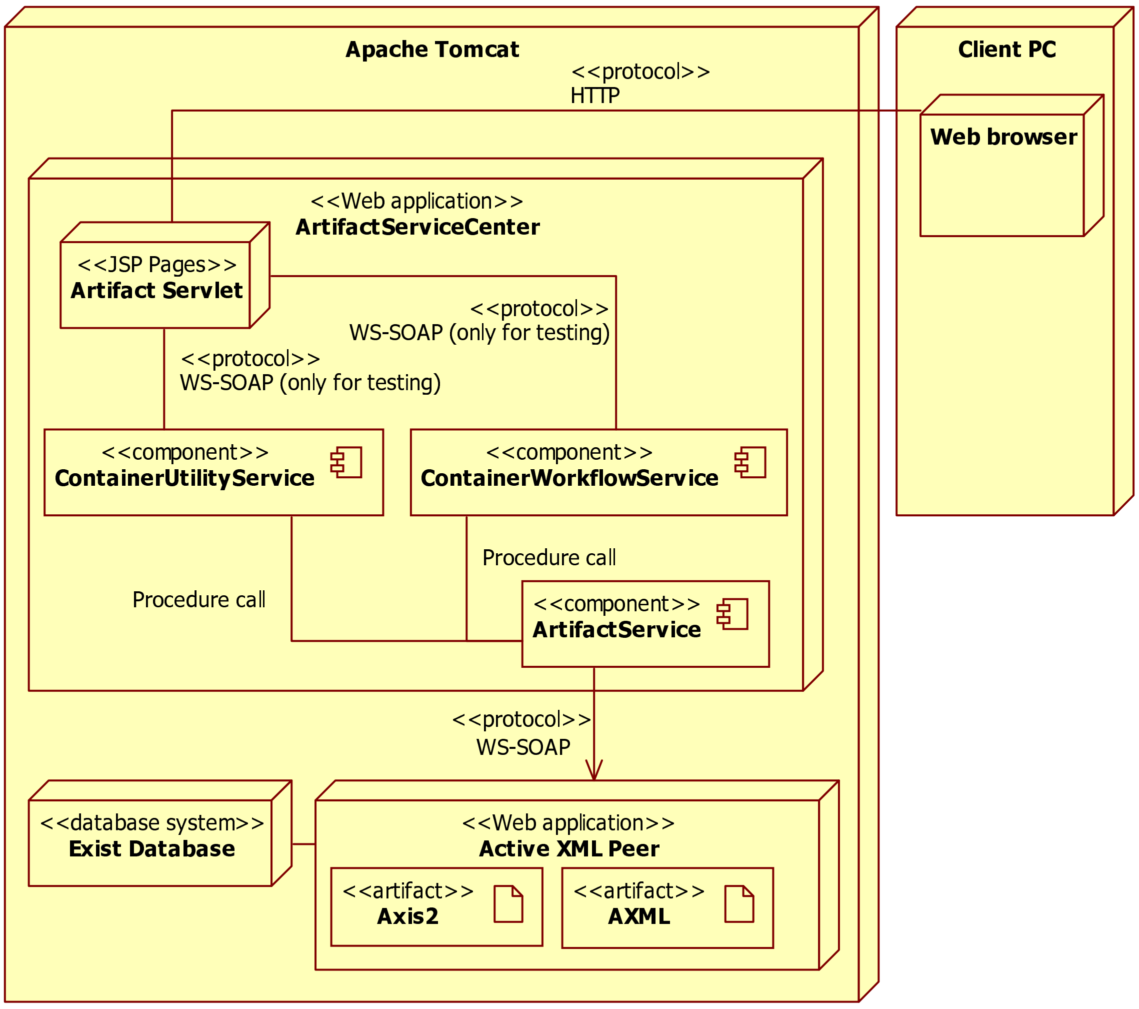
7. Artifact Service Center Architecture

In this section we present the high level architecture of the Artifact Service Center.

7.1 Overview

The architecture of the Artifact Service Center is shown in the figure 7.1. It is based on Active XML architecture, which is made up from a following component:

- Web server, currently Apache Tomcat 5.5.
- XML database, currently eXist.
- Axis2 web service engine.
- Active XML service calls execution engine.



Apache Tomcat
<<protocol>> HTTP
Client PC
Web browser
<<Web application>> ArtifactServiceCenter
<<JSP Pages>> Artifact Servlet
<<protocol>> WS-SOAP (only for testing)
<<protocol>> WS-SOAP (only for testing)
<<component>> ContainerUtilityService
<<component>> ContainerWorkflowService
Procedure call
Procedure call
<<component>> ArtifactService
<<protocol>> WS-SOAP
<<database system>> Exist Database
<<Web application>> Active XML Peer
<<artifact>> Axis2
<<artifact>> AXML

Figure 7.1: Deployment diagram of the Artifact Service Center

We extend this architecture with new web application, *Artifact Service Center*, that implements new functionality and creates wrapper around Active XML. We deploy this application to the same web server as Active XML. These two applications are loosely coupled and ASC does not modify Active XML at all. They see each other as a black-box and communicate only via web services. There are two general rules for communication:

1. ASC never access XML database directly, but always uses Active XML as a mediator.
2. External environment never access Active XML directly, but always uses ASC as a mediator.

The ASC application is composed from three main parts. The most important is the core component, ArtifactManager, that creates a wrapper around Active XML by providing an interface for managing business entities. It also assures correct semantics of operations when managing the artifacts, like events ordering, synchronization and others. Second important part are web services, that provide access to the ArtifactManager from external environment. Third part is user interface that enables human performers to submit their action and sent events to the system.

Communication between ArtifactManager and web services uses direct procedure calls. Different situation is communication between user interface and ArtifactManager. User interface does not communicate directly with ArtifactManager, but it uses web services for communication. It is important to mention, that *this is only for testing purposes* in our prototype, because in real situation the user interface could communicate with ArtifactManager using direct procedure calls, providing that both components are in the same web container, which is a case of our prototype. But we use this style to show that ASC is able to communicate with the external environment exclusively by exposed web services. Communication between ArtifactManager and Active XML is only using web services that Active XML provides in its default implementation.

7.2 Main components

In this section we describe main components of the system.

7.2.1 ArtifactManager

ArtifactManager is the core component of the system. This component works as the mediator between Active XML and the rest of the system by providing an interface for managing business artifacts and their lifecycle. Also, it helps to assure correct semantics of operations when working with business artifacts, like events ordering and synchronization. To assure this semantics, there must always be only one ArtifactManager for each `ArtifactServiceCenter` and Active XML Peer. For this reason, ArtifactManager is implemented as a singleton using the Enum Singleton Pattern.

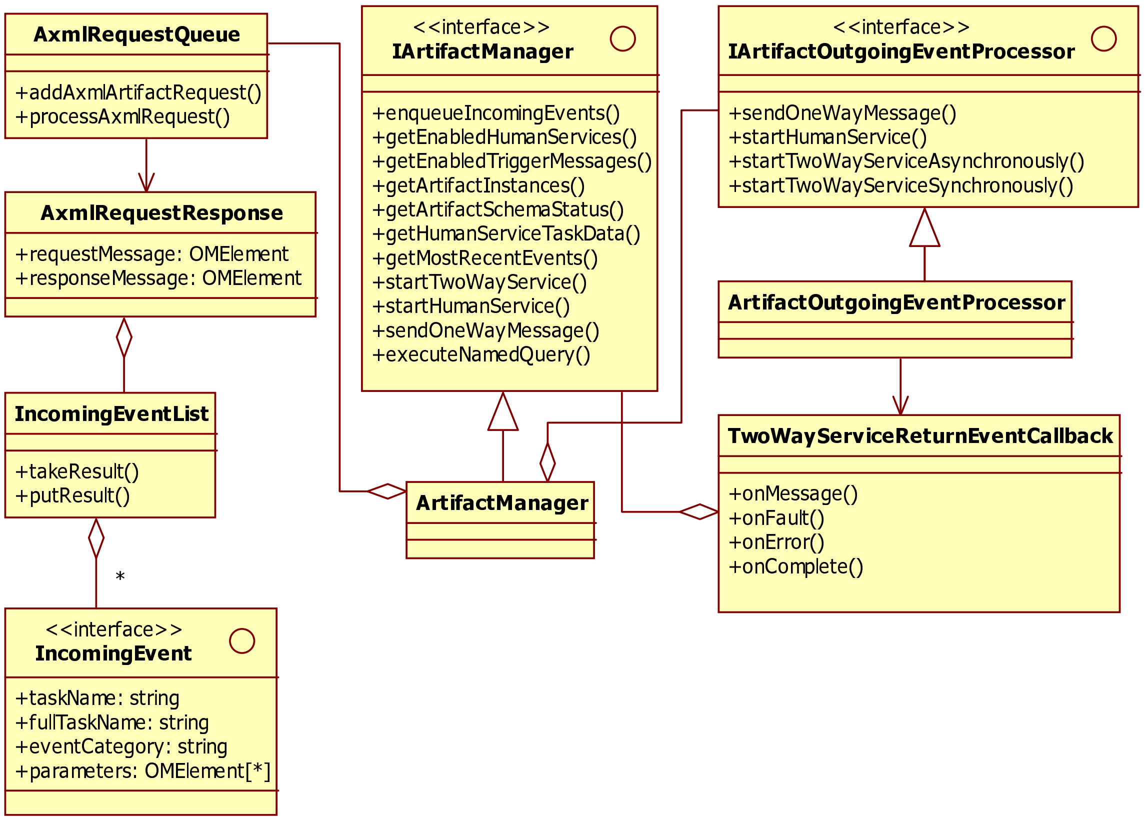


Figure 7.2: A prototype of the Job Information dialog

Figure 7.2 shows ArtifactManager class and most important related classes. The *ArtifactManager* contains three categories of operations.

First category contains lifecycle operation *scheduleExecuteIncomingEvents* for receiving incoming events that originated from external world. This method accepts incoming events, creates a request message that will be later send to Active XML for event processing and adds this message along with incoming events to the request queue.

Second category are lifecycle operations for reacting to outgoing events that originated from ActiveXML during processing of incoming event. That means task opening events. These operations are *startTwoWayService* for processing invocation of given two-way service, *startHumanService* for processing invocation of given human service and *sendOneWayMessage* for processing sending one-way message. *ArtifactManager* delegates processing of these events to *IArtifactOutgoingEventHandler*, because how these events are actually processed is not important for the core system. This copies the semantics of the GSM model, where how task are processed is not the responsibility of the model. All that the core system needs is to get back the response when these tasks ends. Also, *IArtifactOutgoingEventHandler* can be the extension point where custom processing can occur. For example, it is not strictly necessary to call external web services using Axis2, only because Active XML uses Axis2. Different technologies could be used when providing custom implementation of *IArtifactOutgoingEventHandler*. Also, processing human service can actually be implemented as simple notification of associated human performer, for example by sending him a message or e-mail that new task associated to him has started.

Third category are operations that query information about artifact schema status, artifact instance details, task instance details and most recent events. These operations only invoke predefined queries stored in Active XML. They are primarily designed for user interface to provide it with necessary data. Therefore in real implementation the list of operations in this category and their implementation will always be application specific and will depend on the requirements for user interface. These operations are *getArtifactSchemaStatus* to get information about state of entire artifact schema, *getHumanServiceTaskData* to get information about concrete human task instance, *getMostRecentEvents* to get the list of most recent events that occurred in the system and *getArtifactData* to get details about concrete artifact instance. Now there are prepared classes for communication like *ArtifactShortDescription* that contains only active stages and achieved milestones, but there could be a case where more information is necessary and then new classes can be added.

7.2.2 Messages

There are two types of messages in the ASC. Messages for communication between external world and the ASC and messages for communication between the ASC and Active XML. We will refer to them as external messages and internal messages. This distinction is done on purpose, because we want to point out the major difference between these two types of messages that would occur when developing the real application.

Internal messages are core system messages that must always have the same structure because there is a contract between the ASC and Active XML that both sides understand. XQuery code in Active XML that performs event processing expects all messages to have this predefined structure to correctly read the event payload and event meta-data like event name, event category, source artifact id, source artifact class and others. All these data are necessary in our implementation of BA-GSM model and we always need this data no matter what application we develop on top. We do not assume to change the core system and processing queries, so we also do not assume to change the structure of internal messages. We only expect to change the values of event meta-data and event payload. Of course if we decided to change the core system, we could change the structure of these messages as well.

External messages are messages of the system that is implemented on top of the core system and are used for communication between external world and ASC via web services. They are part of external interface of the implemented system. When developing the application, we expect that this messages will be application specific and their specification will depend on application needs. When we change the application, we will probably change these messages as well but we will not change the structure of internal messages, because the core system remains the same.

7.2.3 Processing incoming event

Incoming event is an event that originates in external world and is received by the ASC. This can be either human service return, two-way service return, create call or incoming one-way message. This event can be sent to the system either via web service or directly to ArtifactManager. This depends on the kind of the event. Two-way service return event is always sent directly to the ArtifactManager, because it is created automatically by the system when the service call ends. Human service return events, incoming one-way-messages and create calls are usually sent via web service, because these are usually sent from user interface by human user. But they can be also send directly to ArtifactManager. We will describe how event is sent via web service operation.

The event originates in user interface. User fills event form that contains data for event payload. When he submits the form, the event is sent to the ASC via web service operation. The ASC has a web service *ContainerWorkflowService* that has for each human service return event, each incoming one-way message and each create call one operation to receive this event. This event name is a concatenation of artifact class name and operation name. When the event is received, it is transformed to the message in the format that is used for communication with Active XML. This is one of:

- *HumanServiceReturnEvent* for return from human service.
- *IncomingOneWayMessageEvent* for incoming one way message.
- *CreateChildArtifactEvent* for request to create child artifact instance.
- *CreateRootArtifactEvent* for request to create root artifact instance.
- *TwoWayServiceReturnEvent* for return from two-way service call.
- *OneWayMessageSentEvent* for confirmation of sending one-way message.

This message is then sent to the ArtifactManager with method *scheduleExecuteIncomingEvents*. However it cannot be sent immediately to the Active XML for processing, because Active XML could be processing another event at the moment and only one event can be processed at any time. For this reason ArtifactManager contains linked blocking queue, which contains all incoming unprocessed events. When ArtifactManager receives an event, it first inserts it into this queue where it waits for processing. ArtifactManager then processes events from this queue on another thread. This thread always waits until previous event is fully processed and after that it takes another event from the queue and sends it to the Active XML.

Processing event with Active XML is always done in two steps. We use two steps for optimization reason related to XQuery compilation.

In the first step ArtifactManager only sends event to the Active XML via web service *GenericQueryService* and operation *executeGenericQuery*. This operation gets query that will be executed in the database. We use query that only calls function insert-event-list that is declared in insertModule.xq. This function only inserts the event to the list of received events in events.xml.

In the second step we tell Active XML to start processing the event it just received. This is again done via web service *GenericQueryService* and *execute-GenericQuery*. We use query that only calls function process-event-list-final that is declared in module main.xq. This function starts processing of the event. When processing ends, this query returns a list of outgoing events that were generated during processing of the event.

7.2.4 Processing incoming event by the query

Actual processing of an incoming event is done by XQuery code inside Active XML. Incoming event is sent to Active XML and processed with function process-event-list. This function takes event that was previously inserted to the unprocessed event list in events.xml and starts its processing. The processing is always the same. Apply immediate effect and then evaluate PAC rules. In order to apply immediate effect the query must find affected artifact(s). Which artifacts are affected depends on event type.

1. If it is *CreateRootArtifactEvent*, then it must first create new artifact. Affected artifact is the created artifact.
2. If it is *CreateChildArtifactEvent*, then it must first find parent artifact. Parent artifact ID is part of the event, so the query finds the parent artifact by this ID. Then it creates child artifact inside parent artifact. Affected artifact is the created artifact.
3. If it is *TwoWayServiceReturnEvent*, *OneWayMessageSentEvent* or when it is *HumanServiceReturnEvent*, then there is exactly one affected artifact that already exists. Artifact ID is part of the event, so the query finds the artifact by this ID.
4. If it is *IncomingOneWayMessageEvent*, then there may be many affected artifacts that already exist. Artifact IDs are part of the event, so the query finds the artifacts by this IDs.

If the event is task completion event, which is either *HumanServiceReturnEvent*, *TwoWayServiceReturnEvent* or *OneWayMessageSentEvent*, the query also deactivates the task.

When the query finds affected artifact, it can apply immediate effect. It sets most recent event of the artifact. Then it needs to do the binding. All bindings function are in file binding.xq, which was generated from eXolutio. It needs to determine which binding function to use. Because every event is associated with exactly one artifact class, the concatenation of artifact class name and event name is unique identifier that determines appropriate binding. After binding is done, the PAC rules evaluation is started.

7.2.5 Outgoing event

Outgoing event always originates in ASC. It can be two-way service call event, human service call event and outgoing one-way message event. Outgoing event is created during processing of some incoming event in Active XML. This processing can create entire set of outgoing events. Further processing depends on Active XML representation mode.

Pure query approach

When the event processing ends, this set of outgoing events is returned to ASC as a query result. This is because event processing is actually a call to *executeGenericQuery* operation of *GenericQueryService* that executes query that processes incoming event and returns new outgoing events as a result.

ArtifactManager takes this set of outgoing events and for each one of them it calls handler method of *IArtifactOugoingEventHandler*. This handler is supposed to be overridden if necessary depending on application needs. We describe here default implementation.

Human service call For human service call event it only logs the event, because the human service was already set as started in a query during event processing in Active XML. It is now up to the user to perform the human task and submit response to the system. In real application the handler method could also notify the user who is assigned to this task about new event. For example it could send him a message through the system or send him an email.

One-way message call For one-way message call event it invokes the service call with settings obtained from *ServicePropertiesMapper*. It must obtain service namespace, method and URL. It immediately returns.

Two-way service call For two-way service call event it also invokes the service call. Service call is invoked in asynchronous way so that it does not block other tasks. It accepts the callback that should be called when the service call returns. This is necessary because the return event represents new incoming event that must be eventually propagated back to the system for processing. When the service call ends, the callback is called. It creates message *TwoWayServiceReturnEvent* that represents two way service return event and sends it directly to ArtifactManager via method *scheduleExecuteIncomingEvents*. Thanks to this method the event will be processed as an incoming event when the system is ready. Note that this is the case when incoming event is sent directly via ArtifactManager and not send via web service. This is because the event originated in the system itself.

Continuous calls and naive approach

In case of naive approach, opened tasks are started directly from Active XML, as described in section 5.2.1.

7.3 UI and ArtifactServlet

As we said before, user interface is designed primarily for testing purposes to demonstrate that the core system works. It is implemented with standard HTTP servlet and JSP technologies.

It provides two basic pages: home page that shows current artifact schema status and edit page that allows user to edit and send events to the core system.

The servlet communicates with the core system via web services that provide all necessary data. We will refer to these web services as utility service and workflow service.

1. *ContainerWorkflowService* is used to submit workflow events to the core system. These are human task response events, incoming one way messages and create calls.

2. *ContainerUtilityService* is used to obtain information about current artifact schema status, information about tasks and event history.

Servlet uses both POST and GET methods. POST method is used for save actions to submit events to the core system and GET method is used for view and edit actions.

7.3.1 Actions

There are three kind of actions related to the artifacts that we can do in user interface. We can view artifact data, edit incoming events and send these events to the core system.

The servlet uses URL patterns to decide which action is appropriate. The pattern is:

workspace/ARTIFACT/ACTION/EVENT_CATEGORY/EVENT_NAME

View action View action means that we want to view detailed information about concrete artifact. When servlet recognizes the view action, it contacts the core system via utility web service operation *getArtifactData* that returns the artifact data. Artifact instance is specified by its class and ID. Example:

workspace/CustomerOrder/view?artifactID=1

Edit action Edit action means that we want to create and edit event that we will later submit to the core system. To edit an event, the user interface displays HTML form that represents the event message. User can fill the form data and then send it to the core system.

When servlet recognizes the edit action, it can either directly render the form or first contact the core system. This depends on the event category. Forms that represent create call events and one-way message events can be rendered immediately, because they do not require any additional data. One the other way, human task response events must first load data from the core system, because the human task can contain input parameters. And even if task did not contain any input parameters, the user interface would not know that. So in case of human response event, the servlet first contacts the core system with web service operation *getHumanServiceTaskData* and then renders the form using obtained data. Event is specified by its name, category, artifact class and ID. For example:

workspace/CustomerOrder/edit/human/SetProductCode?artifactID=1

Save action Save action means that we want to submit event to the core system. When servlet recognizes save action, it submits appropriate event to the core system via web service. The save action must always follow from edit action, where user filled the form for the event that he wants to send to the core system. This means that the data from the form must be first transformed into XML representation that is sent in SOAP message. The name of the operation is uniquely identified by the event. For example this uniquely identifies the operation *CustomerOrderSetProductCode*

workspace/CustomerOrder/save/human/SetProductCode

No action No action means that we want to render home page that shows current artifact schema status. When servlet recognizes no action, it loads current schema status from the core system with web service operation *getArtifactSchemaStatus*. It also loads history of most recent events in the system with web service operation *getMostRecentEvents*.

/Container/workspace

This is default home page that shows current artifact schema status.

8. Testing, experiments and evaluation

8.1 Implementation

We implemented an extension into eXolutio based on analysis and design provided in chapter 6. This extension supports working with GSM models, concretely:

- **Modeling**. Definition of GSM model using conceptual modeling, definition of artifact lifecycles and definition and import of event bindings and service properties including their validation against model definition.
- **Generation**. Generation of defined GSM model into Active XML representation, including derivation of invariant preserving rules and rules ordering to ensure evaluation correctness. Generated representation is integrated into source code of an executable system and as such can be subsequently edited or compiled and deployed.
- **Execution**. Deployment of an executable system with generated Active XML representation into Tomcat application server directly from eXolutio. Functionality can be then demonstrated using web browser and manual progress through model workflow, or using automated tests for example models.

8.2 Testing

We proposed and implemented three Active XML representations described in chapter 5: a naive approach, continuous calls approach and pure query approach. When application is starting, it reads from property file which representation to work with. It is important for expected representation to match actual representation generated from eXolutio, because for naive approach, eXolutio must generate embedded service calls into business artifacts document.

We implemented helper testing project, called *ArtifactCenterTests*. It is a simple web service client that reads scenario from the XML file and invokes workflow web services against *ArtifactServiceCenter* according to that scenario. Scenario is a sequence of incoming event messages along with web service operation that should be called. Calling sequentially scenario items corresponds to progress in model workflow and provides automatic testing. Figure 8.1 shows one call item from the customer order scenario. Same functionality could be achieved for example with SoapUI. We also created test cases in Selenium UI to test the system from user interface.

8.3 Evaluation

We can now reason about response time for one incoming event processing in comparison between three Active XML representations that we analyzed in this

```
<call method="CustomerOrderSendToManufacturer" wait="2000">
   <TaskResponse>
      <artifactID>1</artifactID>
      <CustomerOrder>
         <manufacturerID>manufacturer1</manufacturerID>
      </CustomerOrder>
   </TaskResponse>
</call>
```

Figure 8.1: Customer order scenario test call item. This item states that test will first wait 2000 ms and then call a method *CustomerOrderSendToManufacturer* on the *ContainerWorkflowService* with message starting from element *TaskResponse*.

thesis. Intuitively, this is related with amount of service calls that event processing must perform in order to process an incoming event, including starting opened tasks and handle their termination.

- With pure query approach, when an incoming event is set for processing, it is processed by Active XML with one call to *executeGenericQuery* operation. This operation performs B-step processing and returns set of outgoing events. For every event, the ASC invokes one web service call. Therefore for n outgoing events, there is n+1 calls in total.

- With continuous call approach, when an incoming event is set for processing, it is processed by Active XML with one call to *evaluate* operation. This operation invokes *executeGenericQuery* to apply immediate effect and to evaluate rules. Then it invokes *continuousCall* for every opened task, which transitively calls processing operation in the ASC to invoke service asynchronously. Therefore for n outgoing events, there is 2n+1 calls in total.

- With continuous call approach, when an incoming event is set for processing, it is processed by Active XML with one call to *evaluate* operation. This operation invokes *executeGenericQuery* to apply immediate effect and to evaluate rules. Then it invokes one embedded service call for every opened task, which transitively calls processing operation in the ASC to invoke service asynchronously. Therefore for n outgoing events, there is 2n+1 calls in total.

We can see that continuous call approach and naive approach need more service calls, but these additional calls are not very costly. This is because Active XML performs embedded service calls as local operation calls for those services that are located in the same peer. So in our case, additional embedded service calls only invoke local web services using local operation call.

Additional factor is that as more embedded service calls are used in the XML document, the more time Active XML framework needs to load them from the document, create an activation dependency graph and evaluate them in proper order. On the other hand, when invoking corresponding web services from the ASC directly as in pure query approach, there is no need to load and order embedded service calls. However, such loading and ordering does not need to be costly for continuous call approach, because in our case there are only few embedded service calls and their dependencies are simple. In naive approach

Action	Pure query	Continuous call
CustomerOrder.CreateCall	2404	4761
CustomerOrder.SetProductCode	640	6121
CustomerOrder.SendToManufacturer	422	6168
CustomerOrder.CreateWO	514	4745
WorkOrder.EventCreateLI	407	6167
WorkOrder.CreateLI	327	7744
WorkOrder.EventCreateLI	422	7541
WorkOrder.CreateLI	343	8073
WorkOrder.EventCreateLI	437	7825
WorkOrder.CreateLI	327	8147
WorkOrder.EventCreateMPO	421	7962
WorkOrder.CreateMPO	313	8072
MaterialOrder.EventAddLI	484	7915
MaterialOrder.AddLI	328	8166
MaterialOrder.EventAddLI	312	7744
MaterialOrder.AddLI	329	6278
MaterialOrder.EventAddLI	313	6308
MaterialOrder.AddLI	328	8056
MaterialOrder.SendMPOs	328	7793
LineItem.SetToSent	469	7822
LineItem.SetToSent	374	12725
LineItem.SetToSent	390	7810
LineItem.SetToShipped	468	8383
LineItem.SetToShipped	329	10335
LineItem.SetToShipped	313	7899
WorkOrder.assemble	390	8080
CustomerOrder.EventShip	344	8009
CustomerOrder.Ship	406	8321

Figure 8.2: Average response times for CustomerOrder model (in milliseconds).

however, there will be great amount of embedded service calls and more will be added over time, so loading and ordering will be slower.

We measured response times for processing of one incoming event for individual workflow operations using Active XML representation. We measured on two models.

For first measurement we used *CustomerOrder* model, which is defined in chapter 6. We performed eight runs, always starting with empty document for business artifacts. We measured for pure query approach and for continuous call approach. Table 8.2 and shows average measured values.

For second measurement we used *Questionnaire* model, which is used in attached tutorial. We performed eight runs, always starting with empty document for business artifacts. We measured for pure query approach and for continuous call approach. Tables 8.3 show averages measured values.

In the pure query approach, we can see that first workflow operation is always slower (first row in both tables). This is caused by initial compilation of XQuery code that is used for rules evaluation and event bindings. This code is usually quite large, in both models it has about 2500 lines in total. This code is cached after first usage, so following response times are smaller.

Continuous call approach is currently much slower than pure query approach, because processing embedded service calls requires generally great amount of time. This is probably due to prototype implementation of Active XML frame-

Action	**Pure query**	**Continuous call**
Questionnaire.CreateCall	1785	4446
Questionnaire.EventAddQuestionAnswer	357	6550
Questionnaire.CreateQuestionAnswer	620	6567
QuestionAnswer.CreateQuestion	346	6185
Question.EventAddAnswer	287	7459
Question.AddAnswer	321	6991
Question.PublishQuestion	273	7062
Questionnaire.EventAddQuestionAnswer	266	7254
Questionnaire.CreateQuestionAnswer	356	7502
QuestionAnswer.CreateQuestion	315	7360
Question.EventAddAnswer	269	7442
Question.AddAnswer	294	6975
Question.EventAddAnswer	268	7679
Question.AddAnswer	256	7374
Question.PublishQuestion	262	7439
Questionnaire.EventQuestionnaireDefined	267	7620
Questionnaire.EventAddParticipant	251	7635
Questionnaire.AddParticipant	270	6149
Questionnaire.EventParticipantsSelected	381	6434
QuestionAnswer.AnswerMultipleChoiceQuestion	285	12783
QuestionAnswer.CommintAnswer	354	7455
QuestionAnswer.AnswerMultipleChoiceQuestion	291	7084
QuestionAnswer.CommintAnswer	406	7350

Figure 8.3: Average response times for Questionnaire model (in milliseconds).

work, because we also noticed slower responses in Active XML examples from the distribution. To verify this, we tested how long it takes for Active XML to evaluate one embedded call in the document with multiple embedded calls. The testing document contained sequence of embedded calls to operation *getVersion* of service *Version*, which is a sample web service included in Axis 2. This operation returns Axis version. We created seven different Active XML documents with 1, 2, 5, 10, 20, 40 and 80 embedded calls. An example for document with two embedded calls is shown in figure 8.4. Test always stored the document into the database and then evaluated only one of the embedded calls. Table 8.5 shows measured values. We can see that as more embedded calls are in the document, the longer the evaluation of single embedded call takes. However, most of the time was spent in loading the document into the document manager and obtaining the calls, not in actual call invocation.

Number of embedded calls	Evaluation time
1	754
2	890
5	1599
10	3039
20	7267
40	21816
80	91105

Figure 8.5: Average response times for evaluation of one embedded call in Active XML document with multiple embedded calls. Times are in milliseconds.

```
<example>
   <axml:sc axml:id="1">
      <axml:return>
         <axml:append/>
      </axml:return>
      <axml:ws-soap
         endpoint="http://127.0.0.1:6969/Peer/services/Version">
         <getVersion:getVersion/>
      </axml:ws-soap>
   </axml:sc>
   <axml:sc axml:id="2">
      <axml:return>
         <axml:append/>
      </axml:return>
      <axml:ws-soap
         endpoint="http://127.0.0.1:6969/Peer/services/Version">
         <getVersion:getVersion/>
      </axml:ws-soap>
   </axml:sc>
</example>
```

Figure 8.4: Tested AXML document with two embedded calls

We expect that these times would be much smaller in non-prototype implementation. To conclude, we believe that continuous call approach and pure query approach could be both used interchangeably, under important assumption of smaller response times in non-prototype Active XML implementation. Naive approach is unsuitable due to described factors.

9. Conclusion

In this thesis, we presented an approach to connect *Guard-Stage-Milestone model* with *Conceptual model for XML* to enable conceptual modeling of business artifacts using Active XML as a basis. We analyzed and proposed possible representations of business artifacts in an executable model using Active XML and discussed the limitations of Active XML when using it for an implementation of the GSM models. We also presented how the Conceptual model for XML can be useful for definition of the information models of business artifacts and analyzed how to extend eXolutio to support definition of GSM models including an automatic generation of prototypical executable system that would demonstrate the functionality of the model. Finally, we implemented the mentioned extensions to eXolutio and implemented executable system on top of Active XML, that can be used as a basis for generated models from eXolutio. Finally, we performed experiments by modeling concrete GSM models in eXolutio, generating them into executable system and walking through workflow according to the model.

We began with introducing the artifact-centric approach to business process modeling in chapter 1. We pointed out the difference with traditional workflow approaches and described a motivation for this modeling paradigm. In chapter 2, we described the GSM meta-model and business artifacts and presented formal foundations, which we needed in later chapters. This chapter also indicated what problems we need to solve when designing XML realization for business artifacts. Chapter 3 described the Active XML framework and presented related approach, called AXML Artifacts.

Chapter 4 described the Conceptual model for XML, introduced the concept of model driven architecture and discussed its advantages. We started our analysis in chapter 5, where we analyzed and designed possible representations of business artifacts in an executable model. We proposed three alternatives how to realize artifacts evolution using Active XML and discussed their advantages. In chapter 6, we analyzed and designed how to extend eXolutio to support definition of GSM models and analyzed how to generate these models automatically to our executable model from chapter 5. In chapter 7, we described concrete implementation for Artifact Service Center system that executes generated representations using Active XML framework, based on analysis and design from chapter 5. In chapter 8, we evaluated individual representation approaches.

Implementation of eXolutio with integrated extensions is available in attached CD along with three model examples. These models can be translated to executable Active XML representation into attached Active XML distribution and used for demonstration or further reused and edited.

9.1 Main contribution

The main contribution of our work is introducing business artifacts into the world of XML using model driven architecture, by a combination of conceptual XML data modeling and business artifact modeling into one modeling framework. Designer can define information models of business artifacts in the platform independent level, specify final XML representation in the platform specific level and define artifact lifecycles and declarative rules using concepts from Guard-Stage-Milestone meta-model. He can also define PSM schemas for incoming event messages that are used in the model. Putting all this together, designer can finally generate executable Active XML representation of the model, immediately deploy it and inspect functionality of the model. Designer can also reuse and extend generated representation to build his own application based on business artifacts.

9.2 Future work

9.2.1 GSM visualization for lifecycle models

For the purpose of our prototype, we used our custom visualization for lifecycle models that is different from the visualization used in the original GSM lifecycles, although it can express the same basic concepts that are necessary for us. Unlike the original visualization, it does not support visualized attachment of sentries to guards and milestones and it does not support flow-macros. However, these are both advanced and rather *syntactic sugar* concepts and lifecycle models can be equally defined without them. Nevertheless, it would be nice to support the original visualization for lifecycle models, at least for basic concepts, in order to maintain the appearance consistency.

9.2.2 Artifact data distribution

Currently, an entire artifact information model is contained in one peer. Since Active XML supports data distribution, it would be interesting to extend eXolutio modeling to define some data attributes to be external, located in different peer. This would make sense for example for data that are already located in different peer and accessible through web services. Of course, obtaining such data when needed can be modeled as concrete task instead, but this approach can be useful for data that needs to be used frequently in multiple different tasks and needs to be always up to date. This is where Active XML would be very helpful, because it would always update this data in the document prior the rules evaluation. This could be better with Active XML lazy query evaluation, so that data would be updated only when really necessary. However, current prototype Active XML representation does not support lazy query evaluation.

9.2.3 Optimization in artifacts representation

Currently, all artifact instances are stored in the same XML document in generated Active XML representation. This means that as this document grows over time, the rule evaluation must process greater amount of business artifact instances, although there can be artifact instances that are not relevant to current rules evaluation. This is strongly connected to dependency between artifact instances across different business process instances. For example, customer order artifact from our example in chapter 5 has child artifacts and all these artifacts are relevant during rules evaluation, but other customer orders, that correspond to another ordering business process instance, might not. This is however not a general case, because designer can always create a new rule that would depend on another customer order instances. A direct approach would be to inspect defined rules and decide if there are any dependencies and if not, artifacts can be placed in different documents. But this immediately brings a problem with rules evolution, for example when designer adds a new rule that would depend on both divided instances. Since these two artifacts would be already placed in different documents, rules evaluation would need to take this into account and be able to evaluate over multiple documents.

10. CD contents

The attached CD contains the following filesystem structure with the related artifacts of this work:

- `Code`
  - `Exolutio` - this folder contains source code of eXolutio with integrated extensions.
  - `ArtifactServiceCenter` - this folder contains source code for Artifact Service Center. This source code contains also Eclipse project file.
- `Bin`
  - `Exolutio` - this folder contains an installer of eXolutio with integrated extensions.
  - `ActiveXML` - this folder contains used version of Active XML distribution.
  - `Ant` - this folder contains Apache Ant.
- `Samples` - this folder contains samples of business artifact models. Every model is in own folder containing eXolutio project file and event bindings file.
- `thesis.pdf` - PDF version of this thesis.
- `tutorial.pdf` - PDF version of user tutorial.

Bibliography

[1] A. Nigam and N. S. Caswell. Business artifacts: An approach to operational specification. *IBM Systems Journal*, volume 42, issue 3, pages 428–445. IBM Corp., Riverton, NJ, USA, 2003. ISSN 0018-8670.

[2] K. Bhattacharya, R. Hull and J. Su. A data-centric design methodology for business processes. *Handbook of Research on Business Process Modeling*, 2009. ISBN 1605662887.

[3] W. M. P. van der Aalst and M. Weske. Case handling: a new paradigm for business process support. *Data and Knowledge Engineering*, volume 53, issue 2, pages 129–162. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, 2005. ISSN 0169-023X.

[4] R. Hull et al. A formal introduction to business artifacts with guard-stage-milestone lifecycles, Version 0.8, May, 2011. Draft IBM Research internal report, available online at `http://researcher.watson.ibm.com/researcher/view_page.php?id=1710`.

[5] K. Bhattacharya, N. S. Caswell, S. Kumaran, A. Nigam and F. Y. Wu. Artifact-centered operational modeling: Lessons from customer engagements. *IBM Systems Journal*, volume 46, issue 4, pages 703–721. IBM Corp., Riverton, NJ, USA, 2007. ISSN 0018-8670.

[6] *Artifact-centric service interoperation.* `http://www.acsi-project.eu`.

[7] R. Hull et al. Introducing the guard-stage-milestone approach for specifying business entity lifecycles. In *Proceedings of the 7th international conference on Web services and formal methods* (WS-FM'10). Lecture Notes in Computer Science. Springer-Verlag, Berlin, Heidelberg, 2010. ISBN 978-3-642-19588-4.

[8] R. Hull et al. Business artifacts with guard-stage-milestone lifecycles: Managing artifact interactions with conditions and events. In *Proceedings of the 5th ACM international conference on Distributed event-based system* (DEBS '11), pages 51-62. ACM, New York, NY, USA, 2011. ISBN 978-1-4503-0423-8.

[9] *IBM Research: Richard Hull Profile Page.* `http://researcher.watson.ibm.com/researcher/view_person_subpage.php?id=1710`

[10] E. Damaggio, R. Hull and R Vaculin. On the equivalence of incremental and fixpoint semantics for business artifacts with guard-stage-milestone lifecycles. *Journal Information Systems*, volume 38, issue 4, pages 561-584. Elsevier Science Ltd., Oxford, UK, UK, 2013. ISSN 0306-4379.

[11] R. Hull. Artifact-centric business process models: Brief survey of research results and challenges. In *Proceedings of the OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008. Part II on On the Move to Meaningful Internet Systems* (OTM '08), pages 1152-1163. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 978-3-540-88872-7.

[12] T. Chao et al. Artifact-based transformation of IBM Global Financing: A case study. In *Proceedings of the 7th International Conference on Business Process Management (BPM)*, pages 261-277. Springer-Verlag, Berlin, Heidelberg, 2009. ISBN 978-3-642-03847-1.

[13] Y. Sun, R. Hull, R. Vaculin. Parallel Processing for Business Artifacts with Declarative Lifecycles. *On the Move to Meaningful Internet Systems: OTM 2012*, pages 433-443. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 2012. ISBN 978-3-642-33606-5.

[14] S. Abiteboul, O. Benjelloun, I. Manolescu, T. Milo and R. Weber. Active XML: Peer-to-Peer Data and Web Services Integration. In *Proceedings of the 28th international conference on Very Large Data Bases* (VLDB '02), 2002.

[15] S. Abiteboul, O. Benjelloun, I. Manolescu, T. Milo and R. Weber. Active XML: A Data-Centric Perspective on Web services, 2002.

[16] D. Booth and C. K. Liu. *Web Services Description Language (WSDL) Version 2.0 Part 0: Primer.* W3C, June 2007. `http://www.w3.org/TR/wsdl20-primer`.

[17] N. Mitra and Y. Lafon. *SOAP Version 1.2 Part 0: Primer (Second Edition).* W3C, April 2007. `http://www.w3.org/TR/soap12-part0`.

[18] *Extensible Markup Language (XML).* T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler and F. Yergeau. *Extensible Markup Language (XML) 1.0 (Fifth Edition).* W3C, September 2006. `http://www.w3.org/TR/REC-xml`.

[19] S. Abiteboul et al. Lazy query evaluation for Active XML. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 227-238. ACM New York, NY, USA, 2004. ISBN 1-58113-859-8.

[20] S. Abiteboul, O. Benjelloun and T. Milo. The Active XML project: an overview. *The VLDB Journal* volume 17, issue 5, pages 1019-1040. Springer-Verlag, New York, Secaucus, NJ, USA, 2008. ISSN 1066-8888.

[21] The Active XML team. *Active XML Primer.* `ftp://ftp.inria.fr/INRIA/Projects/gemo/gemo/GemoReport-307.pdf`.

[22] A. Ghitescu and E. Taroza. *Active XML Documentation Version 2.1.4*, 2008. http://webdam.inria.fr/axml/axmlv2/resources/axmldoc.pdf/.

[23] I. Manolescu. *Re: Lazy Query Evaluation is available?.* 27 Feb 2008. Accessible at `http://mail-archive.ow2.org/activexml/2008-02/msg00016.html`.

[24] S. Cook. *Domain-Specific Modeling.* `http://msdn.microsoft.com/en-us/library/bb245773.aspx`.

[25] *Object Management Group.* `http://www.omg.org`.

[26] M. Necasky and I. Mlynkova. On Different Perspectives of XML Data Evolution. *Proceedings of the 2009 20th International Workshop on Database and Expert Systems Application* (DEXA '09), pages 422-426, IEEE Computer Society, Washington, 2009. ISBN: 978-0-7695-3763-4, ISSN: 1529-4188.

[27] M. Necasky. *Conceptual Modeling for XML, Ph.D. thesis.* 2008.

[28] M. Necasky, I. Mlynkova, J. Klimek and J. Maly. When conceptual model meets grammar: A dual approach to XML data modeling. *Data and Knowledge Engineering*, volume 72, pages 1-30, 2012. ISSN: 0169-023X.

[29] Object Management Group. *Documents Associated With Business Process Model And Notation (BPMN) Version 2.0*, 2011. `http://www.omg.org/spec/BPMN/2.0`.

[30] ACSI - Artifact-Centric Service Interoperation. *The core ACSI artifact paradigm: artifact-layer and realization-layer*, Deliverable 1.1, 2011. `http://www.acsi-project.eu/deliverables/D1.1_The_core_ACSI_artifact_paradigm.pdf`.

[31] W. M. Meier, L. J Olsson. *Tuning the Database*, 2011. `http://cdi.uvm.edu/exist/tuning.xml`.

[32] S. Abiteboul, P. Bourhis, B. Marinoiu, and A. Galland. AXART - Enabling Collaborative Work With AXML Artifacts. *Proceedings of the VLDB Endowment*, volume 3, issue 1-2, pages 1553-1556, 2010. ISSN 2150-8097.

[33] P. Bourhis. *Dell Supply Chain in Active XML 2.1.4.* `http://www.labri.fr/perso/anca/docflow/meeting19-05-08_files/bourhis-dell.pdf`.

[34] JBoss Community. *Drools Expert.* `http://www.jboss.org/drools/drools-expert.html`.