Technische Universität Berlin

Forschungsberichte
der Fakultät IV – Elektrotechnik und Informatik

Metamodels and Transformations for Software and Data Integration

Henning Agt, Gregor Bauhoff, Daniel Kumpe, Ralf Kutsche,
Nikola Milanovic, Michael Shtelma, Jürgen Widiker

Technische Universität Berlin
Fakultät IV (Elektrotechnik und Informatik)
Fachgebiet Datenbanksysteme und Informationsmanagement
Einsteinufer. 17
D-10587 Berlin
{hagt,gbauhoff,dkumpe, rkutsche,nmilanov,mshtelma,jwidiker}
@cs.tu-berlin.de
Homepage: http://www.dima.tu-berlin.de

Bericht-Nr. 2010-02
ISSN 1436-9915

Metamodels and Transformations for Software and Data Integration

Henning Agt, Gregor Bauhoff, Daniel Kumpe, Ralf Kutsche,
Nikola Milanovic, Michael Shtelma, Jürgen Widiker
Technische Universität Berlin
Fakultät IV (Elektrotechnik und Informatik)
Fachgebiet Datenbanksysteme und Informationsmanagement
Einsteinufer. 17
D-10587 Berlin
{hagt,gbauhoff,dkumpe, rkutsche,nmilanov,mshtelma,jwidiker}
@cs.tu-berlin.de
Homepage: http://www.dima.tu-berlin.de

March 19, 2010

Abstract

Metamodels define a foundation for describing software system interfaces which can be used during software or data integration processes. The report is part of the BIZYCLE project, which examines applicability of model-based methods, technologies and tools to the large-scale industrial software and data integration scenarios.

The developed metamodels are thus part of the overall BIZYCLE process, comprising of semantic, structural, communication, behavior and property analysis, aiming at facilitating and improving standard integration practice. Therefore, the project framework will be briefly introduced first, followed by the detailed metamodel and transformation description as well as motivation/illustration scenarios.

Contents

1 Introduction into BIZYCLE **5**

2 Computation Independent Metamodel **7**
2.1 Abstract Syntax 7
2.2 Views 9
2.3 Constraints 10
2.4 CIM Example 12

3 Platform Specific Metamodels **15**
3.1 SAP R/3 18
3.1.1 SAP R/3 Core Package 19
3.1.2 SAP R/3 Communication Package 20
3.1.3 SAP R/3 Structure Package 21
3.1.4 SAP R/3 Constraints 22
3.1.5 SAP R/3 Model Example 24
3.2 Relational Database Systems 24
3.2.1 Relational Database Core Package 24
3.2.2 Relational Database Communication Package 26
3.2.3 Relational Database Structure Package 27
3.2.4 Relational Database Constraints 27
3.2.5 Relational Database Model Example 28
3.3 Java 2 Platform, Enterprise Edition (J2EE) 30
3.3.1 J2EE Components Core Package 31
3.3.2 J2EE Components Communication Package 31
3.3.3 J2EE Components Structure Package 32
3.3.4 J2EE Components Constraints 33
3.3.5 J2EE Components Model Example 33
3.4 Web Services 35
3.4.1 Web Services Core Package 35
3.4.2 Web Services Communication Package 35
3.4.3 Web Services Structure Package 37
3.4.4 Web Services Constraints 37
3.4.5 Web Services Model Example 37
3.5 Extensible Markup Language (XML) 39
3.5.1 XML Core Package 39
3.5.2 XML Communication Package 39
3.5.3 XML Structure Package 40
3.5.4 XML Model Example 40
3.6 XML Schema Definition (XSD) 40

4 Platform Independent Metamodel **42**
4.1 PIMM Core Package 42
4.2 PIMM Structure Package 42
4.3 PIMM Communication Package 44

5 Semantic Metamodel **46**
5.1 Abstract Syntax 46
5.2 Semantic Metamodel Constraints 46
5.3 Ontology Example 48

6 Annotation Metamodel **49**
6.1 Core Annotation Metamodel 49
6.2 Data Annotation 50
6.3 Functional Annotation 51
6.4 Value Annotation 53
6.5 Annotation Metamodel Constraints 53
6.6 Annotation Examples 56

7 Property Metamodel **58**
7.1 NFP Taxonomy 58
7.2 Non-functional Property Metamodel 60

8 Connector Metamodel **64**
8.1 Connector Package 64
8.2 The Expression Metamodel 65
8.3 The Structure packages: *structure* and *message* 66
8.4 Concrete Syntax 67
8.5 Example Scenario 68

9 Model Transformation **71**
9.1 Core Package Transformations 72
9.1.1 SAP R/3 72
9.1.2 Relational Database Systems 73
9.1.3 Java 2 Platform, Enterprise Edition(J2EE) 74
9.1.4 Web Services 74
9.1.5 Extensible Markup Language (XML) 75
9.2 Structure Package Transformations 75
9.2.1 SAP R/3 75
9.2.2 Relational Database Systems 75
9.2.3 Java 2 Platform, Enterprise Edition(J2EE) 75
9.2.4 Web Services 75
9.2.5 Extensible Markup Language (XML) 76
9.2.6 XSD Lite 76
9.3 Communication Package Transformation 76
9.3.1 SAP R/3 76
9.3.2 Relational Database Systems 77
9.3.3 Java 2 Platform, Enterprise Edition(J2EE) 77
9.3.4 Web Services 78
9.3.5 Extensible Markup Language (XML) 78
9.4 Semantic Package Transformations 78
9.4.1 SAP R/3 78

9.4.2 Relational Database Systems 79
9.4.3 Java 2 Platform, Enterprise Edition(J2EE) 79
9.4.4 Web Services 79
9.4.5 XSD Lite 79
9.5 Transformation Examples 79
9.5.1 SAP R/3 79
9.5.2 Relational Database Systems 79
9.5.3 Java 2 Platform, Enterprise Edition(J2EE) 79
9.5.4 Web Services 79
9.5.5 Extensible Markup Language (XML) 80

10 Summary **83**

1 Introduction into BIZYCLE

This report documents part of the BIZYCLE interoperability platform, a joint industry/academy R&D effort to investigate in large-scale the potential of model-based software and data integration methodologies, tool support and practical applicability for different industrial domains. Already published specific details about this platform can be found in [14, 15, 17, 4, 5, 16, 20]. The consortium consists of six industrial partners and academia and is part of the program of the German government, the Berlin-Brandenburg Regional Business Initiative BIZYCLE (www.bizycle.de). The long-term goal is to create a model-based tool generation and interoperability platform, in order to allow for improved and partially automated processes in software component and data integration. These integration tasks are performed by experienced software engineers and application experts, manually programming the *connectors* i.e. *the glue* among software components or (sub)systems. This requires domain-specific as well as integration requirements' analysis skills. It is a very expensive procedure, estimated between 50 and 80 per cent of the overall IT investments (see e.g., [18]). In order to reduce this cost factor, the BIZYCLE initiative develops a methodology, tools and metatools for semi-automated integration according to the MDA paradigm.

With Model Driven Architecture (MDA) and Model Driven Development (MDD), the Object Management Group (OMG) has defined a standard process of modeling software systems by a top-down approach on three different levels of abstraction: computation independent (CIM), platform independent (PIM) and platform specific (PSM). BIZYCLE takes advantage of this approach and supports all three model levels (CIM, PIM, PSM) for the purpose of developing a process for integration of software artifacts, components, systems or data (hereinafter referred to as artifacts).

Integration with BIZYCLE starts at the CIM level specifying the integration process and requirements in terms of an integration scenario. Unlike the MDA approach (CIM to PIM to PSM), the description of software artifacts themselves starts at the PSM level. The artifacts are treated as black boxes and only their interfaces are taken into account. The artifacts are not initially described at the PIM level because they already exist and cannot be modified. Platform specific descriptions of different artifacts are transformed to the PIM level for comparison and composition conflict analysis. BIZYCLE offers a set of metamodels at each of the MDA levels. Table 1 gives an overview of the BIZYCLE metamodels.

The *Computation Independent Metamodel (CIMM)* offers elements for the description of an integration scenario with its artifacts, processes and abstract data exchanges.

Various *Platform Specific Metamodels (PSMM)* are used to describe homogeneous platform specific interface *families*, e.g., SQL database interfaces, Web Services, SAP R/3 BAPI and IDOC interfaces, XML files, J2EE and .NET components.

The *Platform Independent Metamodel (PIMM)* represents the common abstrac-

Level	Purpose	Metamodels	Multilevel Metamodels	
Computation independent	Business scenario	CIMM	PMM	SMM, AMM
Platform independent	Conflict analysis	PIMM		
Platform specific	Technical interfaces	PSMMs		

Table 1: BIZYCLE metamodel overview

tion basis for interface description of the integration artifacts. Model instances of that metamodel are created using model-to-model transformation of PSMs, currently realized with ATLAS transformation language (ATL)[2].
General properties and attributes of integration artifacts are organized in multilevel metamodels in order to be able to use them at all three MDA levels. With the *Semantic Metamodel (SMM)* it is possible to build an ontology of the integration domain. The knowledge contained in the ontology is used to declare the meaning of the artifact's details at all three MDA levels. For non-functional properties of the artifacts, such as call order and quality of service, the *Property Metamodel* (PMM) is defined.
The multilevel metamodels are linked to the metamodels of the CIM, PIM and PSM level. Content of the semantic model can be shared between different levels. The *Annotation Metamodel* (AMM) realizes the linkage between all levels of abstraction.
Finally, the *Connector Metamodel* (CMM) describes integration solution, that is, the component that mediates between system interfaces that are to be integrated. The CMM is logically placed at the PIM level, and is used to generate connector code.

2 Computation Independent Metamodel

The BIZYCLE integration process captures early stages of an integration scenario at the Computation Independent Model (CIM) level. It describes scenario requirements with an abstract business process and data flow model, regardless of the technical details of the underlying systems. In the following sections we describe the Computation Independent Metamodel (CIMM) in terms of UML class diagrams as well as views that are defined to show relevant aspects of the models. Constraints on the metaclasses and associations are given and examples are presented.

2.1 Abstract Syntax

The Computation Independent Metamodel describes software integration scenarios from an abstract business perspective. The main purpose of modeling a scenario at CIM level is to create a comprehension model for people involved in the integration project and to define integration requirements that are used in the analysis of interface conflicts and code generation. The CIMM provides means to name software and data-related artifacts that are involved in the integration, to model the process of the integration in terms of activities and transitions that shall be performed by the artifacts and structural and data exchange aspects, regardless of any technical system's details.

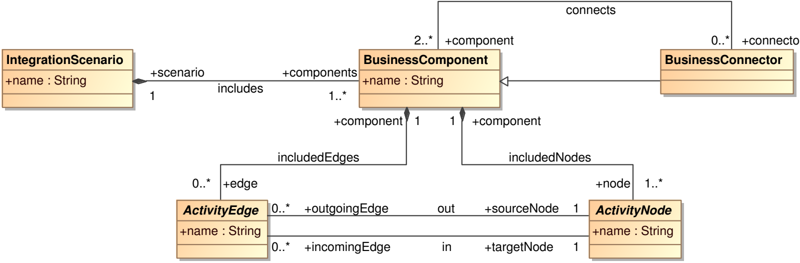


Figure 1: CIMM – Basic metaclasses for artifact and process modeling

Figure 1 illustrates the main CIMM metaclasses that are used for artifact and process modeling. `IntegrationScenario` names an integration project or part of it and is used as root model element container because of the implementation with Eclipse Modeling Framework's Ecore that demands containment hierarchies in tree-based editors. A `BusinessComponent` represents software or data artifacts, that are involved in an integration project (e. g., a relational database). It does not necessarily have to be a concrete technical interface. For example, it can represent a customer relationship management system consisting of a MySQL database and a Web Service that can be accessed separately. A special business component is the `BusinessConnector`, that stands for the interconnection between different systems. It handles interoperability of the artifacts, that cannot be performed by themselves. A business connector connects

two or more business components. The abstract metaclasses `ActivityNode` and `ActivityEdge` are used to express data and control flow of the integration scenario. Nodes are connected through incoming and outgoing edges. Fork and join is represented with multiple outgoing edges and multiple incoming edges respectively. A single edge must be connected with exactly one source and one target node.
The CIMM defines several types of activities that can be performed by a business component and business connector (Figure 2). `ControlNodes` determine the beginning (`InitialNode`) and the end (`FinalNode`) of an integration scenario. `ExportInterface` and `ImportInterface` are used to model component interaction by data transfer or functional coupling. Using `InternalComponentAction` it is possible to express activities that are not relevant to data flow but are helpful for understanding the overall scenario context.
The metamodel also defines the metaclass `BusinessFunction` that represents functionality of the integrated artifacts. In the scope of an integration scenario it processes incoming data or performs a task required by other artifacts. Business functions can only be used inside of business components. They can be semantically annotated (see Section 6) and further described with text (attribute `description`).
`ConnectorFunctions` are part of business connectors. A business architect uses them to predefine connector functionality, in case he has prior knowledge of, for example, data transformation requirements. The seven connector function types shown in Figure 2 are abstracted from the set of Enterprise Application Integration (EAI) patterns [11]. They represent control (e. g., `Timer)` or conflict resolution functions (e. g. `Transformer, Router or Aggregator`). Semantic annotation is inherited from business functions.

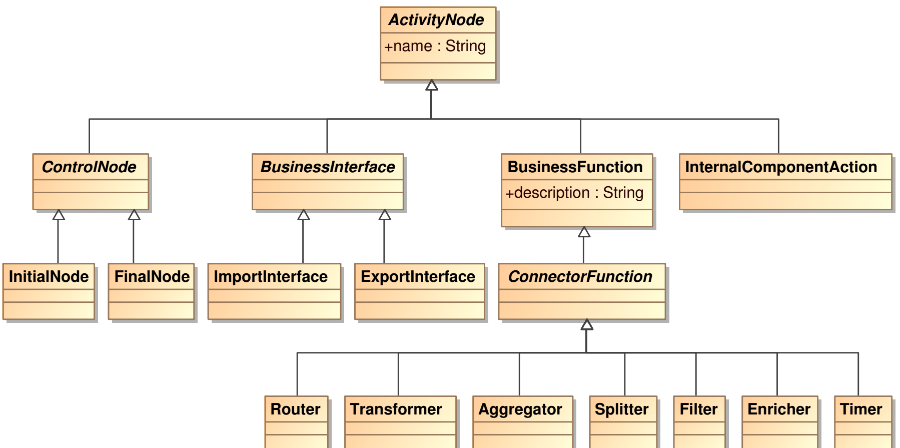


Figure 2: CIMM - Activities of business components

Two different kinds of edges can connect activities of the integration process

(see Figure 3). The `ControlFlow` is used to express transition from one activity to another inside of business components. The `Connection` models data and control flow between different business components or connectors. It connects export interfaces, import interfaces and connector functions and is responsible for transport of `BusinessObjects`.

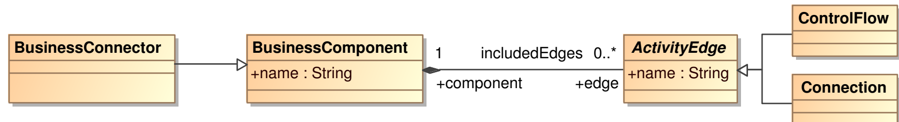


Figure 3: CIMM - Activity edges

Finally the CIMM defines the `BusinessObject` class to describe data exchanged by the artifacts independent of any data type of the underlying interfaces. Figure 4 shows all relations of the business object to other metaclasses of the CIMM. A connection transports at least one business object and a business object can be transported at several locations of the integration scenario. They are produced or consumed by business functions and can be hierarchically structured. Business objects can also be semantically annotated (see Section 6).

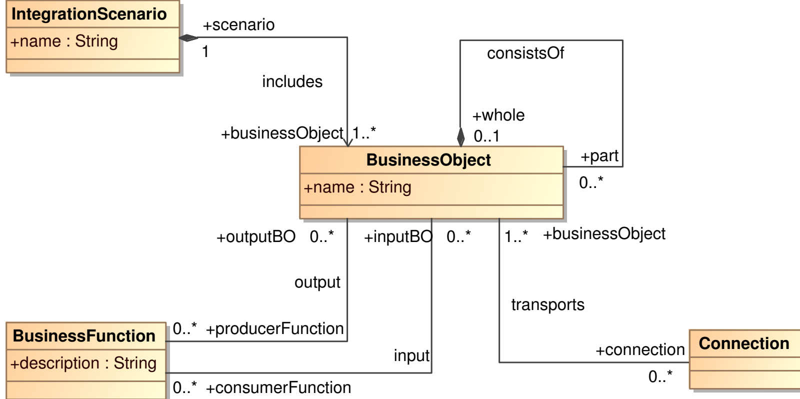


Figure 4: CIMM - Business object

2.2 Views

The CIMM allows to model various aspects of an integration scenario, e. g., the flow and data-oriented aspects. With regard to the usability of a modeling module of the BIZYCLE model-based integration framework (MBIF), an integration scenario should not be presented on the whole to the user. We identified five potential graphical views, which are described in the following. The con-

crete graphical syntax of the CIM views is realized with the Graphical Modeling Framework (GMF) of Eclipse. Table 2 gives an overview of the proposed views.

View	Purpose
Flow View	Flow of the integration scenario
Object View	Hierarchical structure of the business objects
Connection View	Connectivity and data transport
Function View	Business function dependencies
Semantic View	Semantic annotation of the scenario

Table 2: Computation Independent Model Views

Integration scenario modeling starts with the *Flow View* that includes the complete secnario process. The visualization is similar to the UML activity diagram. It visualizes instantiated `BusinessComponents` as swimlanes, `ActivityNodes` with different shapes being part of the components and `ControlFlow` as arrows between nodes inside the swimlanes as well as `Connections` as arrows between `BusinessInterfaces` of different components. The view also shows `BusinessConnector` and included `ConnectorFunctions`.
The *Object View* is responsible for hierarchical modeling of `BusinessObjects` refinements. Therefore only instances of this metaclass and their consistsOf-relations are shown.
The *Connection View* hides all internal activities of components and shows only data transport characteristics. That includes `BusinessComponents` with their `BusinessInterfaces` and `Connections` and additionally the root `Business-Objects`, which are exchanged among them. `BusinessObjects` are associated to `Connections` that transport them.
The *Function View* shows `BusinessFunctions` and sub-classes including `ConnectorFunctions` as well as the `BusinessObjects`, that are input and output of functions. The view shows relevant business data flow describing how `Business-Objects` are being processed and transformed into each other.
The *Semantic View* is used to semantically describe integration scenarios. Additional semantic knowledge is added in terms of semantic annotations to annotatable metaclasses (`BusinessObject` and `BusinessFunction`). Both business objects and functions can be semantically enriched at this level. The view is implemented as Eclipse Property View in which business objects and functions can be linked to domain objects and functions of the semantic metamodel (see section 5). Examples of the CIM views are presented in section 2.4.

2.3 Constraints

To ensure valid usage of the Computation Independent Metamodel several OCL constraints have been added to metamodel.

- Integration scenario cannot have more than one initial node.

```
context IntegrationScenario inv:
  self.components.node->select(n |
  n.oclIsTypeOf(InitialNode))->size() = 1
```

- Integration scenario cannot have more than one final node.

```
context IntegrationScenario inv:
  self.components.node->select(n |
  n.oclIsTypeOf(FinalNode))->size() = 1
```

- Business components, business connectors and business objects must have a unique name within the integration scenario.

```
context IntegrationScenario inv:
  self.components->isUnique(name)
context IntegrationScenario inv:
  self.businessObject->isUnique(name)
```

- Business connector may only contain connector functions and internal component actions.

```
context BusinessConnector inv:
  self.node->forAll(n | n.oclIsKindOf(ConnectorFunction) or
  n.oclIsKindOf(InternalComponentAction))
```

- Business component cannot contain connector functions.

```
context BusinessComponent inv:
  self.oclIsTypeOf(BusinessComponent) implies self.node->
  select(n | n.oclIsKindOf(ConnectorFunction))->isEmpty()
```

- Activity nodes must have unique names within the business component.

```
context BusinessComponent inv: self.node->isUnique(name)
```

- Initial node cannot have any incoming edges.

```
context InitialNode inv: self.incomingEdge->isEmpty()
```

- Outgoing edges of an initial node must be of type control flow.

```
context InitialNode inv:
  self.outgoingEdge->forAll(c | c.oclIsTypeOf(ControlFlow))
```

- Final node cannot have any outgoing edges.

```
context FinalNode inv: self.outgoingEdge->isEmpty()
```

- Incoming edges of a final node must be of type control flow.

```
context FinalNode inv:
  self.incomingEdge->forAll(c | c.oclIsTypeOf(ControlFlow))
```

- Outgoing edges of export interfaces must be of type connection.

```
context ExportInterface inv:
  self.outgoingEdge->forAll(c | c.oclIsTypeOf(Connection))
```

- Incoming edges of export interfaces must be of type control flow.

```
context ExportInterface inv:
  self.incomingEdge->forAll(c | c.oclIsTypeOf(ControlFlow))
```

- Incoming edges of import interfaces must be of type connection.

```
context ImportInterface inv:
  self.incomingEdge->forAll(c | c.oclIsTypeOf(Connection))
```

- Outgoing edges of import interfaces must be of type control flow.

```
context ImportInterface inv:
  self.outgoingEdge->forAll(c | c.oclIsTypeOf(ControlFlow))
```

- Activity nodes cannot be connected with themselves.

```
context ActivityNode inv: self.outgoingEdge->forAll(e :
   ActivityEdge | e.targetNode <> self)
```

2.4 CIM Example

In this section an integration scenario is presented that demonstrates the features of the Computation Independent Metamodel and its views. Two systems are involved in the integration: a Web Shop and an ERP system. The Web Shop is responsible for taking orders of customers. If a new order is processed, parts of the order item list shall be transferred to the ERP system for stock calculations. In Figure 5 the Flow View of the scenario is shown. The Web Shop and the ERP system are modeled as business components on the left and right. Functionality of the systems that is relevant to integration is modeled as business function activity inside the components (*Customer Order Processing*

and *Process Stock*). The integration flow includes Web Shop export interface *Send Order Data* and ERP system import interface *Receive Item List*. Therefore, systems must provide read and write access (technical interfaces) for their data. The scenario also includes a business connector *Item Processor* that filters (*Order Item Filter*) the data exported by the Web Shop. The filter accepts data from the Web Shop via Connection *Transport Order Data* and delivers data to the import interface of the ERP system via Connection *Transport Item List*.

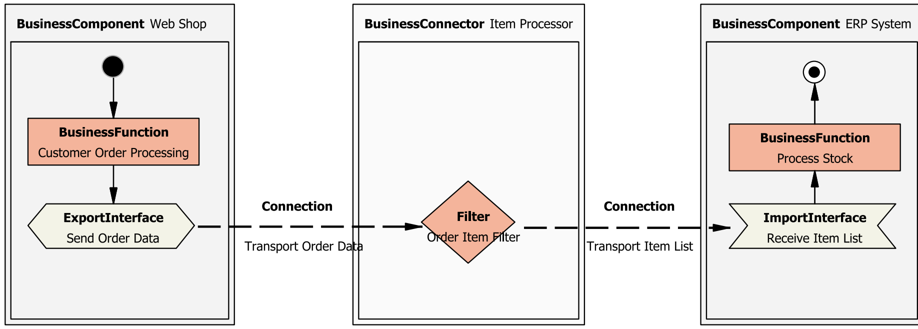


Figure 5: Example integration scenario - Flow View

The CIM Object View shows business/data objects. Figure 6 illustrates two business objects and their structure that are used in the example scenario. *Order Data* represents the data exported by the Web Shop, while *Item List* is the data accepted by the ERP system. The order consists of customer details, order number and details on the items purchased. The *Item List* is composed of identifier and item quantity.

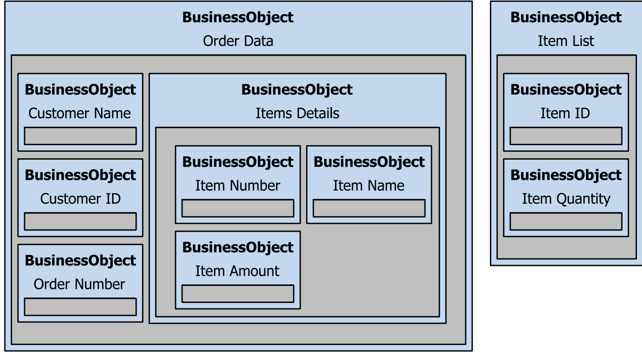


Figure 6: Example integration scenario - Object View

After business objects have been defined, top-level objects are associated to connections in the Connection View (Figure 7). *OrderData* is transported by the connection between the export interface and the filter and *ItemList* is transported by the second connection. If a new business object is added in the object

view it appears in the connection and function views.

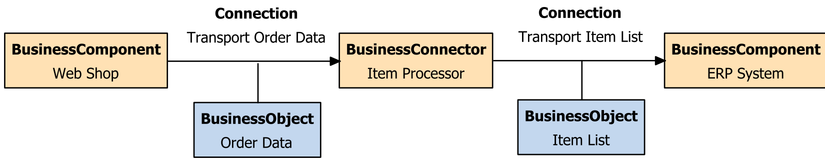


Figure 7: Example integration scenario - Connection View

Business function dependencies are defined in Function View of the CIM (Figure 8a) in form of the object data flow. Input and output links are created between business functions and objects to determine which data is produced or consumed by which functionality. In contrast to the Connection View in which transport paths are modeled, the Function View specifies sources and sinks of business objects. Figure 8b shows the whole computation independent model of the example integration scenario as a tree editor implemented with the Eclipse Modeling Framework. An example of the Semantic View is given in Section 6.

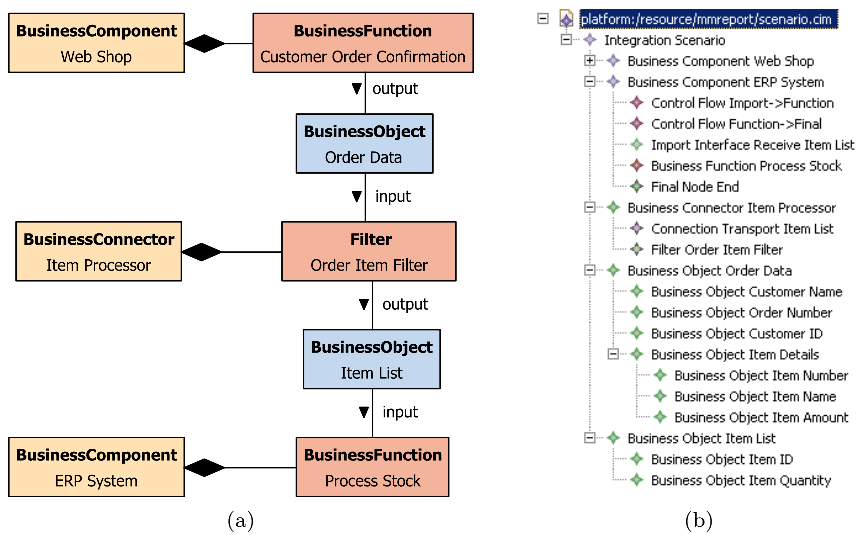


Figure 8: Example integration scenario - Function View and Model tree

3 Platform Specific Metamodels

Platform specific metamodels (PSMM) describe software system interfaces on a technical level of abstraction. Currently we support relational database systems, J2EE applications, SAP R/3 ERP systems, Web Services and XML structured flat files. We further plan to support .NET applications in the near future. System interfaces separate internal methods from communication methods exposed to external systems, thus encapsulating shared business logic. Communication with external systems can be carried out in different ways: SAP R/3 systems provide remote accessible methods (BAPIs) and predefined structured files (IDOCs), J2EE applications have remote accessible methods (EJBs). Many systems provide direct access to their backend datastore (e.g., relational database) database, which is used as internal storage. If systems have neither remote accessible functions, nor accessible datastore, they often perform data exchange via flat files, for example XML- or CSV-structured files. Another way for data exchange within the Microsoft Office Applications is the Object Linking and Embedding (OLE) technology. The OLE technology is currently not translated to a PSMM.

The PSMMs represent the abstract syntax of a domain specific language (DSL) family for describing technical aspects of system interfaces, expressed in terms of Ecore models. The abstract syntax is supplemented with additional constraints and multiple concrete syntaxes. Constraints are defined on metaclasses and associations.

All metamodels are structured into the following packages: platform specific information (core), structure (interface static signature), interface behavior, non-functional properties and communication protocols.

The core package describes general platform specific aspects of an interface family under study, such as the interface type, remote methods and parameters. It provides the basic structure which can be instantiated, complemented and refined by other packages.

The **structure package** provides information about data structure and types supported by the modeled interfaces and their parameters. It further refines properties of parameters specified in the core package. Most systems distinguish between simple and complex types. Simple types have no further internal structure in contrast to complex types, which allow record-, message- or object-like structures, depending on particular realization.

Apart from simple or complex types, parameters can also have value ranges, which are often implemented as key tables. Parameters which reference the key table can only have values which are defined within the table. In PSMs the value range is modeled using `ValueRange` and `Entry` (Figure 9). With the `ValueRange` element, it is possible to create a table entry with key and its description. One can either model only keys (e.g., values of NORTH, SOUTH, EAST, and WEST in Figure 10) or keys with descriptions (e.g., document type: WA = Goods Issue, WE = Goods Receipt, WL = Goods Issue/Delivery in Figure 10). The element `Entry` inherits from the abstract concept `AnnotatableValue`.

The **communication package** describes information required to establish phys-

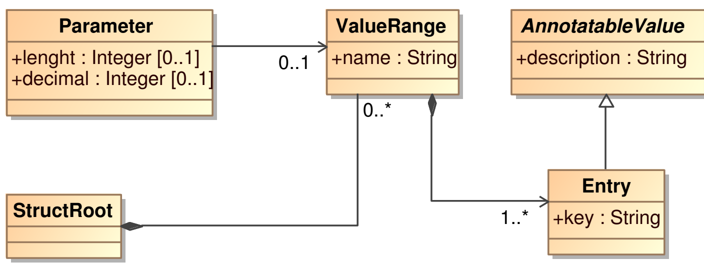


Figure 9: Value Range Definition

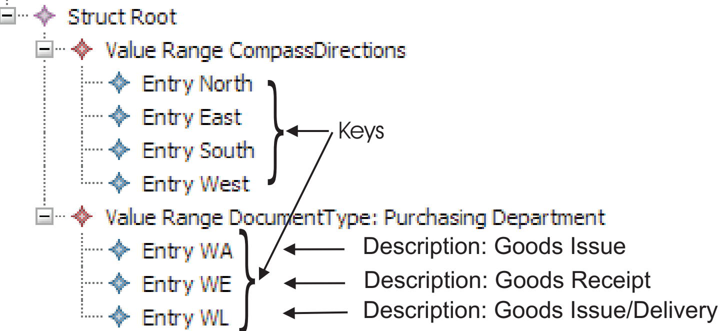


Figure 10: Value Range Example

ical connection with the system under study, such as access-parameter, transport layer, character set and encoding.
The **property package** describes non functional properties associated to or required by the interface under study, such as performance, authorization, security and logging. Further information can be seen in chapter 7. The **behavior package** describes detailed characteristics of interfaces, methods, objects and parameter. The information is modeled via OCL constraints.
The `AnnotatableElement`, `AnnotatableFunction` and `AnnotatableValue` (Figure 11) mark model elements within a PSMM. All instances of elements which inherits from these abstract classes can be marked with concepts and instances of concepts from an ontology (see chapters 5 and 6).

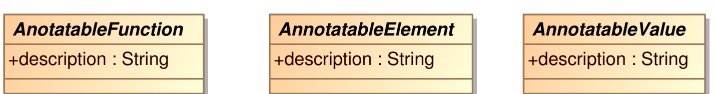


Figure 11: Annotatable Model Elements

The purpose of the PSMMs can be summarized as follows:

- Description of software system interfaces at technical level, including interface static signature (types), interface behavior, its communication protocols and non-functional properties
- Modeling of system interfaces using EMF editors, as well as automatic model extraction
- Interface documentation
- Client code generation
- Basis for model transformation to PIMM for the purpose of facilitating conflict-analysis process

Ecore serves as a metamodel for the PSMMs. Consequently the PSMMs are at the M2-level of the linear MOF hierarchy. The elements of the PSMMs are linguistic instances of the Ecore metamodel (see [13]). In other words, PSMMs conform to the Ecore metamodel. The most important concepts of Ecore that are used within the PSMMs, are EClass, EPackage, EReference, EAttribute, and EDataType. With these elements it is possible to model structural metamodels. PSMMs define a domain specific language (DSL) for description of software system interfaces, specifying the abstract DSL syntax. Interface models at the MOF M1-level conform to PSMMs at the MOF M2-level. PSMM elements thus build the vocabulary for interface descriptions. The system under study is provided at the MOF M0-level (e.g., an SAP R/3 instance). PSMMs are constructed over a classification abstraction of the existing system interfaces. An overview of this hierarchy is given in Figure 12.

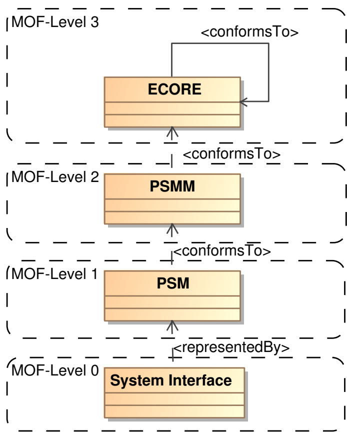


Figure 12: MOF-Level

A survey of different metamodel layers is given in [10]. Every PSMM is hierarchic and tree-based. This is necessary for the development of tree-based editors, which are implemented using Eclipse Modeling Framework (EMF) [8].

The following prerequisites and constraints have been observed, related to the use of EMF:

- The name of modeled elements must consist of letters, digits and underscores only
- The name must begin with a character.
- No multiple inheritance is allowed
- Names of all `EClasses` must be unique
- Attribute names must be unique within inheritance relationships.
- Names must not match with Java keywords.
- Names must not begin with the name of a Package.

The EMF validation framework enables to check models against their metamodel. The EMF tree-based editors support cardinality constraints check by default. For example, if the maximal number of child elements is reached, it is not possible to create more. Logical failures have to be catched with OCL constraint checks. The constraints are formulated via annotations, directly in the metamodels in the context of an EClass. An OCL console helps to develop the constraints before putting them in the metamodels. The OCL console allows to test each constraint with direct response from the interpreter and useful additional features such as code-completion. The validation of a PSM can be executed within the editor.
In the following we describe metamodels that were developed to support the following platforms: SAP R/3 ERP systems, relational databases, J2EE applications/components, Web services and XML flat files.

3.1 SAP R/3

SAP R/3 is one of the most widely used enterprise resource planning (ERP) systems. Most of the medium and major sized enterprises have at least one SAP R/3 system running. This, together with the fact that all consortial partners perform SAP integration tasks, influenced our decision to support SAP R/3 directly within a separate metamodel. Furthermore, its platform specific details cannot be easily aligned with other modeled platforms.
This metamodel, as well as other PSMMs, does not represent one-to-one reverse engineered image of an R/3 system – this is impractical and unnecessary. Instead, R/3 ERP system is treated as a black box and only features and details relevant for its integration are modeled. The platform specific metamodel elements consist mainly of classes referring to the remote accessible methods, so-called Business Application Programming Interfaces (BAPI) and Intermediate Documents (IDOC), as well as their types and communication channels supported.

IDOCs are structured files in ASCII text format. They are not self-describing, so the schema of an IDOC file is defined within an IDOC Type located in the SAP System. IDOC is a file-based communication concept since SAP release 2.2, which was long before remote functions (such as BAPIs) were introduced. It is based on business documents, which are exchanged in electronic way. The original idea was to avoid media disruption. The `IDOCType` element within the metamodel consists of a hierarchic tree with `Segments` and `Fields` inside the `Segments`. Each line of an IDOC file consists of one `Segment` with its identification number and `Fields` with concrete data afterwards.
BAPIs are standardized and remote accessible methods associated to SAP business objects. They are compliant with specifications of the Open Applications Group (OAG) and the Common Object Request Broker Architecture (CORBA) of the Object Management Group (OMG). Furthermore BAPIs are remote accessible functions which have input and output parameters. There are three different parameter types: structured (`StructType`), table (`TableType`) and field types (`FieldType`). `StructTypes` and `TableTypes` can consist of `FieldTypes`. `FieldTypes` are atomic and have no internal structure. The SAP models are automatically instantiated using an extractor component by querying the SAP R/3 Business Object Repository (BOR) service. As it is not possible to get the related type description, every parameter has a containment relation to its own type. Import parameters can be further marked as mandatory.
BAPIs can be instance creating (e.g., Create or CreateFromData), which means that new data instances are added to the system. If a BAPI retrieves information about a specific data instance, it is instance dependent (e.g., SalesOrder.GetDetail).
Standard SAP R/3 instance has several hundred BAPIs and IDOCs available. In the metamodel they are sorted into business components (`BusinessComponent`) and business objects (`BusinessObject`) (see Table 3 for examples). Each BAPI must have at least one import and export parameter. Every BAPI reports exception and success information through the Return parameter. Detailed information are accessible using BapiService.MessageGetDetail() and BapiService.ApplicationLogGetDetail() functions.

3.1.1 SAP R/3 Core Package

The core SAP R/3 package is given in Figure 13. The entry point to the hierarchic model is the element `SAP R/3`. It represents a concrete SAP R/3 system installation. The `SAP R/3` consists of at least one SAP R/3 Interface (`SAP_R3_Interface`), which serves as a container for BAPIs and IDOCs. The interface concept was adopted from the Java Interface, which declares existing methods within a class. This element consist of the elements `Access`, `SAPBusinessComponent` and `IDOCType`.
The PSM model can be generated semi-automatically. The whole parameter and structure part can be build automatically through extraction of needed information using the SAP Business Object Repository (BOR). Information about the structure of BAPIs can be extracted programatically using SAP Java Connector

Main Business Components	**Business Objects**
SAP BASIS (Basis Technology)	CATimeSheetManager
	CATimeSheetRecord
	EmployeeCATimeSheet
SAP HR (Human Resources)	PTimeOverview
	TimeAvailSchedule
SAP FI (Financials)	Company (Gesellschaft)
	Debtor (Debitor)
	Vendor (Lieferant)
	Customer (Kunde)
	CostCenter(Kostenstelle)
SAP LO (Logistics)	Material
	MaterialBOM (Materialstückliste)
	MaterialGroup (Warengruppe)
	ProductCatalog (Produktkatalog)
	SalesOrder (Kundenauftrag)
	PurchaseOrder (Bestellung)
	PurchaseRequisition (Bestellanforderung)
	WarehouseStock (Lagerbestand)

Table 3: Example for Business Components and Objects

(JCO) library, and IDOC type definition can be extracted indirectly with the BAPI IDOCTYPE_READ_COMPLETE. The BAPIs RFC_FUNCTION_SEARCH, RFC_FUNCTION_SEARCH_WITHGROUP, RPY_BOR_TREE_INIT and RPY_OBJECTTYPE_READ are useful for searching the needed BAPIs or building a customized BAPI Explorer. The following BAPIs are useful for the information exchange over IDOCs: IDOC_INBOUND_SYNCHRONOUS (single IDoc per call), INBOUND_IDOC_PROCESS (standard in tRFC, batch of IDocs allowed/ recommended), IDOC_INBOUND_SINGLE (single IDoc per call) and IDOC_INBOUND_ASYNCHRONOUS (standard in tRFC batch of IDocs allowed/ recommended). Finally, the following BAPI is useful to acquire information about the Interfaces: HelpValues.GetList() gets information about the valid values of parameters.

3.1.2 SAP R/3 Communication Package

The `CommunicationChannel` and `Access` elements belong to the package communication, they are linked to `SAP_R3_Interface`. All information about the physical access is collected within this package, including username, password, hostname, language, system number, sap client and the communication channel (`CommunicationChannel`). The channel can be synchronous (sRFC), asynchronous (aRFC), transactional (tRFC) or queued RFC (qRFC), simple file transfer or via email. The concrete communication channel depends on the kind of information exchange. IDOCs are exchanged over file transfer, email or tRFC, while BAPIs are accessed over sRFC. The concrete communication channel for IDOCs belongs to the SAP R/3 settings (i.e. file path). In european SAP R/3 systems there is usually an ISO-8859-1 character encoding, this conforms to the SAP code page 1100 for western-european languages.

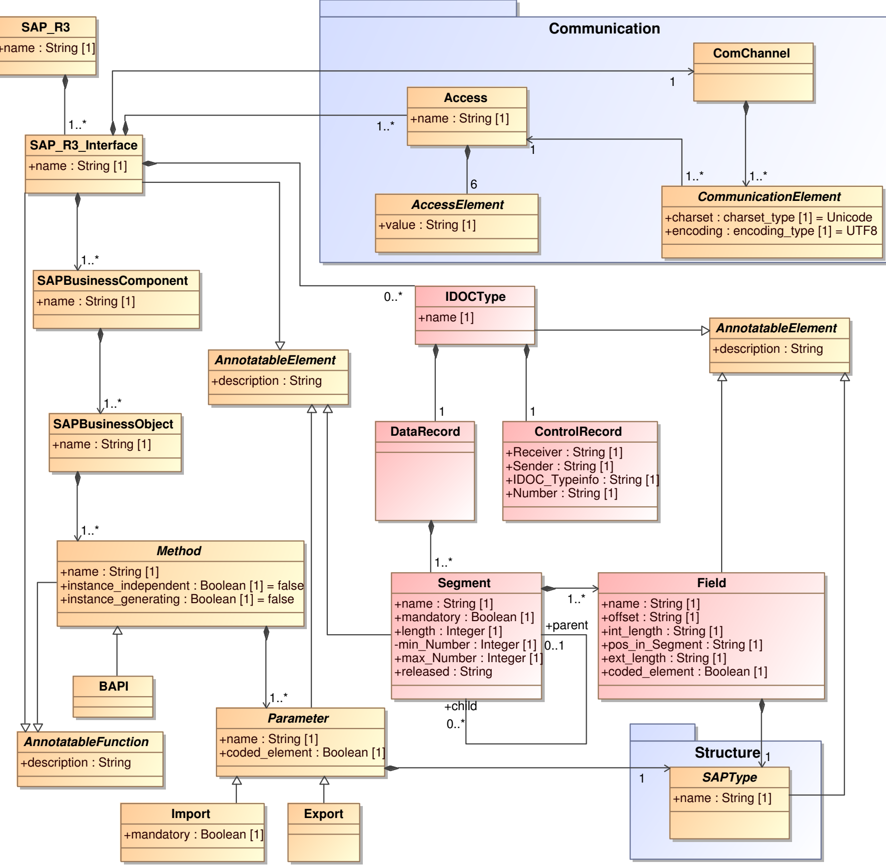


Figure 13: SAP R/3 PSM core package

3.1.3 SAP R/3 Structure Package

The structure package of the SAP R/3 metamodel is given in Figure 15. An `Import` and `Export` parameter can consist of three different types: structured (`StructType`), table (`TableType`) or field types (`FieldType`). The field type is atomic, structured and table types consist of at least one field type. Table and structured types only consist of field types. Field types are Java Connector (JCO) types, more precisely they are wrapper classes. JCO offers a library for Java which supports the access to SAP R/3 systems. JCO converts SAP R/3 types to Java types and vice versa.
SAP R/3 differentiates between import, export and table parameters. The direction of import and export parameter are self-explanatory, but table parameters can be used in import or export direction, depending on the modeled BAPI. This means that the platform specialist has to know parameter direction, that is, wheter a BAPI uses the table parameter for import or export.

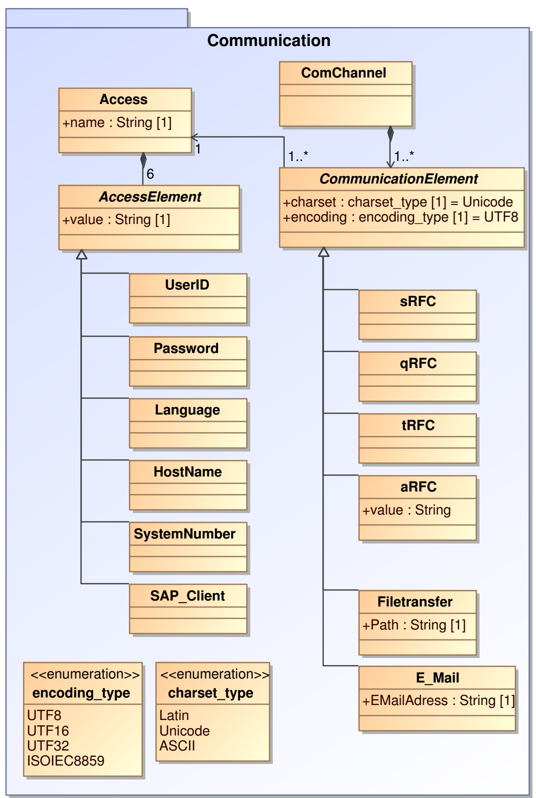


Figure 14: SAP R/3 PSM communication package

Field types can have a value range, where valid parameter values are predefined within a key table. The value range can be modeled with the elements `ValueRange` and `Entry` and `FieldTypes` can further reference them.
The SAP PSMM uses type system based on Java (JCO) types. The conversion schema can be seen in Table 4.

3.1.4 SAP R/3 Constraints

1. The name of each import or export parameter has to be unique within the same method.
   context Method inv:
   $self.has_Param- > forAll(c1, c2|c1 <> c2impliesc1.name <> c2.name)$

2. The name of each BAPI and IDOC type has to be unique within the same Business Object.
   context SAPBusinessObject inv:
   $self.has_Method- > forAll(c1, c2|c1 <> c2impliesc1.name <> c2.name)$

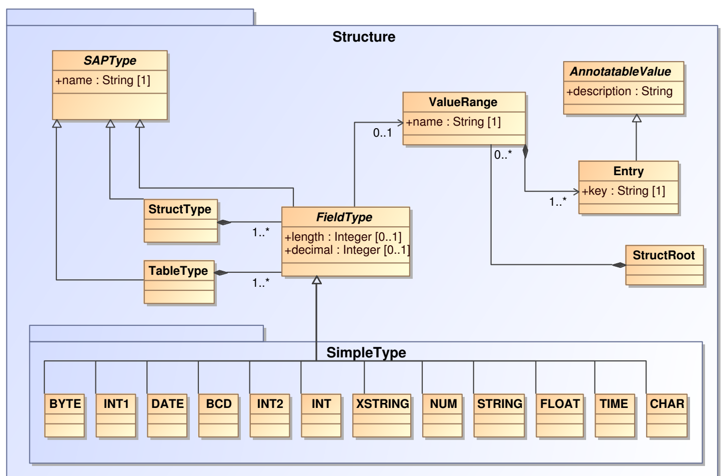


Figure 15: SAP R/3 structure package

SAP	ABAP	JCO	JAVA
ACCP	N(6)	NUM	STRING
CHAR n	C(n)	CHAR	STRING
CLNT	C(3)	CHAR	STRING
CUKY	C(5)	CHAR	STRING
CURR n,m	P((n+1)/2)DECIMAL m	BCD	BigDecimal
DEC n,m	P((n+1)/2)DECIMAL m	BCD	BigDecimal
DATS	D(8)	DATE	Date
FLTP	F(8)	FLOAT	Float
INT1	X(1)	BYTE	Byte
INT2	X(2)	BYTE	Byte
INT4	X(4)	BYTE	Byte
LANG	C(1)	CHAR	STRING
NUMC n	N(n)	NUM	String
PREC	X(2)	BYTE	Byte
QUAN n,m	P((n+1)/2)DECIMAL m	BCD	BigDecimal
RAW n	X(n)	BYTE	Byte
TIMS	T(6)	TIME	Date
UNIT	C(n)	CHAR	STRING
VARC n	C(n)	CHAR	STRING
LRAW	X(n)	BYTE	Byte

Table 4: SAP/JCO/Java type conversion

3. BAPI has the communication channels sRFC, qRFC, aRFC or tRFC.
4. IDOC has the communication channels file or email.
5. Table parameter is either import or export, not both at the same time

within the same BAPI.

3.1.5 SAP R/3 Model Example

The BAPI modeled in this example (Figure 16) has a `SAP_R3_Interface` with the name IDES. The character set is Unicode with UTF-8 encoding, this equates to SAP code page 6100. The release of the system is 4.6 C. The `Communication-Channel` is synchronous RFC (`sRFC`) and the access parameters for the login to the SAP system are modeled. The owner of the modeled BAPI Requisition Items is the `BusinessObject` Requisition and the `BusinessComponent` Logistic. The `Export` parameter has the `FieldType` `CHAR` with the name NUMBER and the length 10. The `Import` parameter Requisition Items has a `TableType`, with the `FieldTypes` PREQ_NO (`CHAR`), PREQ_ITEM (`NUM`), DOC_TYPE (`CHAR`), MATERIAL (`CHAR`), PUR_MAT (`CHAR`), PLANT (`CHAR`), QUANTITY (`BCD`) and UNIT (`CHAR`). The import parameter is further mandatory (not shown in Figure 16).

3.2 Relational Database Systems

Relational Database Systems (RDBS) are often used to persist structured data within applications. For example, the open source Web shop system XT-Commerce uses MySQL database to store order, products, customer, transaction, etc. data. The database access is direct access to the data-layer, without the use of remote functions or wrappers. Therefore, a prerequisite is to write a query against data model. The query can be written manually, or designed with the help of a query builder such as ER-Win.
The platform specific metamodel for relational database systems is used to model data being retrieved or persisted by the relational database engine. Therefore, parameters and structure of SQL queries are the focus of this metamodel.

3.2.1 Relational Database Core Package

The root element of the PSMM is `DBMS`, which collects SQL interfaces (Figure 17). The element `SQL_Interface` further aggregates queries within the `Query` element. A query can be a stored procedure (`StoredProcedure`) or standard SQL query such as `Create`, `Select`, `Update`, `Insert` or `Delete`. In order to abstract from SQL syntax as much as possible, all queries are specialized into Create, Read, Update or Delete (CRUD). A query consists of parameters which can have input direction (e.g., parameters of a WHERE clause) or output direction (e.g., results of a SELECT clause).
We support automatic model generation for given prepared SQL statements. A prepared statement is an ordinary SQL statement, with question marks substituting (marking) input parameters. A SQL parser converts prepared statements in conjunction with the access parameters to a model. Parameter type information is automatically extracted from the underlying database schema.

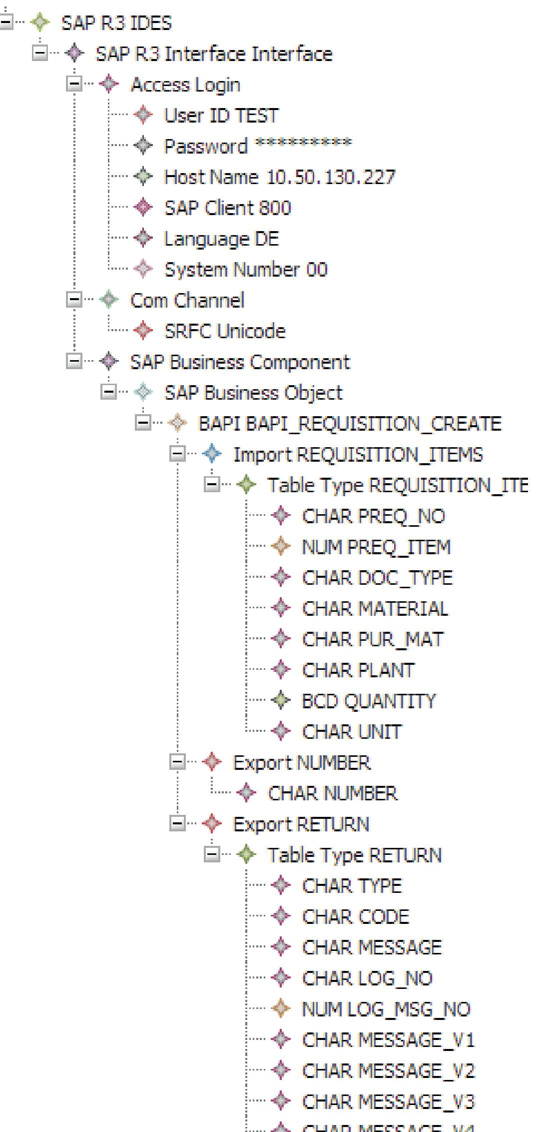


Figure 16: SAP R/3 model example

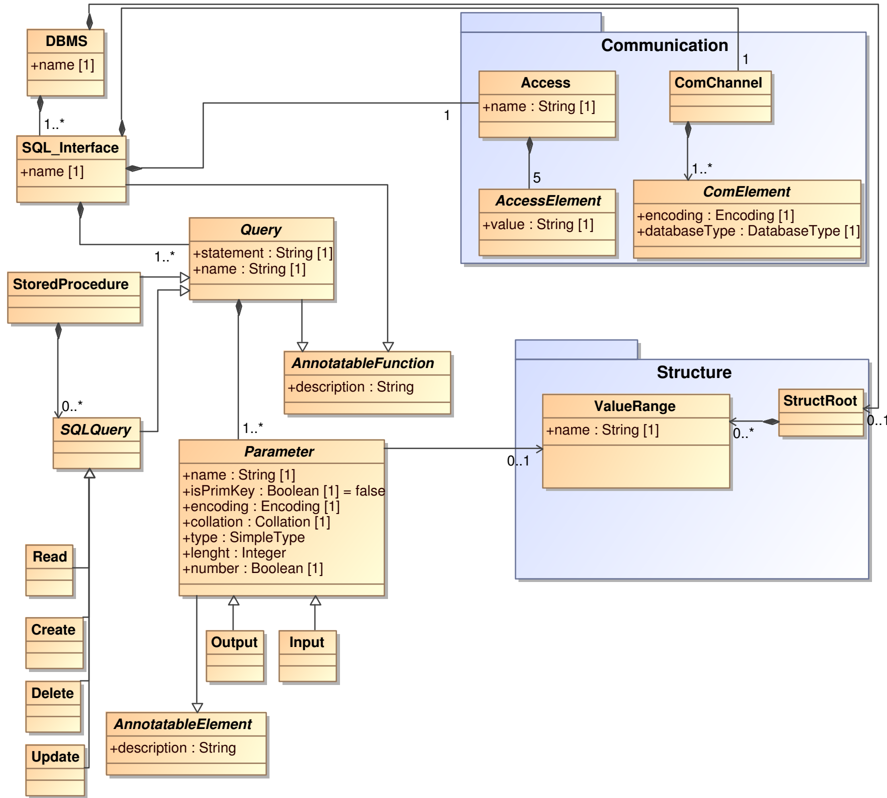


Figure 17: Relational database PSM core package

3.2.2 Relational Database Communication Package

The `Access` and `CommunicationChannel` elements collect information about the access to a database system. `Access` elements are user, password, host, database name and port. `CommunicationChannels` are Java Database Connectivity (`JDBC`), Open Database Connectivity (`ODBC`) or structured files like comma separated values (`csv`) or tab delimited (`tsv`). The encoding of a RDBS is very important for the transfer of data elements. Within a database, every column is associated with a character set for the information about the encoding and a collation. The column inherits the encoding and the collation from the table and the table again from the database, if the attributes where not overridden. The collation is important for the comparison and sorting of two text type columns. For example the string value "Ü-Ei" will be internally transformed to "Ue-Ei", by the use of the collation latin1_german1_ci during the sorting.

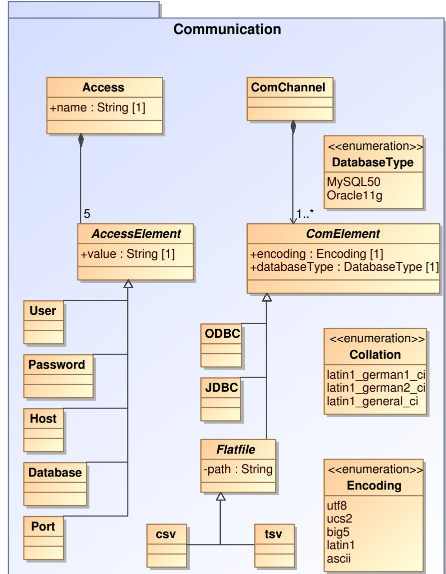


Figure 18: Relational database PSM communication package

3.2.3 Relational Database Structure Package

Every `Parameter` has a specific name, which is equal to its name given in the SQL statement. The default value for the encoding is UTF-8, the encoding and the collation can be specified for each `Parameter`. Every `Parameter` has a specific length.
The structure part consist of `SimpleTypes` for the description of `Parameters` and `ValueRanges` with `Entries` for `Parameters`. A `Query` collects the `Input` and `Output` parameter with `SimpleTypes`. Every `Query` consists of a prepared statement as annotation. Every `Parameter` must appear in this statement. The PSMM uses JDBC types (Figure 19), and conversion schemas for supported database engines have to be provided. Exemplary, MySQL conversion rules are given in Table 5. We additionally support Micorosft SQL Server and Oracle with type conversion.

3.2.4 Relational Database Constraints

1. The name of each SQL Interface has to be unique.
2. All Queries must have at least one output parameter.
3. The Create, Delete and Update Query must have at least one input parameter.
4. The parameter statement must be a wellformed SQL-statement.

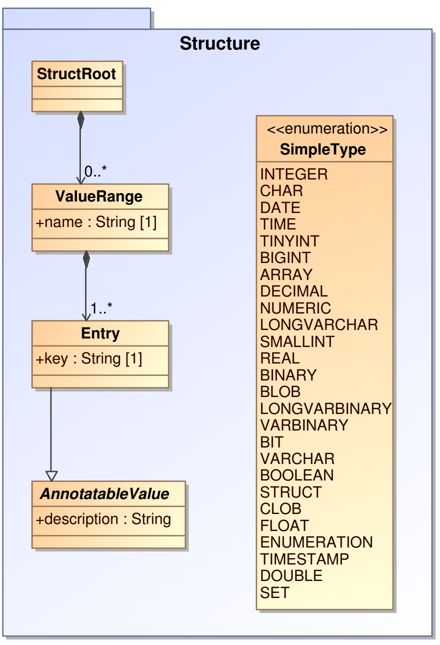


Figure 19: Relational database PSM structure package

3.2.5 Relational Database Model Example

The following example shows an XT-Commerce web shop system. XT-Commerce is released under General Public License (GPL). It is coded in PHP in conjunction with a MySQL database and supports the web based administration of products, customers, orders, manufacturers etc.
In this example we want to export all orders, total value of each order and to update the status of an order when it is paid. It is necessary to analyze the database schema to find the tables with the needed information manually. The XT-Commerce web shop consists of 124 tables with mostly self describing names. The tables of interests for our scenario are orders and orders_total. The table orders stores information about the customer, delivery name and address, billing name and address, payment method, credit card information, currency, shipping method etc. The table orders_total stores different values like tax, shipping cost and the total value of an order. The total value of an order is needed to check the incoming payment against this value. The table order_status defines the value range for the parameter order_status: "Pending", "Processing" and "Delivered". All payed orders are to be marked with the status "Processing", unpayed orders remain in status "Pending". This operation is realized with three SQL queries:

1. The first SQL-statement exports all orders with a given status out of the XT-Commerce system:

MySQL-Type	**JDBC-Type**
BIGINT	BIGINT
BIGINT UNSIGNED	BIGINT
BINARY	BINARY
BIT	BIT
BLOB	BLOB
BOOL	BOOLEAN
CHAR	CHAR
DATE	DATE
DATETIME	LONGVARCHAR
DOUBLE	FLOAT
DECIMAL	DECIMAL
ENUM	LONGVARCHAR
FLOAT	FLOAT
INT	INTEGER
INT UNSIGNED	INTEGER
LONGBLOB	BLOB
LONGTEXT	VARCHAR
MEDIUMBLOB	BLOB
MEDIUMINT	INTEGER
MEDIUMINT UNSIGNED	INTEGER
MEDIUMTEXT	LONGVARCHAR
SET	LONGVARCHAR
SMALLINT	SMALLINT
SMALLINT UNSIGNED	SMALLINT
TEXT	LONGVARCHAR
TIME	TIME
TIMESTAMP	TIMESTAMP
TINYBLOB	BLOB
TINYINT	TINYINT
TINYINT UNSIGNED	TINYINT
TINYTEXT	VARCHAR
VARBINARY	VARBINARY
VARCHAR	VARCHAR
YEAR	INTEGER

Table 5: MySQL Conversion Schema

Name: getOrders
SELECT * FROM ORDERS WHERE order_status = ?

2. The second SQL-statement exports the total value for a given order:
   Name: getOrdersTotalValue
   SELECT value FROM orders_total WHERE (class = 'ot_total') AND (orders_id = ?))

3. The third SQL-statement updates the order_status parameter of an order with a given orders_id for received transactions:
   Name: UPDATE
   orders SET order_status = ? WHERE orders_id = ?

The above SQL statements are prepared statements which have to be modeled within the PSM (Figure 20). Platform specific types must be converted from database specific to JDBC types.

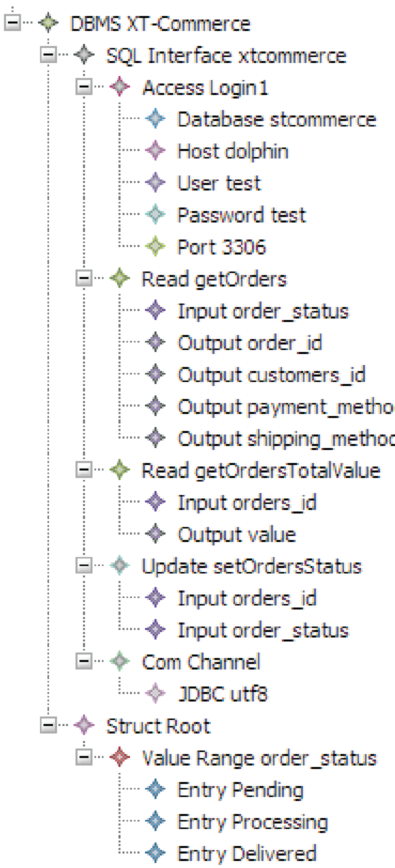


Figure 20: Relational database PSM example

3.3 Java 2 Platform, Enterprise Edition (J2EE)

J2EE is a wide-spread middleware and development platform for web and business applications. Therefore it is important to provide a PSMM to describe interfaces of J2EE components. The focus of the proposed metamodel is access to Enterprise Java Beans (EJB) interfaces. In the current EJB specifications there are three types of components: session, entity and message-drive beans. Session beans encapsulate business logic coded in Java. Message beans serves as entry points for asynchronous communication and computation. Entity beans encapsulate access to structured persistent data. Remote methods offered by

EJB components can be accessed via remote, local or home interfaces. The EJB components are deployed and executed within application servers, for example JBoss or Glassfish.

3.3.1 J2EE Components Core Package

The root element of this metamodel is `J2EE`, which aggregates `J2EE_Components`. They further collect (`EnterpriseJavaBean`) elements. Currently only session beans are supported. They expose J2EE Interfaces which can be remote, local or home, with associated `Input` and `Output` parameters. `Input` parameter will be assigned as an array to the `Method`, `Output` parameter are the return parameters of a method.

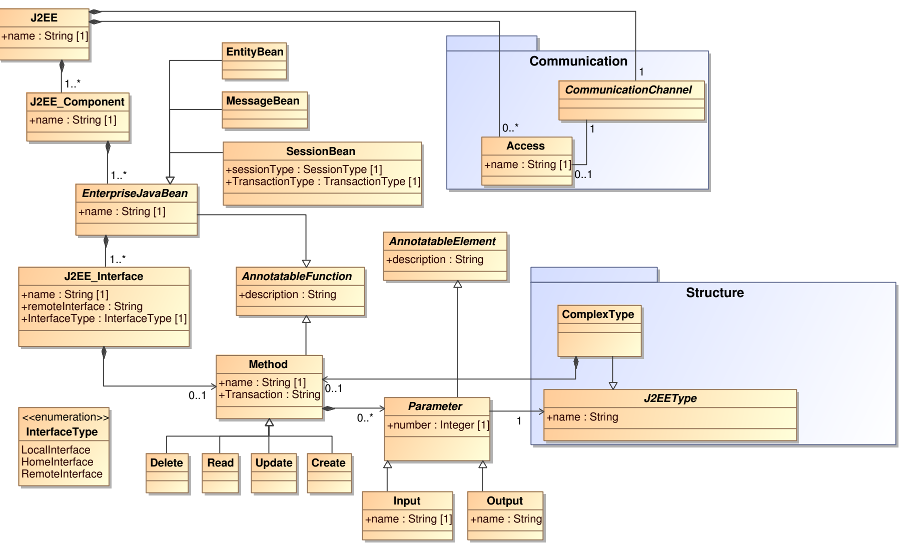


Figure 21: J2EE components PSM core package

3.3.2 J2EE Components Communication Package

The `Access` and `CommunicationChannel` elements collect information about the physical access to a J2EE platform/component. The `Access` element collects information about the user name and password, if necessary.
Currently only the RMI-IIOP (Remote Method Invocation over the Internet Inter-Orb Protocol) communication channel is supported. Information about the system can be accessed over the Java Naming and Directory Interface (`JNDI_Property`) using the following elements: Java Naming Factory URL PKGS, Java Naming Provider URL, Java Naming Factory Initial, JNP Socket Factory and JNP Timeout.

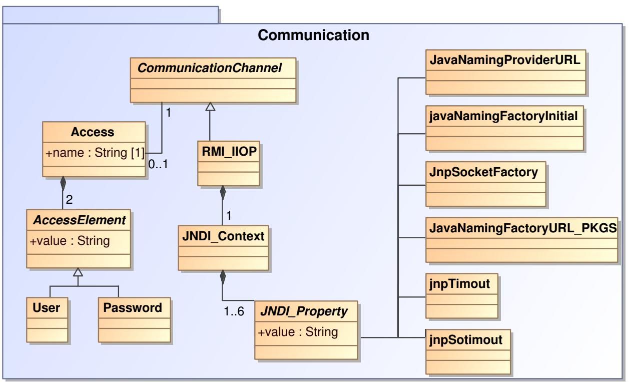


Figure 22: J2EE components PSM communication package

3.3.3 J2EE Components Structure Package

A J2EE `Method` (e.g., remote interface method) can have unlimited `Input` parameters, but only one `Output` parameter. The type of a `Parameter` can be `CollectionType`, `ComplexType` or `SimpleType`. A `ComplexType` can consist of further `ComplexTypes`, `SimpleTypes`, Methods or Collections. `SimpleTypes` are native Java types. Within a `ComplexType` it is important to know the name and the type of a `Parameter`. Therefore a `ComplexType` consists of `Fields`, storing the name of a `Parameter`. The `Field` is then related to the type of a parameter (`J2EEType`). `CollectionType` could be a list, set, map or a queue. `Parameter` and `Field` of a `ComplexType` can have `ValueRange`.

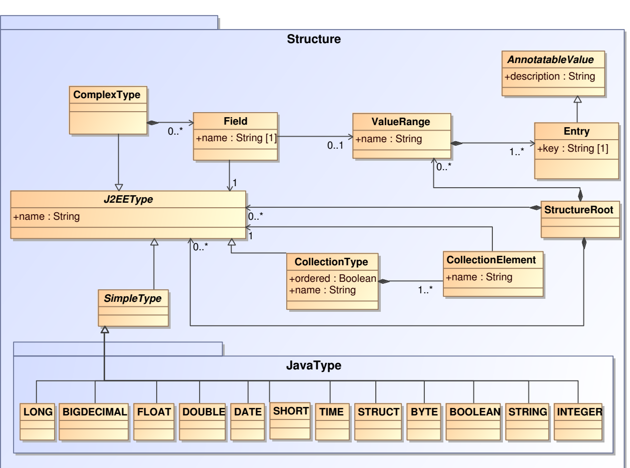


Figure 23: J2EE components PSM structure package

3.3.4 J2EE Components Constraints

1. The name of each J2EE Interface has to be unique.
2. A method can be either owned by a complex type or a J2EE Interface.
3. A parameter can only have a value range when it has a simple type.

3.3.5 J2EE Components Model Example

The following scenario shows the modeling of a J2EE component (EJB 2.1), representing part of the wholesaler software system. The interface CheckedBookEJB delivers information about the availability of books. The interface provides the following methods:

Method signature	**Description**
CheckedBookInterface checkBook(String ISBN)	The method checkBook gets a ISBN number of type String as input and provides an object of type CheckedBookInterface

Table 6: checkBook

Method signature	**Description**
String getISBN()	Provides the ISBN number of the checked book
Integer getAvailable()	Provides a Boolean value to the availability of the checked book
Boolean getQuantity()	Provides the number of available books
Date getDeliverDate()	Provides the earliest delivery date

Table 7: CheckedBookInterface

Beside the structural information concerning methods, it is also important to model information about the application server (in this example JBoss 5.0.0). The following information are necessary for the client to execute the provided methods on the application server. With all this information it is possible now to create the PSM for the scenario (see Figure 24).

Configuration	**Value**
Initial Context Factory	*org.jnp.interfaces.NamingContextFactory*
Provider URL	*jnp://wombat-vm-2.cis.cs.tu-berlin.de*
URL PKG Prefixes	*org.jboss.naming:org.jnp.interfaces*
Socket Factory	*org.jnp.interfaces.TimedSocketFactory*

Table 8: JBOSS Configuration

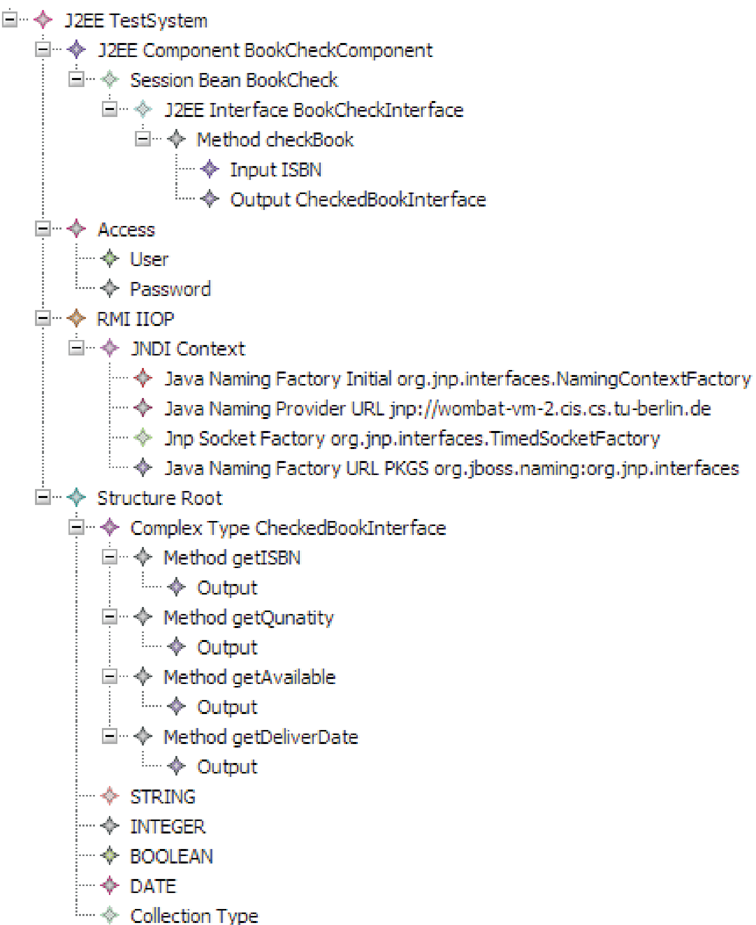


Figure 24: J2EE component model example

3.4 Web Services

Web Services are an important SOA technology that offers service-based functionality without depending on a special programming language. Thus Web Services and their clients can be implemented with different languages like Java, C-Sharp or Ruby and remain compatible. A Web Service interface description is formally described by the XML-based Web Service Definition Language (WSDL), which conforms to the WSDL schema [1]. To process a WSDL-file inside the MOF-based metamodel hierarchy one has to transform all relevant WSDL information into a M1-level model which conforms to the M2-level WSDL schema-derived metamodel. The current WSDL metamodel considers only SOAP version 1.1 as the message protocol and HTTP as the transport protocol.

3.4.1 Web Services Core Package

The Web Services metamodel (Figure 25) starts with the general `ModelRoot` which aggregates one `StructureRoot`, one `CommunicationRoot` and varying `ServiceDefinitions`.
A `ServiceDefinition` aggregates different WSDL elements: `Services`, `Bindings`, `PortTypes` and `Messages`. A `PortType` is the abstract description of a set of exposed `Operations` and its corresponding `InputParameters`, `ExportParameters` and `FaultParameters` and optional `ParameterOrderItem`. The three different parameter types are generalized by an abstract `Parameter`. Every parameter has an association to exact one `Message`, which is itself divided into several `MessageParts`. A MessagePart represents a data structure, described by an XSD schema. For this reason, a practical lite version of the XSD schema is modeled in the separate *XSDLite* metamodel (see Chapter 3.6), also used by XML flat files.
A `Service` represents a single Web Service and aggregates a `Port`. A Port holds the communication address (`location`) and an association to exact one `Binding` but vice versa can be associated to more then one Port. A Binding holds the protocol information (`transport`) and data encoding (`style`). Binding aggregates further SOAP-specific metaclasses (with prefix `SOAP`) which add SOAP-specific details for `Operations`, `Parameters` and `MessageParts` (Figure 26).
All these metaclasses are generalized by `NamedElement`, which holds the attributes `name` and a optional `documentation`.

3.4.2 Web Services Communication Package

The communication package (Figure 25) includes the `MessageExchangePattern` and access information covered by `SecurityData`. Further communication metaclasses can be aggregated later under the `CommunicationRoot` metaclass.

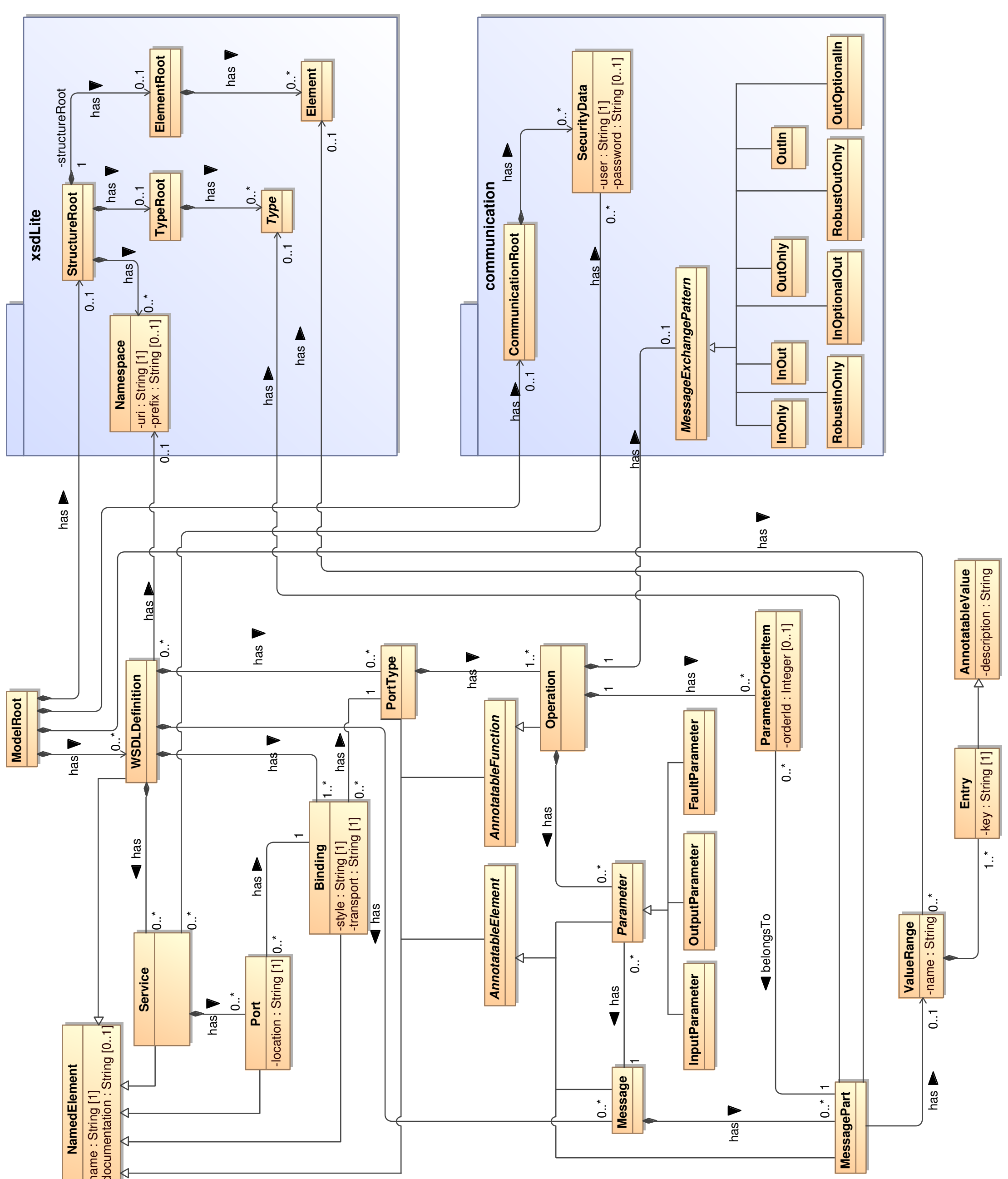


Figure 25: Web Services Metamodel

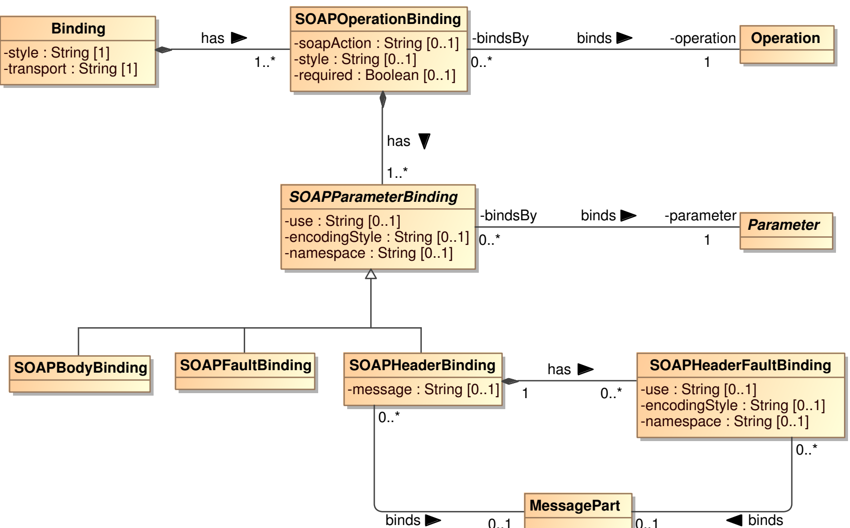


Figure 26: SOAP specific excerpt of PSMMWSDL Metamodel

3.4.3 Web Services Structure Package

The WSDL structure package is covered by the *XSDLite* metamodel which is described in chapter 3.6.

3.4.4 Web Services Constraints

- exactOneXSDReference: A MessagePart instance must have either a type-reference or a element-reference (XOR)
  `context MessagePart: (self.type->sum()+self.element->sum())=1`

3.4.5 Web Services Model Example

This example is derived from a WSDL file which describes a Web service from the event service domain, offering `createEvent` Web method.

- Model Root
  - Structure Root
    - Element Root
      - Element decisionTableName
      - Element execution
      - Element parameters
      - Element name
      - Element value
      - Element return
      - Element createEvent
      - Element createEventResponse
    - Attribute Root
    - Type Root
      - Complex Type createEvent
        - Sequence 1
      - Xsd Base Type string
      - Complex Type eventParameterInput
        - Sequence 1
      - Complex Type createEventResponse
        - Sequence 1
    - Namespace http://schemas.xmlsoap.org/wsdl/
    - Namespace http://event.jaxws.service.eem.de/
    - Namespace http://www.w3.org/2001/XMLSchema
    - Namespace http://schemas.xmlsoap.org/wsdl/soap/
  - WSDL Definition EventServiceService
    - Service EventServiceService
      - Port EventServicePort
    - Binding EventServiceBinding
      - SOAP Operation Binding
    - Message EventService_createEvent
      - Message Part createEvent
    - Message EventService_createEventResponse
      - Message Part createEventResponse
    - Port Type EventService
      - Operation createEvent
        - Input Parameter
        - Output Parameter

Figure 27: Web Services Model Example

3.5 Extensible Markup Language (XML)

XML is a widely used standard for platform independent data representation. In the context of MOF the XML metamodel which includes a metamodel called *XSDLite* representing the XSD schema (Chapter 3.6) is at the M2-level. Domain specific XSD files are at the M1-level and real XML-data conforming to a concrete domain specific XSD from are at the M0-level. The XML metamodel (Figure 28) is divided into two packages: (1) `xsdlite` covering structural aspects of XML-data and (2) `communication` covering information for accessing XML data at the M0-level.

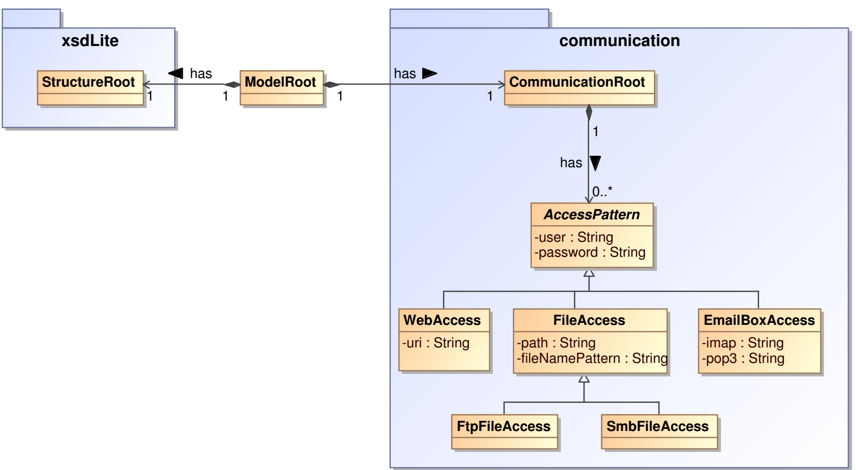


Figure 28: XML Metamodel (excerpt)

3.5.1 XML Core Package

The XML metamodel core package is subsumed by the `xsdlite` metamodel, described in Section 3.6.

3.5.2 XML Communication Package

The `communication` package depicted in Figure 28 includes all communication details concerning the access to XML data which is covered by the abstract metaclass `AccessPattern` including the two properties `user` and `password`. For file-based access (`FTPFileAccess`, `SMBFileAccess`) the special properties `path`, `user`,`password` and `fileNamePattern` can be modeled with `FileAccess`. Web-based XML sources can be modeled with `WebAccess` and email-based XML-transfer with `EmailBoxAccess`.

3.5.3 XML Structure Package

The structure part can be modeled with the generic XSDLite metamodel (chapter 3.6), a practical (lightweight) representation of the XSD schema.

3.5.4 XML Model Example

Since both XML and Web services metamodels use `xsdlite` as their structure package, we do not give XML model example, as it is very similar to Web service model example given in Figure 27, differing only in the communication part.

3.6 XML Schema Definition (XSD)

The XSDLite metamodel (Figure 29) is a practical (lightweight) representation of the XSD schema [3]. XSD is a language definition for describing structural and typing aspects of XML-based data. It replaces the former Document Type Definition (DTD). The XSDLite metamodel starts with the root element `StructuralRoot`, which aggregates one `AttributeRoot`, one `ElementRoot`, one `TypeRoot` and varying `Namespaces`. A Namespace represents an element for naming XSD elements uniquely. The three root metaclasses aggregate `Attribute`, `Element` and `Type`. The metamodel differ between three kinds of Types: `Simple Types`, `ComplexTypes` and `XsdAny`. The last is a special XSD construct for untyped elements. The SimpleType is divided into four sub metaclasses:

1. `XsdBaseType` represents original XSD simple types such as string, boolean, float, double, etc.
2. `List` represents a list of a concrete SimpleType,
3. `Union` represents a set of SimpleTypes and
4. `DerivedType` represents a restricted SimpleType e.g., a string conforming to a regular expression.

Restrictions are handled by `Facets`, which generalize all metaclasses in the `facet` package. A ComplexType aggregates a `Container`, which generalize `Sequence`, `All` and `Choice`. These three sub types have the same semantic as defined in a XSD schema (M1-level) concerning XML data (M0-level): in (1) Sequence all aggregated `ElementRefs` are defined by the `minOccurs` and `maxOccurs` attributes, in (2) Choice only one `ElementRef` is allowed, if (3) All, every Element referred by `ElementRef` must be instantiated. Finally, `Attribute` and `Element` hold the *real* data, hence these are sub classes from `AnnotatableElement`. Type and AnnotatableElement are sub classes from `NamedElement` which has a self reference `embeddedIn`. This optional reference represents local (embedded) Elements and Types known from XSD. An Element can only include a Type and vice versa.

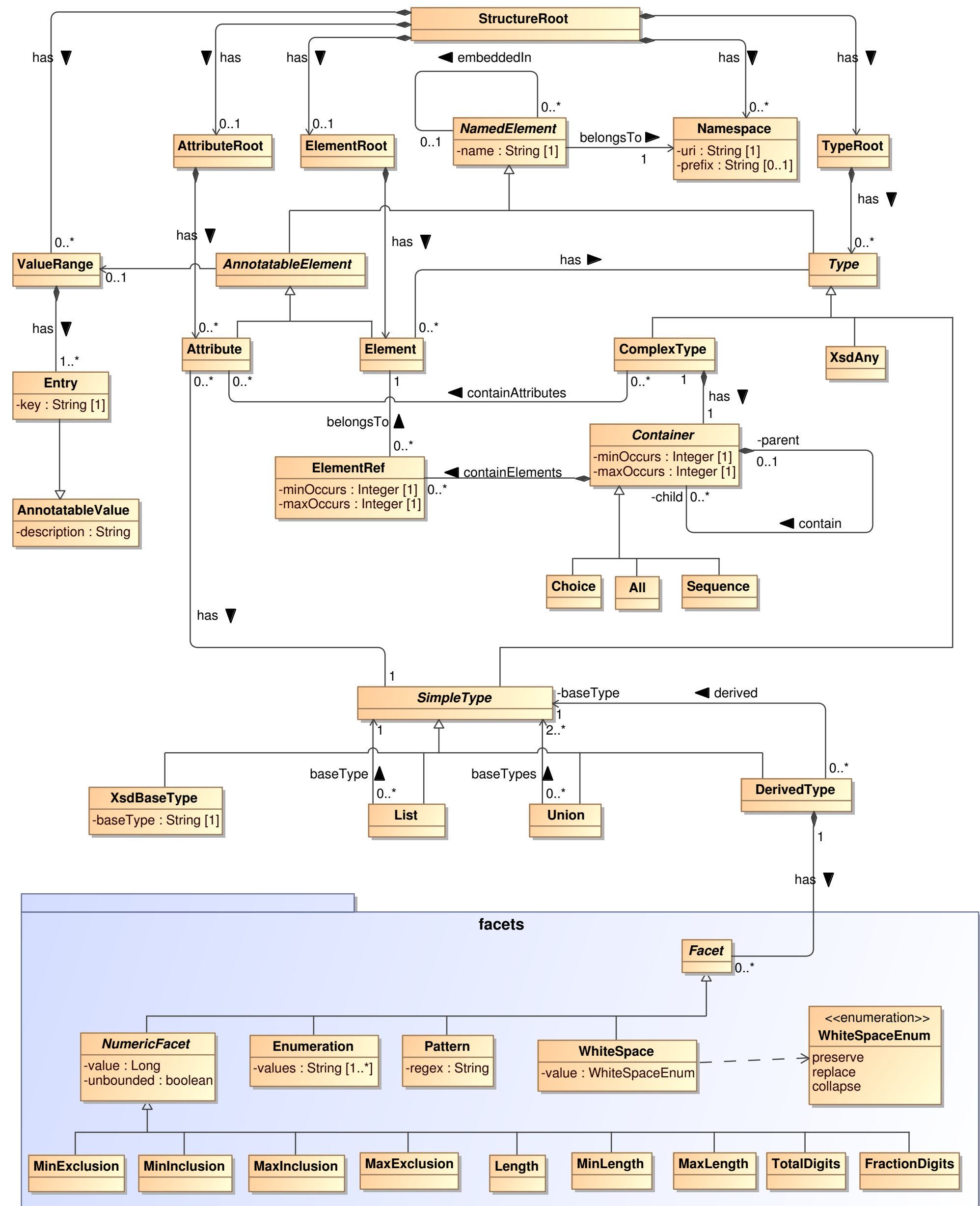


Figure 29: XSD-Lite Metamodel

4 Platform Independent Metamodel

The motivation for introduction of the Platform Independent Metamodel (PIMM) is to facilitate system interoperability by abstracting all platform specific heterogeneous interface details, thus enabling unique integration conflict analysis method, independent of platform-specific details of target systems. These details are divided into different parts: structure, semantic, behavior, (non-functional) properties and communication. This abstraction process is realized by a PSMM-to-PIMM model-to-model transformation realized by ATL [6]. In this way, based on a PSM, a new PIM will be created, hence for every PSMM there exists a set of transformation rules which translates PSMM into the common abstraction layer, the PIMM. As stated in [14] the PIMM eliminates the problems of integrating non-comparable interface descriptions. With the PIMM one is able to represent different platform-specific interface details on a common basis. Based on the technical interface descriptions given by PSMs, a PIM represents an aggregated view of all exposed interfaces of a specific real-world system. Interface details at the PIM level have to be of relevance for the conflict analysis, i.e. a PIM represents a subset of interface details which is machine processable by the conflict analysis tool. A PIM consists of transferred/derived information based on the underlying PSMs.

4.1 PIMM Core Package

The PIMM, like all other BIZYCLE metamodels, follows the EMF containment tree structure. Consequently, it includes a few 'helper nodes' with no specific meaning (e.g. `StructureRoot`) which only span different subtrees. At the PIMM-level an `Interface` represents a single system gateway which is able to handle data as input and/or output in one single step. Hence PIMM-Interfaces represent an abstraction for all platform specific operations, methods, functions, files etc. PIMM differs between two different Interface types according to their meaning: `FunctionInterface` and `DocumentInterface`. The former represents a call oriented Interface (e.g., a Web method), the latter a data structure based Interface (e.g., an XML-file). Every Interface has its own parent container, the system which exposes it. This container is called the `IntegratableElement`. Every Interface contains associations to different parts of the PIMM which will described in the following subsections. See basic PIMM metaclasses in Figure 30.

4.2 PIMM Structure Package

The structural package (given in Figure 31) includes the common type system with the abstract super metaclass `Type` and different sub metaclasses to express `SimpleType`s (String, Number, Boolean) as well as `ComplexType`s. The abstract metaclass `TypedElement` represents an element with a specific meaning (semantic) and is able to hold a real value at runtime, e.g., `ImportParameter`. A TypedElement has exactly one association to a type (1 cardinality) but vice

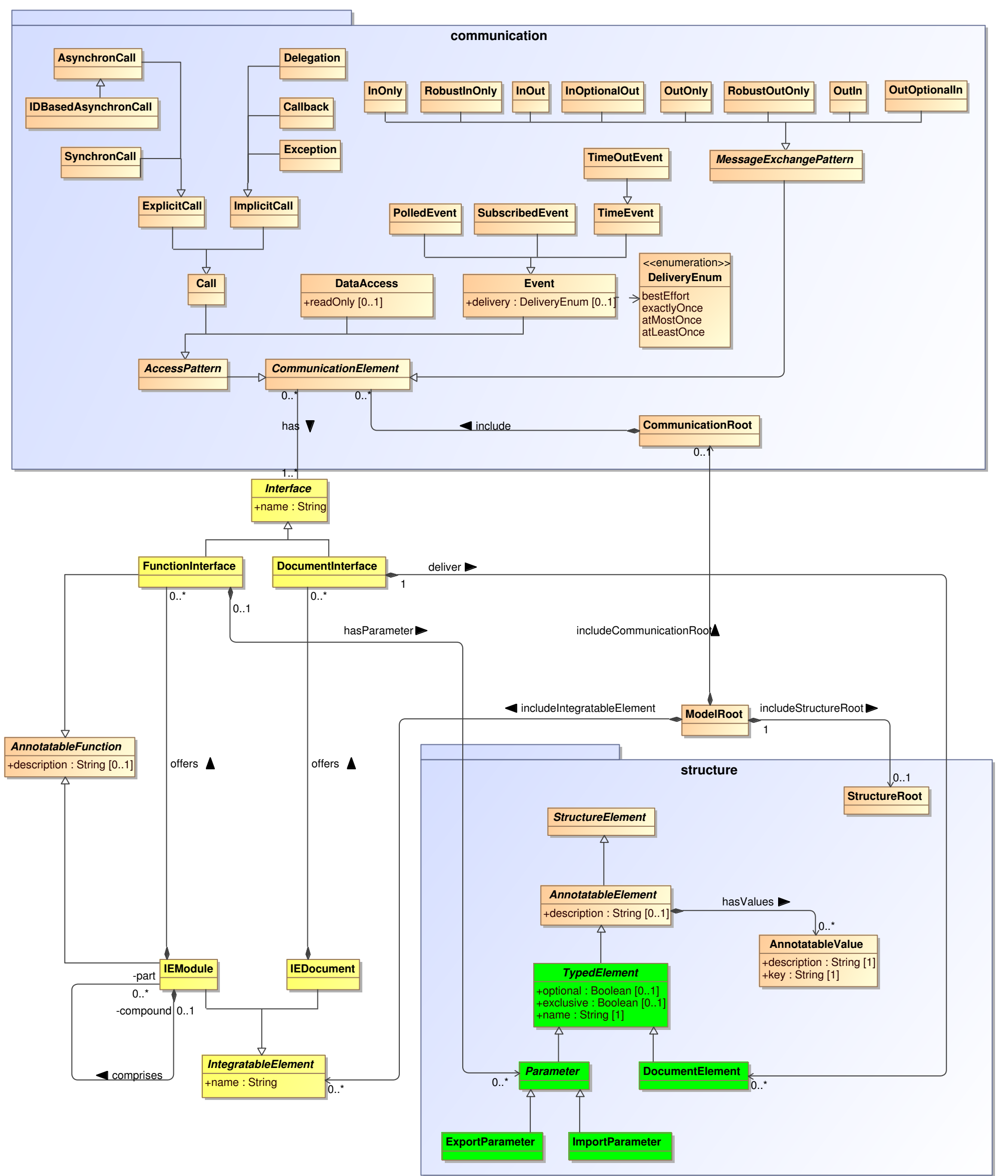


Figure 30: Platform independent metamodel overview

versa a type can belong to arbitrarily TypedElements (* cardinality), that means a Type can be ’reused’.
The TypedElement can be `Parameter` or `DocumentElement`. The first represents generic Parameter common in different operational interfaces. The Document-Parameter represents a data container (document) which wraps structured data in a single entity, mostly wrapped by a container, e.g. XML-file, or SAP IDOC. Further sub metaclasses of TypedElement are `Field` and `ListElement`. Fields are used to construct ComplexTypes and ListElement represents the element in a Collection.
All TypedElements can be semantically annotated, which is realized by a super metaclass `AnnotatableElement`.

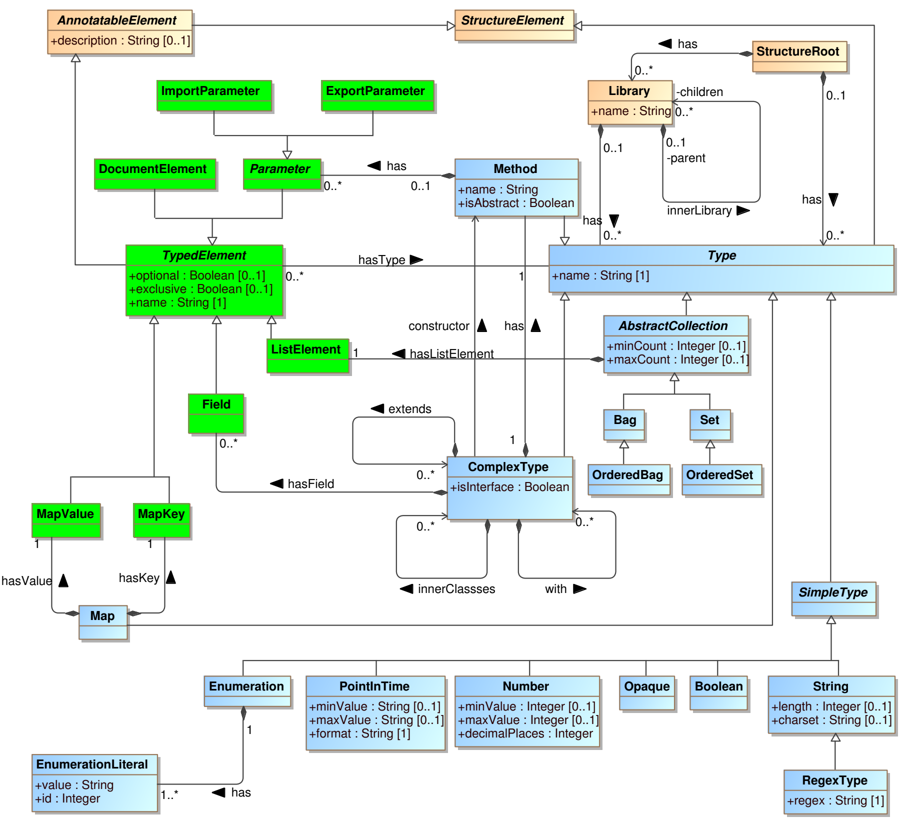


Figure 31: PIMM Structure Package

4.3 PIMM Communication Package

The communication package, as part of Figure 30, contains all communication-related interface properties at the platform independent level.

Every interface has its own interaction rules. These can be described by different communication patterns such as `MessageExchangePatterns`, different `Call`- or `Event` types. All platform specific details are wrapped by an Application Endpoint. In contrast to that, platform independent communication details are essential to interact with every interface in the correct, conflict-free way. During communication conflict analysis incompatible call mechanisms are detected and fixed (as far as possible automatically).

5 Semantic Metamodel

In this section we describe the semantic metamodel (SMM) that is used to express ontologies containing domain knowledge of integration projects. The knowledge is used to declare the meaning of model elements on all abstraction levels. We first present abstract syntax and constraints and then give an example that shows the concrete graphical syntax of the metamodel.

5.1 Abstract Syntax

The semantic metamodel (SMM) defines an abstract DSL syntax for ontology definition. The work with our industrial partners showed that controlled vocabularies, taxonomies or lightweight ontologies are sufficient to semantically describe their models, rather than using full features of ontology languages. Common ontology editors (e.g., Protege) have low industry penetration because of complexity. In many companies ontologies are not used at all. For those reasons we did not directly use existing RDF-S or OWL metamodels to represent ontologies, but rather developed an additional metamodel and offered an Eclipse GMF-based editor as concrete graphical syntax.
Figure 32 shows the metaclasses of the SMM. The `Ontology` contains `Domain` elements that allow containment and grouping of concepts. Semantic concepts can be `DomainObject` (knowledge representation of data), and `DomainFunction` (representation of functionality). Furthermore a domain can contain `DomainValue` elements (representation of knowledge about concrete values). Domain values are instances of domain objects, e. g., a domain object *Country* with its domain values *Germany*, *USA*. Concepts are associated using `Predicate` elements. Domain objects and functions are either in the role of a subject or an object. With this construct it is possible to build semantic statements (RDF-like triples) consisting of subject, predicate and object (e. g., *CustomerName IsA Name*). The metamodel offers predefined predicates: generalization (`IsA`), data processing for functions (`Input`, `Output`), containment (`Has`), data sets (`ListOf`) and equivalence (`IsEquivalentTo`). With `CustomPredicate` it is possible to model user-defined predicates. Ontologies modeled with the SMM may be evaluated to extend the metamodel by often used custom predicates.

5.2 Semantic Metamodel Constraints

Several constraints on the Semantic Metamodel have been defined and implemented in OCL that are described in the following.

- Domain names in ontologies must be unique:

  ```
  context Ontology inv: self.domains->isUnique(name)
  ```

- Semantic concepts in a domain must have unique names.

  ```
  context Domain inv: self.concept->isUnique(name)
  ```

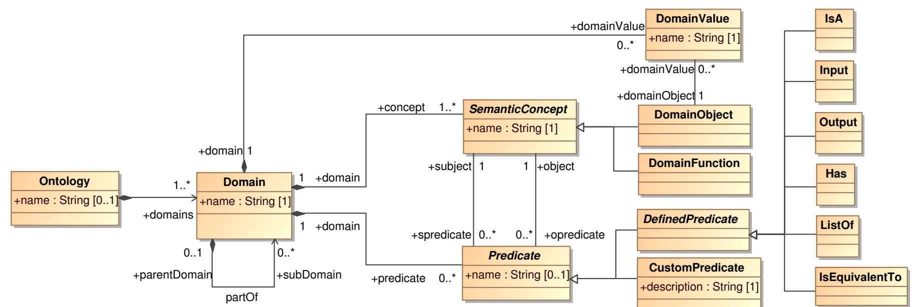


Figure 32: Semantic Metamodel (SMM)

- A semantic concept cannot link to itself with a predicate.

```
context SemanticConcept inv:
  self.spredicate->notEmpty() implies self.spredicate.object->
  notEmpty() implies self.spredicate.object->excludes(self)
```

- Predicates of domain objects that are in a role of a subject must link to domain objects.

```
context DomainObject inv:
  self.spredicate->forAll(sp | sp.object.oclIsTypeOf(DomainObject))
```

- Input and Output predicates of domain functions must link to domain objects.

```
context DomainFunction inv:
  self.spredicate->select(oclIsTypeOf(Input))->forAll(sp |
  sp.object.oclIsTypeOf(DomainObject))

  self.spredicate->select(oclIsTypeOf(Output))->forAll(sp |
  sp.object.oclIsTypeOf(DomainObject))
```

- Has-, IsA- and IsEquivalentTo-predicates of domain functions must link to domain functions.

```
context DomainFunction inv:
  self.spredicate->select(oclIsTypeOf(Has))->forAll(sp |
  sp.object.oclIsTypeOf(DomainFunction))

  self.spredicate->select(oclIsTypeOf(IsA))->forAll(sp |
  sp.object.oclIsTypeOf(DomainFunction))

  self.spredicate->select(oclIsTypeOf(IsEquivalentTo))->
  forAll(sp | sp.object.oclIsTypeOf(DomainFunction))
```

• Domain functions cannot be linked with ListOf predicates.

```
context DomainFunction inv:
  self.spredicate->select(oclIsTypeOf(ListOf))->isEmpty()
  self.opredicate->select(oclIsTypeOf(ListOf))->isEmpty()
```

5.3 Ontology Example

In Figure 33 we give an example instance of the semantic metamodel using our graphical model editor to demonstrate the aforementioned features. The small ontology is used to semantically describe simple integration scenarios and systems that relate to the domain order processing. Domain objects of the ontology are depicted as blue ellipses, like the central element *Order*. Predicates are shown as arrows labeled with type of predicate. For example the *Order* consists (`has`-predicate) of a buying *Customer* and an *ItemList*. The *ItemList* is defined as a list of *Items*. The relation between the synonyms *Client* and *Customer* is modeled with the `IsEquivalent`-predicate. Generalization (`IsA`) is expressed between *ItemPrice* and its specializations *ItemNetPrice* and *ItemGrossPrice*. Functional knowledge is modeled with the domain function *GrossPriceCalculation* (red ellipse) that takes *ItemNetPrice* and *TaxRate* as input and delivers an *ItemGrossPrice*. The correlation between *TaxRate* and *Country* is modeled with a custom predicate `dependsOn`. The ontology also provides two domain values *Germany* and *USA* that are instances of *Country*.

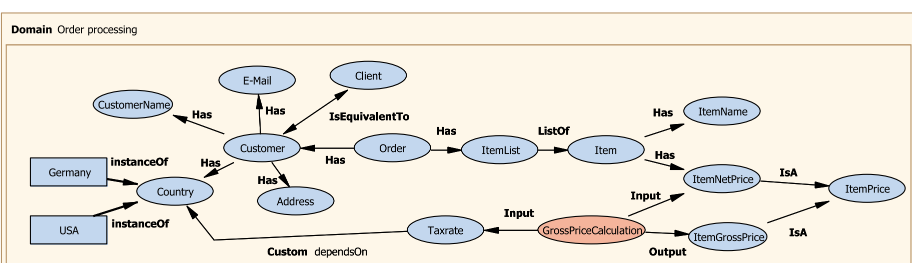


Figure 33: Example ontology

6 Annotation Metamodel

Semantic annotation is the term originating in the Semantic Web community that describes associations of ontology elements to service interfaces, documents and web resources [12, 19, 7]. It enables machine-based processing of these artifacts. We adopted this paradigm to semantically annotate elements of integration scenario models at the CIM level and interface description models at the PSM and PIM level enabling automatic annotation analysis for element matching. The intermediate annotation metamodel (AMM) was developed that links the SMM with respective metamodels of our framework, using the model weaving approach [9].

6.1 Core Annotation Metamodel

The AMM includes a `AnnotationModelRoot` that contains a least one ***`AnnotationElement`*** (Figure 34). It has four specializations: `DomainObjectAnnotation` (data-oriented annotation), `DomainFunctionAnnotation` (function-oriented annotation), `DomainValueAnnotation` (instance annotation) and ***`LogicalOperator`*** (annotation grouping). Data-oriented annotations are used to describe all model elements that represent data at a certain level of abstraction. Functional annotations describe the meaning of computational and processing behavior. Value annotations describe concrete instances of data representation. Logical operators (`AND`, `OR`, `XOR`) link two or more annotations for combination. Each of the annotation types links to at least one respective knowledge definition metaclass of the semantic metamodel (`DomainObject, -Function and -Value`). Besides the core metamodel classes the most important part of the AMM are the associations to our other metamodels, that are described in the following.

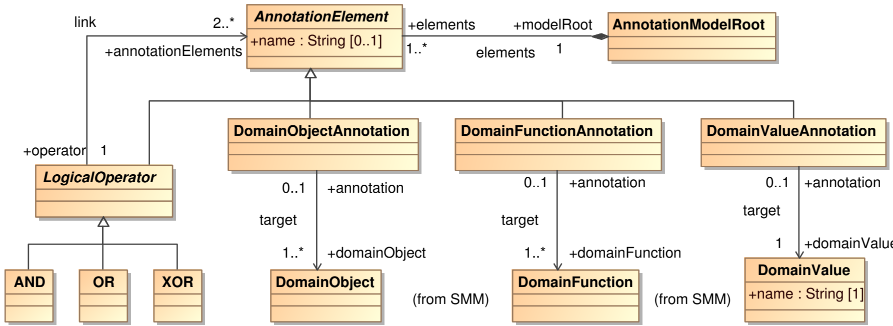


Figure 34: Core elements of the annotation metamodel

6.2 Data Annotation

At the most abstract modeling level (CIM), data objects and their structure are defined using the metaclass `BusinessObject` of the CIMM. The annotation metaclass of the AMM links it to at least one `DomainObject` of the SMM (Figure 35).

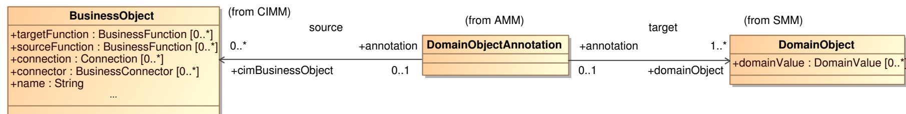


Figure 35: Semantic data annotation at the CIM level

At the PSM level, data annotations refer to interface model elements that have data semantics on a certain level of granularity. Each of the platform-specific metamodels includes the abstract metaclass *`AnnotatableElement`* that is linked with the `DomainObjectAnnotation`. Each model element that shall be annotated inherits from *`AnnotatableElement`*. Figure 36 shows the possible annotations for two of the supported platforms: EJB components running in J2EE containers and relational database management systems (RDBMS). Enterprise JavaBeans offer methods, that have `Input` and `Output` parameters which can be annotated. The parameters have atomic simple types (such as floats and integers) or complex types composed of annotatable `Field` elements. The metamodel for RDBMS offers queries that have a set of `Parameters`, that are either `Input` or `Output`. They represent columns of SQL result sets or parameters of insert and update statements which are important for annotation.

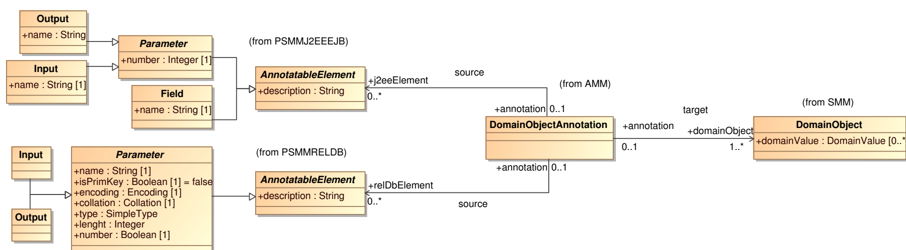


Figure 36: Semantic data annotation at the PSM level – J2EE systems and RDBMS

The platform specific metamodel for the SAP R/3 systems allows the modeling of several data-oriented aspects of the SAP R/3 interfaces. An application specialist can specify Business API functions (BAPI), that contain input and output `Parameter` elements which are typed (`SAPType`) according to their content (tables, structures and simple types). SAP R/3 systems also offer document exchange interfaces (`IDOCType`) with further sub-structure (`Segment` and typed

Field elements). The link to the annotation allows to declare the meaning of the interface structure at respective levels of refinement (Figure 37).

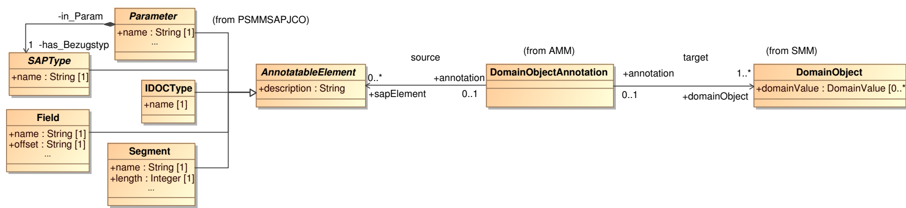


Figure 37: Semantic data annotation at the PSM level – SAP R/3 systems

In Figure 38 we present the data annotations for Web Service descriptions and XML schema (XSD) definitions. The WS metamodel provides WSDL service descriptions with ports and operations, that have annotatable Parameters. Messages are exchanged through parameters that consist of MessageParts which can also be annotated. The structure of a message part is modeled with XML schema Elements of the XSDlite metamodel. Furthermore Attributes of complex XSD types are annotated.

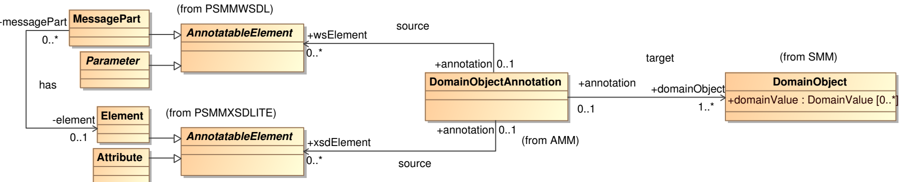


Figure 38: Semantic data annotation at the PSM level – Web Services and XML schema

The annotations illustrated previously enable semantic description of data oriented interface elements at the PSM level. To carry out the conflict analysis, a model-to-model transformation of the PSM models to PIM models is performed (see Section 9). Semantic descriptions given at the PSM level are thus transformed to the PIM level as well. Like on the PSM level, all data elements that can be annotated are specializations of the *AnnotatableElement* metaclass of the PIMM (Figure 39). All TypedElements, such as Parameters and Fields of complex types, and document elements are therefore linked to the annotation.

6.3 Functional Annotation

Functional annotations describe the meaning of computational and processing behavior of model elements such as remote functions or Web service operations. The AMM provides associations between a functional element at the CIM, PSM or PIM level and the domain function from a given ontology using the DomainFunctionAnnotation element.

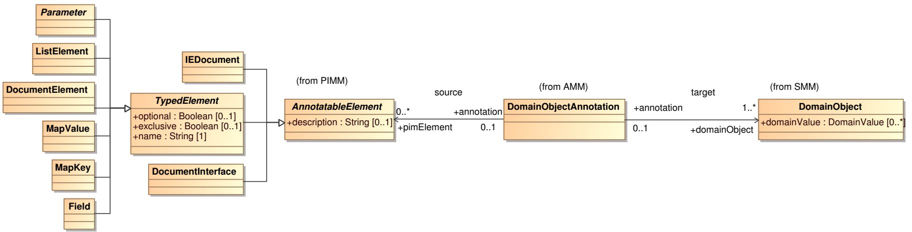


Figure 39: Semantic data annotation at the PIM level

At the CIM level, `BusinessFunction` and `ConnectorFunction` convey the processing semantics of the integratable systems and connectors respectively (Figure 40). Thus they are annotated with domain function annotations.

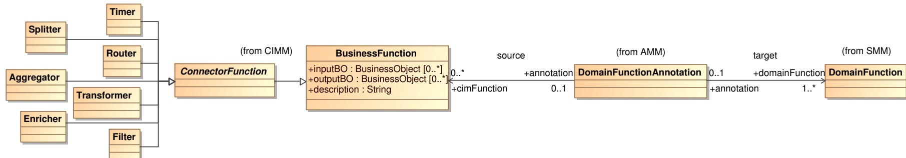


Figure 40: Semantic functional annotation at the CIM level

Figure 41 shows functional annotations for the J2EE and SAP R/3 platform specific metamodel elements. J2EE systems can be annotated by linking `EnterpriseJavaBeans`-elements (coarse granularity) and `Method`-elements of bean interfaces (fine granularity) to the ontology. SAP R/3 functionality is represented with the by annotatable elements `SAP_R3_Interface` and `Method`.

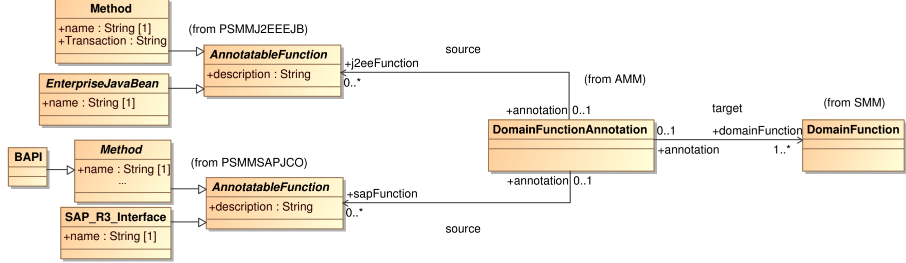


Figure 41: Semantic functional annotation at the PSM level – J2EE and SAP R/3 systems

In the metamodel for interfaces of relational database management systems the following metaclasses have the processing semantics, and are thus functionally annotated: `Query`, `StoredProcedure`, `SQLQuery` and its sub-classes. The PSMM for Web services provides `Operations` that can be annotated (Figure 42).

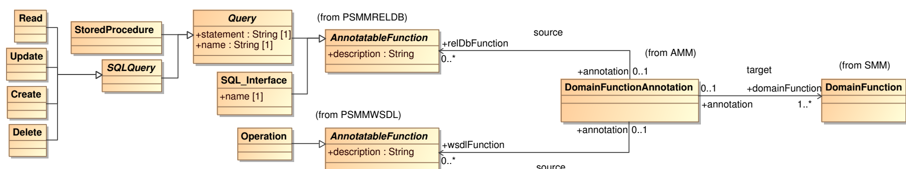


Figure 42: Semantic functional annotation at the PSM level – RDBMS and Web Services

Finally, functional PSM elements are transformed to either `IEModule` (coarse granularity) or `FunctionInterface` (fine granularity) at the PIM level. Both can be annotated with domain functions from the ontology.

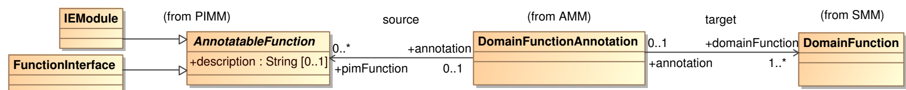


Figure 43: Semantic functional annotation at the PIM level

6.4 Value Annotation

Value annotation is used to semantically describe instance information modeled at the PSM level (see `ValueRange` description in section 3 on page 16). Figure 44 exemplarily shows the annotation of value range's `Entry` of J2EE systems and the value annotation at the PIM level.

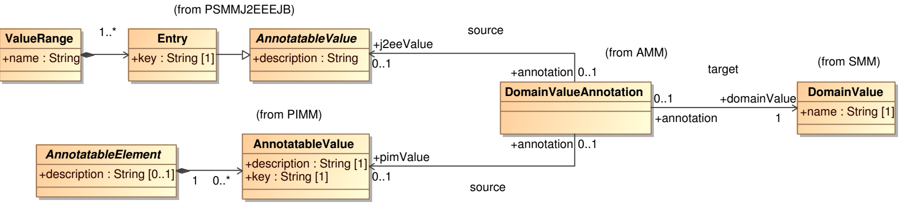


Figure 44: Value annotation at the PSM level (J2EE) and at the PIM level

6.5 Annotation Metamodel Constraints

To ensure valid usage of the Annotation Metamodel several OCL constraints have been added to metamodel.

- An annotation must link at least to one element at either CIM, PSM or PIM level.

```
context DomainObjectAnnotation inv:
  self.cimBusinessObject->notEmpty() or self.pimElement->notEmpty() or
  self.xsdElement->notEmpty() or self.wsElement->notEmpty() or
  self.relDbElement->notEmpty() or self.sapElement->notEmpty() or
  self.j2eeElement->notEmpty()

context DomainFunctionAnnotation inv:
  self.cimFunction->notEmpty() or self.pimFunction->notEmpty() or
  self.relDbFunction->notEmpty() or self.sapFunction->notEmpty() or
  self.wsdlFunction->notEmpty() or self.j2eeFunction->notEmpty()

context DomainValueAnnotation inv:
  self.j2eeValue->notEmpty() or self.relDbValue->notEmpty() or
  self.sapValue->notEmpty() or self.pimValue->notEmpty()
```

- An annotation cannot link elements of different PSM types at the same time. The following constraints implement XOR functionality. It was not possible to use the OCL XOR operator, because for more than two parameters it realizes *at least* semantics.

```
context DomainObjectAnnotation inv:
(
  self.j2eeElement->notEmpty() and not self.relDbElement->notEmpty() and
  not self.sapElement->notEmpty() and not self.wsElement->notEmpty() and
  not self.xsdElement->notEmpty()
) or (
  not self.j2eeElement->notEmpty() and self.relDbElement->notEmpty() and
  not self.sapElement->notEmpty() and not self.wsElement->notEmpty() and
  not self.xsdElement->notEmpty()
) or (
  not self.j2eeElement->notEmpty() and not self.relDbElement->notEmpty() and
  self.sapElement->notEmpty() and not self.wsElement->notEmpty() and
  not self.xsdElement->notEmpty()
) or (
  not self.j2eeElement->notEmpty() and not self.relDbElement->notEmpty() and
  not self.sapElement->notEmpty() and self.wsElement->notEmpty() and
  not self.xsdElement->notEmpty()
) or (
  not self.j2eeElement->notEmpty() and not self.relDbElement->notEmpty() and
  not self.sapElement->notEmpty() and not self.wsElement->notEmpty() and
  self.xsdElement->notEmpty()
) or (
  self.j2eeElement->isEmpty() and self.relDbElement->isEmpty() and
  self.sapElement->isEmpty() and self.wsElement->isEmpty() and
  self.xsdElement->isEmpty()
)

context DomainFunctionAnnotation inv:
```

```
(
  self.j2eeFunction->notEmpty() and not self.relDbFunction->notEmpty() and
  not self.sapFunction->notEmpty() and not self.wsdlFunction->notEmpty()
) or (
  not self.j2eeFunction->notEmpty() and self.relDbFunction->notEmpty() and
  not self.sapFunction->notEmpty() and not self.wsdlFunction->notEmpty()
) or (
  not self.j2eeFunction->notEmpty() and not self.relDbFunction->notEmpty() and
  self.sapFunction->notEmpty() and not self.wsdlFunction->notEmpty()
) or (
  not self.j2eeFunction->notEmpty() and not self.relDbFunction->notEmpty() and
  not self.sapFunction->notEmpty() and self.wsdlFunction->notEmpty()
) or (
  self.j2eeFunction->isEmpty() and self.relDbFunction->isEmpty() and
  self.sapFunction->isEmpty() and self.wsdlFunction->isEmpty()
)

context DomainValueAnnotation inv:
(
  self.j2eeValue->notEmpty() and not self.relDbValue->notEmpty() and
  not self.sapValue->notEmpty()
) or (
  not self.j2eeValue->notEmpty() and self.relDbValue->notEmpty() and
  not self.sapValue->notEmpty()
) or (
  not self.j2eeValue->notEmpty() and not self.relDbValue->notEmpty() and
  self.sapValue->notEmpty()
) or (
  self.j2eeValue->isEmpty() and self.relDbValue->isEmpty() and
  self.sapValue->isEmpty()
)
```

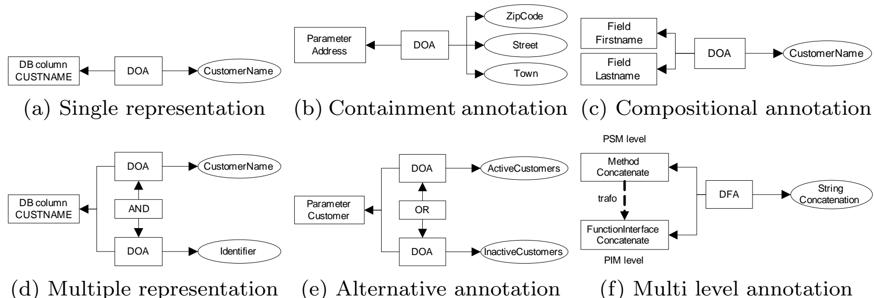
- Logical operators cannot group annotations of different type, e. g., it is not allowed to combine a domain object annotation with a domain function annotation.

```
context LogicalOperator inv:
(
  (self.annotationElements->select(e |
    e.oclIsTypeOf(DomainObjectAnnotation))->notEmpty()) and
  not (self.annotationElements->select(e |
    e.oclIsTypeOf(DomainFunctionAnnotation))->notEmpty()) and
  not (self.annotationElements->select(e |
    e.oclIsTypeOf(DomainValueAnnotation))->notEmpty())
) or (
  not (self.annotationElements->select(e |
    e.oclIsTypeOf(DomainObjectAnnotation))->notEmpty()) and
  (self.annotationElements->select(e |
```

```
    e.oclIsTypeOf(DomainFunctionAnnotation))->notEmpty()) and
  not (self.annotationElements->select(e |
    e.oclIsTypeOf(DomainValueAnnotation))->notEmpty())
) or (
  not (self.annotationElements->select(e |
    e.oclIsTypeOf(DomainObjectAnnotation))->notEmpty()) and
  not (self.annotationElements->select(e |
    e.oclIsTypeOf(DomainFunctionAnnotation))->notEmpty()) and
  (self.annotationElements->select(e |
    e.oclIsTypeOf(DomainValueAnnotation))->notEmpty())
)
```

6.6 Annotation Examples

The annotation metamodel allows to build various combinations of semantic descriptions. We support five different annotation granularity levels (Figure 45a-45e) that are analyzed by the semantic conflict analysis [5] and multi level annotation across CIM,PSM and PIM level of abstraction (Figure 45f). **Single representation annotation** (a) links one model element to one ontology concept (e. g., *DB column CUSTNAME* → *CustomerName*). **Containment annotation** (b) links one coarse grained model element to multiple concepts (e. g., *Parameter Address* → {*Street,Town,ZipCode*}). **Compositional annotation** (c) links multiple model elements to one coarse grained concept (e. g., {*Field Firstname,Field Lastname*} → *CustomerName*). **Multiple and alternative representation annotation** (d,e) combine annotations with the `AND` and `OR/XOR` operators respectively (e. g., *DB column CUSTNAME* →*(Identifier && CustomerName)*). Finally, Figure 45f shows simultaneous annotation of model elements on different levels of abstraction. This annotation type is used to trace model transformation while abstracting from PSM to PIM level for conflict analysis. Furthermore it can connect abstract business model elements at the CIM level to interface realizations at the PSM and PIM level.



(a) Single representation (b) Containment annotation (c) Compositional annotation

(d) Multiple representation (e) Alternative annotation (f) Multi level annotation

Figure 45: Annotation types and examples

Semantic annotation is implemented as Eclipse View in our Model-Based Integration Framework. At the left side of the annotation editor (Figure 46) the user can load ontologies modeled with the graphical or tree-based semantic model

editor. At the right side CIM and PSM models are loaded to list annotatable elements. In-between, annotations are created by using *Add Annotation* with selected model elements at the left and right. A table of existing annotations is shown in the middle. Annotation models are maintained in the project's workspace and can be loaded at any time for additions.

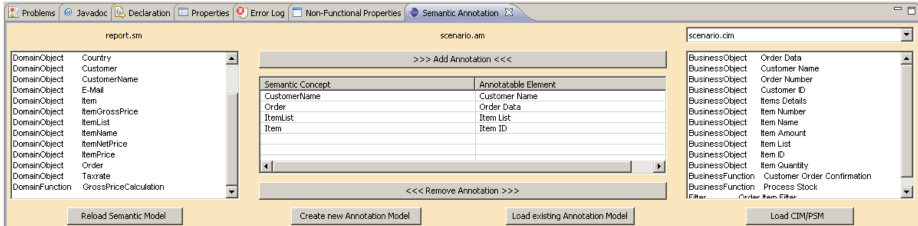


Figure 46: Semantic Annotations Editor

7 Property Metamodel

In order to support the analysis of non-functional properties (NFP) in integration scenarios, the Property metamodel has been created to facilitate specification of relevant properties. NFP such as availability, security, timeliness or cost often play the crucial role in software integration when it comes to satisfying business process requirements. Their analysis is either neglected or informal, following best practices. Sometimes this is not enough as non-functional incompatibilities may compromise not only the quality of integration solution, but also limit its functionality. Therefore, we first discuss and describe the (incomplete) taxonomy of NFP that are relevant for software and data integration, and then propose a metamodel that enables their dynamic definition.

7.1 NFP Taxonomy

The taxonomy is not intended to be complete (next section explains how it can be expanded), but will be used here as a starting point to discuss modeling of non-functional aspects of software systems, interfaces, parameters and connectors, evaluate integration solutions, compute overall properties, detect non-functional mismatches and rank integration alternatives. The first level of taxonomy entries are property categories which thematically group NFP defined at the second level. The third level (not shown in the picture) characterizes each property by description, type (e. g., float values), scope and the default unit (e. g., milliseconds for latency or currency codes for cost per invocation). Units are described either using standards such as ISO 80000-3:2006 (time), ISO 80000-13:2008 (IT units), ISO 8601 (date) or ISO 4217 (currency code), or as enumerations (e.g., secure transport protocols or data encryption algorithms). In the following we provide more details on each property category.

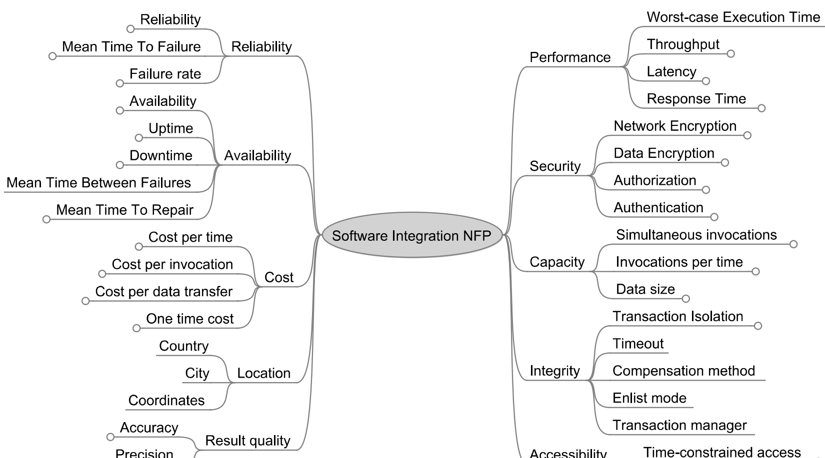


Figure 47: Taxonomy of software integration non-functional properties (excerpt)

Reliability incorporates properties related to the ability of a system or component to perform its required functions under stated conditions for a specified period of time. Reliability is the probability that the system continues to function until some time t. Usually, steady-state reliability is used to characterize integrated systems, where observed interval is the system's lifetime. Reliability attributes such as failure-rate (λ) or mean time to system failure (MTTF) can also be specified. The taxonomy allows to relate attributes and properties using expressions, for example $\lambda = \frac{1}{MTTF}$ (this relation is not shown in Figure 47).
Availability is similar to reliability, with one important exception: it allows for a system to fail and to be repaired. Thus, additional properties appear in this category: mean time to repair (MTTR), mean time between failures (MTBF) as well as average uptime/downtime per year. Again, relations between properties are defined (e.g., $A = \frac{MTTF}{MTTF+MTTR}$). Using this category, fault-tolerant lifecycle of integrated systems and their components can be analyzed.
Cost related NFP are used to compare different software integration alternatives. For example, comparison of *cost per data transfer*-properties is achieved by specifying their unit in terms of a currency code in relation with the base unit byte, combined with a metric or binary prefix. The value type is float-based. Values with different currencies and different amounts of data volumes may be compared by determining the current exchange rate and by converting the prefix, but external source of currency conversion is then required (rate table).
Performance category includes properties that are either bandwidth or timing related. They enable detection of possible integration bottlenecks and allow investigation of connector features such as caching as well as timing constraints. For example, *worst-case execution time (WCET)* is the maximum amount of time that elapses between invocation of a system or interface and its reaction, specified using integer as value type and second as unit.
Security category describes encryption and access control related properties. We will exemplary describe the *network encryption* property, consisting of three sub properties *strength*, *protocol* and *technology*. Encryption strength is the key size expressed by integer values with the default unit bit. It is more difficult to compare protocols or technologies, because they cannot be expressed by numeric values. The property therefore makes use of an enumeration including possible protocols, together with a key. If the protocol property it is assigned to a system, a set of supported protocols is selected and can be compared to other sets. Additionally, rating for each protocol can be added (e. g., based on encryption algorithm and module validation) to facilitate comparison.
Capacity properties are considered to avoid system overloads during integration. For example, *data size* property specifies the maximum amount of data that can be passed to a system at once (value type: integer, default unit: megabyte). Together with *throughput*, duration of integration runs can be thus analyzed.
Integrity describes transactional behavior. *Isolation level* supports read uncommitted, read committed, repeatable read and serializable levels. *Timeout* defines system or interface timeout which can be used to discover timing mismatches, e.g, if a service with 1 hour timeout is waiting for a service with 1 day WCET, there is a timing conflict. *Enlist* specifies transactional modes (support, require

or join). Thus, incompatibilities can be discovered where one system requires transactions but its integration partners do not support them.
Location category describes geographical system/service location, as it may be necessary to determine validity of an integration scenario. For example, confidential data storage may be restricted to particular countries, because of legal considerations. To enable this we include country name and its ISO code, as well as city and GPS coordinates of the system location.
Result quality describes attributes of the data/messages produced by a system. *Accuracy* represents calculation correctness (e.g., number of decimal places or maximal guaranteed computation error), while *precision* is the measure of system output quality which may be gathered and evaluated statistically over a period of time, potentially also by users (in form of a reputation scale).
Accessibility is expressed by *time-constrained access*: it represents concrete time intervals or periodic time slots in which a system is accessible for integration tasks (e. g., Extract-Transform-Load tasks done on Sundays 0:00-5:00 a.m.).

7.2 Non-functional Property Metamodel

The taxonomy presented in the previous section obviously cannot aim to be complete. It represents an excerpt of properties we found to be relevant in the context of software integration. In order to address dynamic generation of additional properties, we propose a property metamodel (PMM) for expressing user-defined NFP and assigning them to other model elements. The abstract syntax of the PMM is used to build non-functional property taxonomies, like the one in the previous section, enabling their usage in model-driven development environments. In the following we make use of the `typewriter` font to refer to metamodel classes and attributes, and describe examples on instance level with *italic* font. In Figure 48 the basic structural features of the metamodel are shown. Properties, units and unit multiples are modeled separately and are grouped into categories. The categories and their elements together form the `PropertyModel`. We also define the `NamedElement` metaclass (not shown in the picture) with the attributes `+shortName:String[1]` and `+longName:String[1]` which are passed on to the following metaclasses: `PropertyModel`, `Category`, `Property`, `SimpleUnit`, `UnitMultiple`, `EnumerationUnit` and `EnumerationLiteral`. The short name is used to express abbreviations, the long name is the full name of an element instance (e. g., for a `Property`: *MTTF*, *Mean Time To Failure*).
Figure 49 depicts details of the `Property` metaclass. The `scope`-attribute defines NFP validity. The `PropertyScope` can either be `System`-wide (e. g., *mean time to repair* of a database system), `Interface`-wide (e. g., *availability* of a Web service), `Function`-related (e. g., *cost per invocation* of a Web service operation) or `Parameter`-related (e. g., *accuracy* of a J2EE component method's return value). The `description`-attribute includes additional explanatory text. The attribute `valueType` constraints the value kind of a property to support comparison: character-based values (`String`), floating point numbers (`Double`), integer-based representations (`Long`) and `Boolean`-values. Properties can be nested with the `partOf`-relation. A property is associated

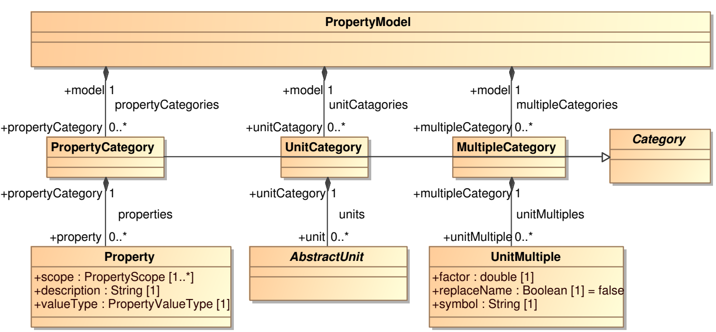


Figure 48: PMM - Structural modeling features

with a default unit, which is the standard unit for its use and comparison (e. g., *Mbit/s* for *throughput*-property). Dependencies between properties can be declared with `PropertyCorrelation`. It is evaluated at runtime and can contain mathematical or boolean `expression`s. For instance, the calculation rule for availability $A = \frac{MTTF}{MTTF+MTTR}$ can be expressed with MathML Content Markup using property names as variables.

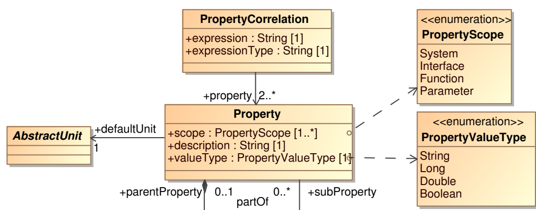


Figure 49: PMM - Non-functional properties and their units

Figure 50 shows four different types of units which are defined to build NFP-unit definitions. The `SimpleUnit` is used to create basic SI- or ISO 80000 units such as *second* or *bit*. Besides the `shortName` and `longName` inherited from `NamedElement`, the `symbol`-attribute is used for symbolic representation of a unit, if available. `DerivedUnit`s are created from simple ones by combining them with `UnitMultiple`. A multiple is either a prefix added in front of the unit name (e. g., metric prefix *kilo*, binary prefix *kibi*) or a non-SI multiple that replaces the whole unit name (e. g., *minute*). A multiple of a unit is represented by a `factor` and a `symbol`. The name change is controlled by the `replaceName`-attribute. Composed units are modeled with the `CompoundUnit` metaclass. It combines left- and right-handed units with an `Operator`. All four unit specializations can be combined, allowing reuse and nesting (for example, units such as kg/m^3 can be created). Finally, the `EnumerationUnit` describes possible values of a property as a set of `EnumerationLiteral`s. The `allowedSelection`-attribute declares whether a property assignment must use exactly one literal or can use

many literals. The `key`-attribute of the `EnumerationLiteral` is an additional unique numeric code. Furthermore a `rating` value can be given that assesses a literal (e. g., encryption technologies). For example, our property model includes the enumeration *ISO 4217 currency names and code elements* that is used for cost properties.

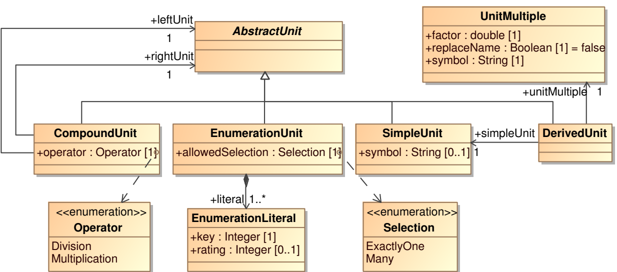


Figure 50: PMM - Simple, derived, compound and enumeration units

The `PropertyAssignment` (Figure 51) is the link between any model element that should be described by a NFP and the actual property, its value and additional attributes. The `type` of an assignment specifies whether the assigned property is offered/provided by the system or interface (e. g., *provided uptime*), or expected/required from other systems (e. g., *required network encryption*, meaning anyone interacting with the system is required to use a certain encryption strength or a secure protocol). Property value is either given as a single value (`PropertyValue`), as `PropertyValueRange` (e. g., minimum and maximum value) or if necessary as a set of single values (`PropertyValueSet`). In case that another unit should be used for assignment, default unit can be overridden by the `unitModifier`-attribute (it is constrained that the unit can only be modified along its quantity, e. g., using other multiples). If a property makes use of an enumeration unit, the assignment selects literal(s) of that enumeration (`enumLiteralSelection`). The cardinality of the attribute is restricted to 0..1 in case `allowedSelection` is set to *ExactlyOne*.

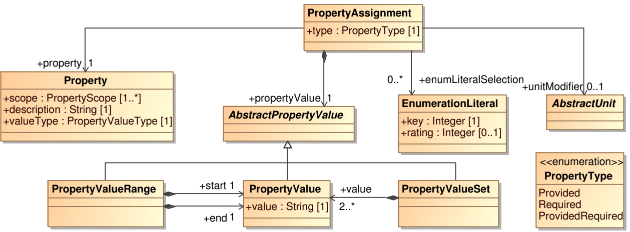


Figure 51: PMM - Property assignment

The example of using property metamodel to annotate an SAP BAPI interface (see Section 3.1 for description of SAP metamodel) is given in Figure 52. It

depicts how required and provided availability, mean time to repair, mean time to failure, and worst-case execution time can be specified.

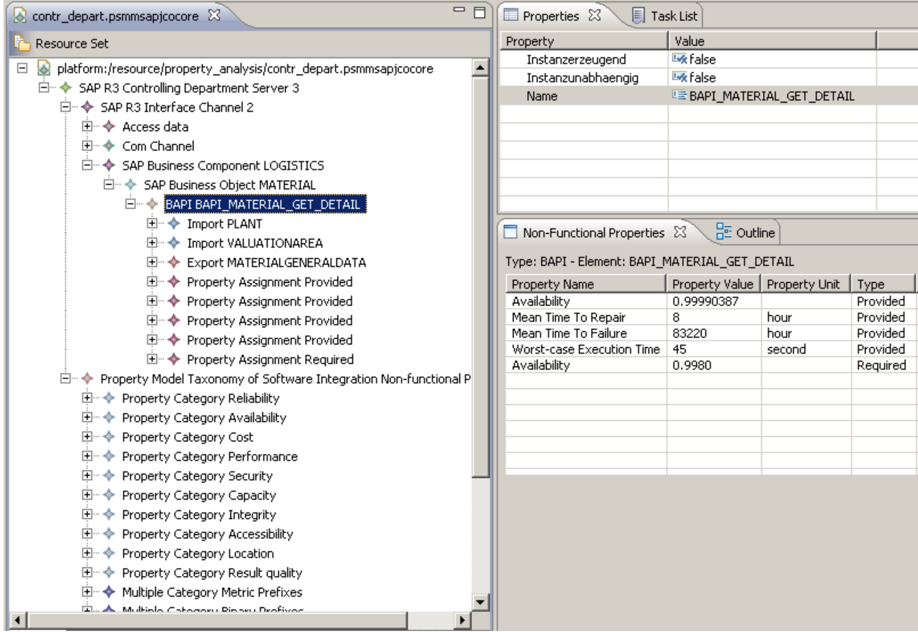


Figure 52: NFP editor example

8 Connector Metamodel

In this section the Connector metamodel will be described. Packages *structure*, *types* and *message* contain static structure description of used libraries, in particular class structure, type system and supported message types. Packages *connector* and *expression* form DSL abstract syntax. The *connector* package contains elements of message processing components (EAI patterns such as transformer, router, filter), application endpoints and message routing logic. The *expression* package contains elements for expression modeling. Message processors have references to corresponding message transformation expressions, which define their behavior.

8.1 Connector Package

The connector package contains elements required to describe message flow and processing. It specifies the upper abstraction level for connector design and references expression package for behaviour specification of message processors. Using messages and message processors to specify connector functionality seems to be the most generic and widely accepted abstraction approach. Furthermore, it enables service-oriented connector realization. There are two basic types of connector components: application endpoints and message processors. Application endpoints generate and consume messages by wrapping modelled system interfaces, and message processors manipulate (transform, route, split etc.) messages. Components send or receive messages of a specific message type via ports. Messages are transported by message channels which have a message channel type and a message exchange pattern. Excerpt from the connector package is given in Figure 53.

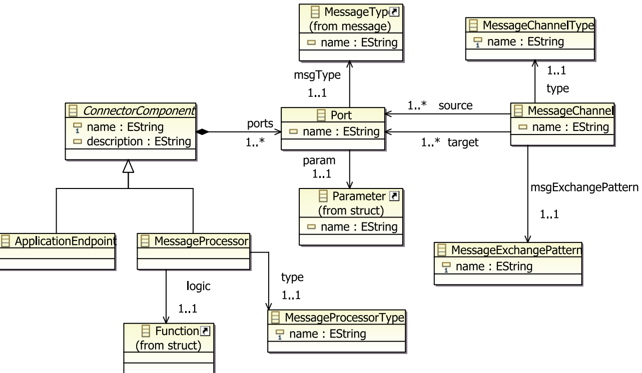


Figure 53: Package *connector* - Connector Metamodel

To avoid metamodel changes which would make refactoring of the connector design and generation framework necessary for most changes, we choose an instance-based library approach instead of using metamodel element specializations. When creating a new connector model there is an automatic reference

to the library connector model. It contains allowed message processor types, message channel types and message exchange patterns which then can be used to specify elements in the new connector model. Message types are not part of the library connector model because they are component-specific and therefore instantiated in the new model. They can be associated to component ports to specify supported data format. Ports connected via message channel must support the same message type.
The following predefined library elements are included in DSL:

- Message processor types: Aggregator (combines several messages to one single message), Content Enricher (extends message content to serve special data requirements of message receivers), Filter (passes only specific messages), Content-based Router (selects message receivers by analyzing message content), Splitter (divides one message into several messages containing specific parts of the original message), Timer (generates event messages to control other message processors or application endpoints) and Transformer (changes message format and/or message content by applying transformation rules). Behavior of all message processors is configured/programmed using elements from the *expression* package, e.g., the logic that Transformer has to execute or routing rules for Content-based Router are thus defined.

- Message channel types: Point-to-Point (transports messages from a sender to exactly one receiver) and Publish-Subscribe (transports messages from a sender to several receivers).

- Message exchange patterns: Out-Only (one-way message exchange where the receiver returns a status), Robust Out-Only (reliable one-way message exchange where the receiver returns a status, if the status is negative the sender returns a status as well), Out-In (two-way message exchange where the receiver responds with a message which is confirmed by the sender with a status) and Out-Optional-In (two-way message exchange where the receiver's response message is optional).

8.2 The Expression Metamodel

The expression package is used to model expressions which define behavior of message processors. One of the main problems when using existing workflow modeling notations is that the actual expressions are not modeled at all. They have to be input as plain text using script language such as OCL, internal proprietary language like in E2E Bridge, or even Java like in Mathilda [22] [21]. We implemented an expression metamodel that allows to model transformation expressions with arbitrary function/operation calls, support for the high-order function calls and lambda expression definitions. The expression flow metamodel is built using abstract notions, which allows us to define actual types of the activities in runtime.

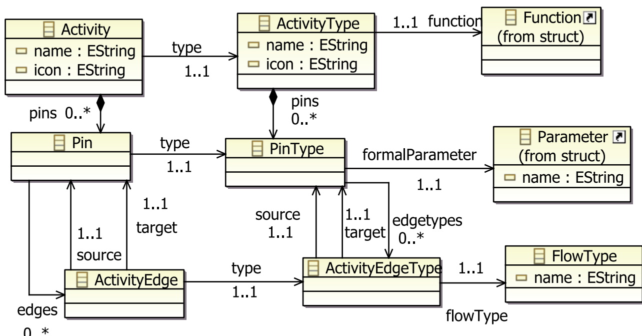


Figure 54: Package *expression* - Expression Metamodel

We now examine the main concepts which are introduced in the expression metamodel (excerpt is shown in Figure 54). The *ActivityType* represents functions or operations. *PinType* abstracts the function or operation formal parameter. Collection of *ActivityEdgeTypes* shows which pin types are allowed to be connected with each other. The metamodel also supports different types of connections, which are represented by the *FlowType* metaclass. In other words combinations of *ActivityType*, *PinType* and *ActivityEdgeType* elements form the library of available actions and allowed connections of specified flow types between them. Actions can represent function or operation calls and connections between them show possible actions, which can produce/consume values that can be used for fulfilling formal parameters. The allowed connections can be determined from the type information stored in the *structure* package.
Activity metaclass represents the actual function or operation call, determined by the corresponding *ActivityType*. The *Pin* metaclass shows a formal parameter value that is passed to a function call. The value is determined using *ActivityEdge* metaclass, which shows directed flow from one *Pin* to another. The expression metamodel allows definition of domain specific semantics and custom operations and functions.

8.3 The Structure packages: *structure* and *message*

The *structure* and *message* packages contain elements that describe static structure of available libraries (Figure 55). They allow storing information about library contents, such as classes, functions and properties. The *message* package allows to model typed messages that can be received and processed by application endpoints and message processors. A message contains header and body parts. Each part has a name and type from the *structure* package. Any class or basic type can be used as a message part type.

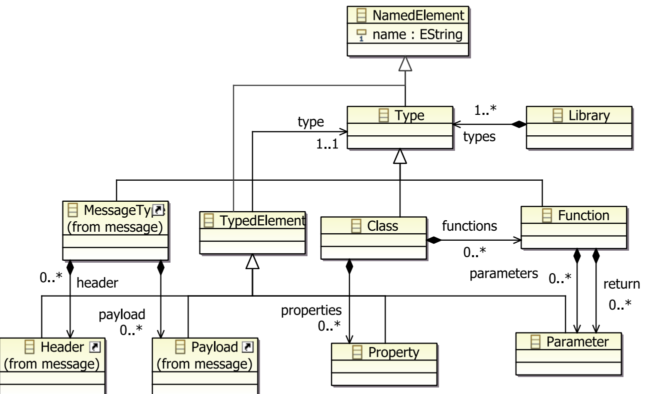


Figure 55: Packages *structure* and *message* - Structure and Messages Metamodel

8.4 Concrete Syntax

In this section, the concrete syntax and semantics of the proposed Connector Metamodel is described. We first discuss graphical notation for instantiating *connector* package, followed by graphical and textual syntax for the *expression* package modeling.

The connector package represents the upper abstraction level where message flow is used to model an integration process. Application endpoints and message processors are parts of the flow and can generate/consume/transform event and data messages. Figure 56 shows an instance of the *connector* package, realized using our graphical syntax implemented with Eclipse GMF. Message types are specified as properties and are not visualized in the process flow. They are assigned to ports of application endpoints and message processors to specify data schema requirements of connector component interfaces. Existing message processor types specified in the component library instance are assigned to message processors. In case that required pattern is not available, it is possible to expand the library instance. Existing patterns are then configured and new patterns are specified by lower level behavior description using the *expression* package.

Supported activity types of the *expression* package (graphical and textual syntax) are given in Table 9. The main concept is function/operation call. In many languages (e.g. C++, C#, Scala) the operator concept is implemented using the function call technique. We also use one element for both concepts. Formal parameters of an activity are represented with the Pin element, as it is done in UML. The flow of values from pin to pin is expressed using activity edge element which describes directed connection between two pins and also specifies connection type. Default flow type for passing values in function calls is message flow. Intermediate calculation results are stored using the variable element. It is also used to express formal parameters. The presented DSL is a functional language, so it enables system integrators to create Lambda-expressions and high-order

Activity Type Name	Graphical Notation	Textual Concrete Syntax
Function/Operation Call	Operation/Function Call	getOrders()
Property Accessors	Property Accessor	getOrders().first().name
Variable/Parameter	Parameter	Order order=getOrders().first();
Lambda Expression		o|o.name.startsWith(”Michael”)
Pin		NA
High-Order Function Call	High-Order Function Call	getOrders().select(o|o.name.startsWith(”Michael”))
For/Iterator	for	List<Item> l = for(o : orders) returning new Item(o);
Conditional Operator	- +	if(expression) { doSmth();} else { doSmthElse();}
Message Flow		NA
Conditional Flow		NA
Collection Flow		getOrders()->items.sum();

Table 9: Activity Types in the DSL Graphical Concrete Syntax

functions. Support of functional concepts allows to offer rich set of collection processing possibilities, such as filtering, sorting or mapping. These functions are provided by the standard library. As additional possibility we introduce collection flow that acts as 'for' operator. The collection flow receives a collection, iterates through it and sends each collection element iteratively or in parallel to a destination pin. The strong support for collection processing was required by the nature of system integration scenarios. The conditional activity allows to implement branching of message flow. Based on the boolean value received using conditional flow, conditional activity selects one branch to which it passes its message.

8.5 Example Scenario

We now introduce a example, of using connector metamodel and concrete syntaxes for defining integration scenarios. It comes from the publishing domain and requires integration of three systems: web shop, ERP system and credit card payment system. The web shop collects orders and credit card information from customers but the system is not able to charge any credit card account. That transaction functionality is provided by the payment system. All successful orders have to be available in the ERP system to create monthly performance reports and to prepare tax documents. A connector between the three systems has to generate a payment transaction for every customer order coming from the web shop and send it to the payment system. Already finished transactions must be identified and the order status in the web shop automatically changed

according to the corresponding transaction status in the payment system. Processed orders are transferred to the ERP system afterwards. Because all three systems rely on different back end technologies (MySQL, MSSQL and SAP R/3) the connector has to overcome technical interface heterogeneity and at the same time it must deal with three different data models.
The connector model (instance of the *connector* package) is given in Figure 56. The message flow starts with an event message, generated every 10 minutes by the Timer component. The event message is transferred via the Publish-Subscribe message channel to the Web shop and Payment system Application Endpoint. Both endpoints adapt (wrap) native system interfaces. After receiving the event both application endpoints send requested data messages to the aggregator which generates one message out of them. This message contains Web shop orders and payment transactions and is delivered to the message transformer. The transformer rearranges message content and generates tuples, each containing one Web shop order and the corresponding payment transaction (if one exists). The message splitter then divides the message into smaller messages containing only one tuple and sends them to the content based router and the OrderToIdoc Transformer which prepares the message for the ERP Application Endpoint. If the message contains an order with corresponding transaction, it is routed to the OnlyOrderIDAndStatus transformer which generates a message containing only order ID and order status. The status is set to paid and the message is sent to the Web shop Application Endpoint. If the message contains only an order and no transaction, it is routed to the OrdersToTransaction transformer which generates a payment transaction message from it.

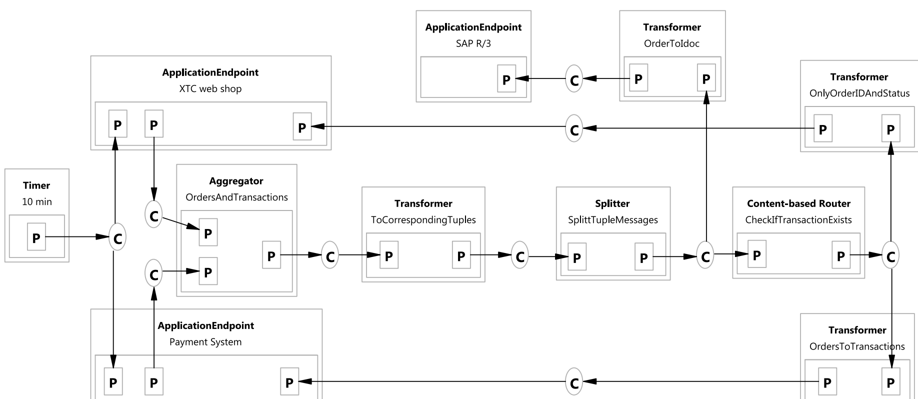


Figure 56: Example component package instance

The code given in Figure 57 shows exemplary expression specified in textual syntax that defines behavior of the *ToCorrespondingTuples* Transformer. It receives two parameters: incoming and outgoing messages. It then obtains orders and transaction collections from the incoming message payload. Order collection is transformed to the new collection of tuples, which contain order and corresponding transaction. The transformation is accomplished using the *for*

operator. Corresponding transactions are obtained from transaction collection using the *select* function. It filters the collection using the supplied lambda expression. The resulting collection of tuples is then set as the payload of the outgoing message. Figure 58 shows the same transformation model expressed using graphical syntax.

```
void process(Message input, Message output) {
   List<Order> orders = input.payload.orders;
   List<Transaction> transactions = input.payload.transactions;

   output.payload.result =  for(o : orders) returning new Tuple(o,
       transactions.select(t | t.orderId == o.orderId));
}
```

Figure 57: Message transformation modeled using the textual DSL

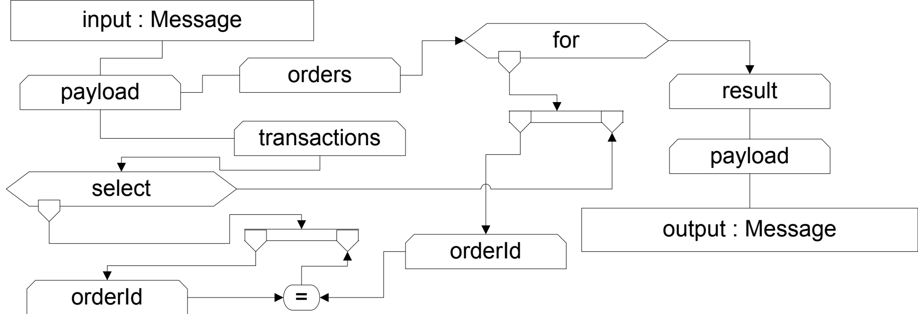


Figure 58: Message transformation modeled using the graphical DSL

9 Model Transformation

Model transformation is one of the central steps in model-driven engineering. It enables machine processable and automated refinement/abstraction over multiple levels of hierarchy. OMG suggests the following logical order of model transformation: CIM → PIM → PSM. Here we adopt slightly different approach and perform CIM → PSM → PIM transformation. The reason is that we perform system integration and need to collect technical information about the existing interfaces first (PSM level) and then abstract them in order to compare and mediate between them (PIM level). Correspondingly one has to define transformation rules between the PSMM to PIMM to enable this step. Formally, a model transformation has to define the way from the source model M_A conforming to metamodel MM_A to the target model M_B conforming to metamodel MM_B (Figure 59).

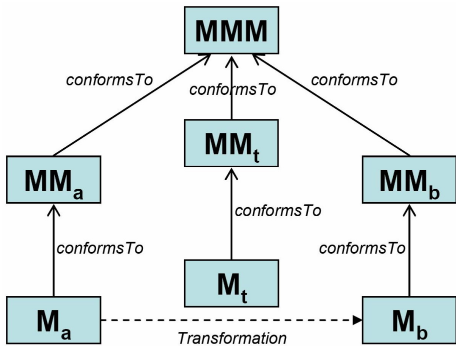


Figure 59: Model transformation hierarchy

In our approach we use the Atlas Transformation Language (ATL) [2] to specify and perform model transformation. It is a hybrid language and combines imperative and relational concepts. Operations on sets are carried out by OCL 2.0 constraints. This is useful to select the correct source elements from the source model needed in the transformation rules. ATL is available as a plugin for the Eclipse Modeling Framework (EMF). The input artifacts for model transformation are two metamodels (source and target) in Ecore format and one source model conforming to the source metamodel. The output artifact is a model conforming to the target metamodel.
The abstraction from platform specific to platform independent models is necessary for the comparison of two (or more) different systems. Within one uniform metamodel one compares the structure, behavior, semantic and properties of the two systems. In the BIZYCLE framework this comparison is made within the conflict analysis [5].
In this report we cover only the transformation from PSMM to PIMM, that is, abstraction of technical interface descriptions to the common, platform independent level. For that purpose, PSMM packages are transformed into PIMM

packages, as depicted in Figure 60. For each PSMM package, an equivalent PIMM package exists. Thus, core packages of all PSMMs are transformed into single PIMM core package. The same is done for other packages such as structure, property, semantic or communication. In this process generalization is performed. Also note that core, structure and communication packages are transformed, while property and semantic packages are extended. The reason is that semantic and property attributes are specified as annotations using model weaving, which although being a technical aspect only, merits the distinction. Details of transformations for all systems and packages are given in the following sections, with one exception: we discuss value range transformation here, as it is equal for all PSMMs. Furthermore, it can serve as a quick example of model transformation.

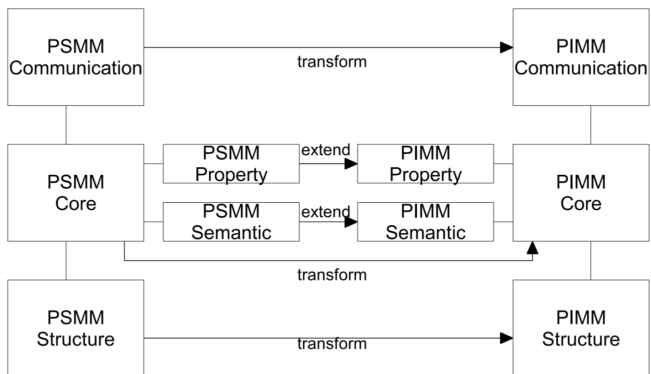


Figure 60: Model transformation in BIZYCLE

The transformation for parameter value ranges are equal within all PSMMs (see Table 10). In figure 61 an example is given where the value range model from chapter 3 is transformed to the PIM level. The `Parameters` at PSM level are referenced with a defined `ValueRange` within the `StructRoot`. The `Entries` will be transformed to `AnnotatableValues` at PIM level. At PIM level all `TypedElements` can have `AnnotatableValues`.

PSMM	⇒	PIMM
StructRoot + ValueRange + Entry	⇒	... + TypedElement + AnnotatableValue

Table 10: Value range transformation rule

9.1 Core Package Transformations

9.1.1 SAP R/3

The root element `SAP_R3` of the SAP R/3 metamodel is transformed to the element `ModelRoot` of the PIMM. The `BAPI` and the `IDOCType` elements are transformed to `Functional-` and `DocumentInterface`.

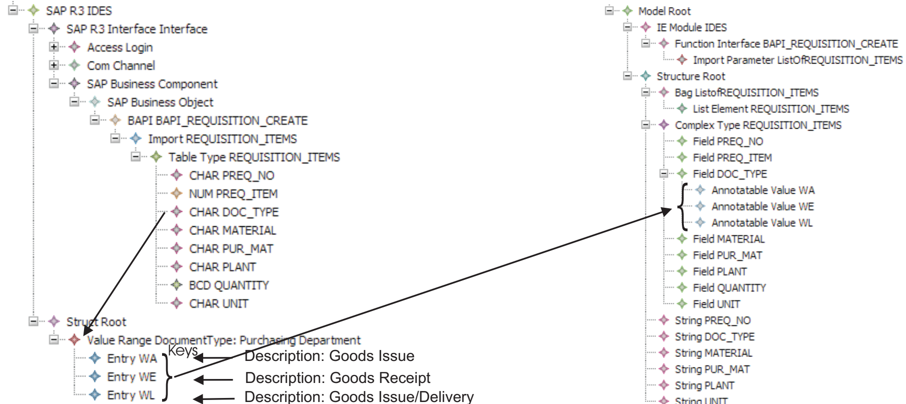


Figure 61: Value range transformation example

The element `IDOCType` consists of the control and data record. Furthermore the `DataRecord` is made up of `Segments` and `Fields`. The proper `Parameter` types are the `Fields`, so only the IDOC `Field` will be generalized to a `DocumentElement`. `Segments` exist only to sort the `Fields`, so they are not required in PIMM. The `ControlRecord` is also not required, because it consist only of administrative information.

PSMMSAPJCO	⇒	PIMM
SAP R3 Interface	⇒	IEModul
... + BAPI	⇒	IEModul + FunctionInterface
... + IDOCTYPE	⇒	IEModul + IEDocumentRoot + DocumentInterface
... + IDOCField	⇒	IEModul + IEDocumentRoot + DocumentInterface + DocumentElement
... + Import	⇒	IEModul + FunctionInterface + ImportParameter
... + Export	⇒	IEModul + FunctionInterface + ExportParameter

Table 11: Transformation of SAP R/3 core package to PIMM

9.1.2 Relational Database Systems

The root element `DBMS` and the `SQL_Interface` element are transformed to the PIMM elements `ModelRoot` and `IEModul`. All `Query` elements are transformed to `FunctionalInterfaces`. The `Input` and `Output` parameter are transformed to `Import` and `Output` parameter at PIMM level and are collected under the `FunctionalInterfaces`. There are two different ways of transforming SQL queries to PIM level. If the `Query` is a `Read` Query, we transform the `Input` parameter to a simple `ImportParameter` with a `SimpleType`. The `Output` parameter will be transformed to `ExportParameter`, but the type is a `Bag` with `ListElements`. The structure of the `ListElements` will be defined within a `ComplexType` with further `Fields` in it. The `Fields` are defined with `SimpleTypes`. The second way of transforming SQL Queries regards the `Create`, `Update` and `Delete` query. Here, the `Input` parameter will be transformed to `ImportParameter` with a `SimpleType`. The `Output` parameter will

be transformed to ExportParameter with a SimpleType.

PSMMRELDBSQLJDBC	⇒	PIMM
DBMS	⇒	ModelRoot
SQL_Interface	⇒	IEModul
SQL_Interface + Query	⇒	IEModul + FunctionInterface
... + Read + Input	⇒	IEModul + FunctionInterface + ImportParameter
	⇒	StructRoot + SimpleType
... + Read + Output	⇒	IEModul + FunctionInterface + ExportParameter
	⇒	StructRoot + Bag + ListElement
	⇒	StructRoot + ComplexType + Field
... + Create + Input	⇒	IEModul + FunctionInterface + ImportParameter
	⇒	StructRoot + Bag + ListElement
	⇒	StructRoot + ComplexType + Field
... + Create + Output	⇒	IEModul + FunctionInterface + ExportParameter
	⇒	StructRoot + SimpleType
... + Update + Input	⇒	IEModul + FunctionInterface + ImportParameter
	⇒	StructRoot + SimpleType
... + Update + Output	⇒	IEModul + FunctionInterface + ExportParameter
	⇒	StructRoot + SimpleType
... + Delete + Input	⇒	IEModul + FunctionInterface + ImportParameter
	⇒	StructRoot + SimpleType
... + Delete + Output	⇒	IEModul + FunctionInterface + ExportParameter
	⇒	StructRoot + SimpleType

Table 12: Transformation of relational database core package to PIMM

9.1.3 Java 2 Platform, Enterprise Edition(J2EE)

The root element J2EE of the J2EE metamodel is transformed to the element ModelRoot of the PIMM. The J2EE_Component and EnterpriseJavaBean are transformed to IEModul. The IEModul can contain further IEModuls as child elements, which are transformed EnterpriseJavaBeans from the PSM level. The FunctionInterfaces are transformed Methods of the EnterpriseJavaBeans. The Input and Output elements of the PSM level are transformed to Import- and ExportParameter.

J2EE	⇒	PIMM
J2EE	⇒	ModelRoot
... + J2EE_Component	⇒	IEModul
... + EnterpriseJavaBean	⇒	IEModul + IEModul
... + J2EE_Interface + Method	⇒	... + IEModul + FunctionInterface
... + Input	⇒	... + FunctionInterface + ImportParameter
... + Output	⇒	... + FunctionInterface + ExportParameter

Table 13: Transformation of J2EE core package to PIMM

9.1.4 Web Services

The transformation of Web service core metamodel package processes only a subset of WSDL elements, more precisely those elements which are relevant for the conflict analysis at the PIMM level. The element ModelRoot is transformed to the element ModelRoot of the PIMM. The WSDLDefinition is transformed to IEModul. An Operation will be transformed to a FunctionInterface. Parameters (InputParameter, OutputParameter) and the referenced Messages and MessageParts are transformed to ImportParameters and ExportParameters respectively. A MessagePart can reference either a Type or an Element. In case of an Element-reference, the Type of this referenced Element is used as the Type for the Parameter. All Type specific transformations are described in chapter 9.2.6.

PSMMWSDL	⇒	PIMM
ModelRoot	⇒	ModelRoot
WSDLDefinition	⇒	IEModul
WSDLDefinition + Operation	⇒	IEModul + FunctionInterface
... + InputParameter + Message + MessagePart	⇒	IEModul + FunctionInterface + ImportParameter
... + OutputParameter + Message + MessagePart	⇒	IEModul + FunctionInterface + ExportParameter

Table 14: Transformation of Web services core package to PIMM

9.1.5 Extensible Markup Language (XML)

The transformation of XML metamodel core package uses the XSD Lite rules from chapter 9.2.6.

9.2 Structure Package Transformations

9.2.1 SAP R/3

The types of the `Import` and `Export` Parameters are collected under `Structure Root`. The three different types, `FieldType`, `StructType` and `TableType`, are transformed to `SimpleType`, `Complex Type` and `Bag` respectively. Every `Import` and `Export` Parameter has its own type within the PSM, after the transformation to PIM as well.

PSMMSAPJCO	⇒	PIMM
... + Struct + SimpleType	⇒	StructRoot + ComplexType + Field
... + Table + SimpleType	⇒	StructRoot + Bag + ListElement
	⇒	StructRoot + ComplexType + Field
... + CHAR	⇒	StructRoot + STRING
... + STRING	⇒	StructRoot + STRING
... + XSTRING	⇒	StructRoot + STRING
... + BYTE	⇒	StructRoot + OPAQUE
... + FLOAT	⇒	StructRoot + NUMBER
... + DATE	⇒	StructRoot + PointInTime
... + TIME	⇒	StructRoot + PointInTime
... + BCD	⇒	StructRoot + NUMBER
... + NUM	⇒	StructRoot + NUMBER
... + INT	⇒	StructRoot + NUMBER
... + INT1	⇒	StructRoot + NUMBER
... + INT2	⇒	StructRoot + NUMBER

Table 15: Transformation of SAP R/3 structure package to PIMM

9.2.2 Relational Database Systems

The `JDBC types` are transformed to simple PIMM types. There are no `Complex Types`. Transformations are given in Table 16.

9.2.3 Java 2 Platform, Enterprise Edition(J2EE)

J2EE `ComplexTypes` are transformed one to one to PIMM complex types, as child elements of the `StructureRoot` at the PIM level (Table 17).

9.2.4 Web Services

The transformation rules are covered by XSD Lite (see chapter 9.2.6).

PSMMRELDBSQLJDBC	⇒	PIMM
... + CHAR	⇒	StructRoot + STRING
... + VARCHAR	⇒	StructRoot + STRING
... + LONGVARCHAR	⇒	StructRoot + STRING
... + NUMERIC	⇒	StructRoot + Number
... + DECIMAL	⇒	StructRoot + Number
... + TINYINT	⇒	StructRoot + Number
... + SMALLINT	⇒	StructRoot + Number
... + INTEGER	⇒	StructRoot + Number
... + BIGINT	⇒	StructRoot + Number
... + REAL	⇒	StructRoot + Number
... + FLOAT	⇒	StructRoot + Number
... + BINARY	⇒	StructRoot + Opaque
... + VARBINARY	⇒	StructRoot + Opaque
... + LONGVARBINARY	⇒	StructRoot + Opaque
... + DATE	⇒	StructRoot + Date
... + TIME	⇒	StructRoot + Time
... + TIMESTAMP	⇒	StructRoot + Date
... + CLOB	⇒	StructRoot + STRING
... + BIT	⇒	StructRoot + BOOLEAN

Table 16: Transformation of relational database structure package to PIMM

J2EE	⇒	PIMM
... + ComplexType + Field	⇒	ComplexType + Field
... + ComplexType + Method	⇒	ComplexType + Method
... + CollectionType	⇒	StructureRoot + Bag/Set
... + CollectionType + SimpleType	⇒	StructureRoot + Bag/Set + SimpleType
... + BYTE	⇒	StructRoot + Opaque
... + BOOLEAN	⇒	StructureRoot + Boolean
... + FLOAT	⇒	StructureRoot + Number
... + DOUBLE	⇒	StructureRoot + Number
... + LONG	⇒	StructureRoot + Number
... + INTEGER	⇒	StructureRoot + Number
... + SHORT	⇒	StructureRoot + Number
... + STRING	⇒	StructureRoot + STRING
... + BIGDECIMAL	⇒	StructureRoot + Number
... + DATE	⇒	StructureRoot + PointInTime
... + TIME	⇒	StructureRoot + PointInTime

Table 17: Transformation of J2EE structure package to PIMM

9.2.5 Extensible Markup Language (XML)

The transformation rules are covered by XSD Lite (see chapter 9.2.6).

9.2.6 XSD Lite

The XSDLite-to-PIMM transformation is used by two other transformations: Web services and XML. The XSDLite metamodel represents a simplification of the XML Schema Definition (XSD), hence the corresponding XSD Lite transformation rules consider only structural aspects as shown in table 18.

9.3 Communication Package Transformation

9.3.1 SAP R/3

The communication channel synchronous RFC is abstracted to `SynchronCall`. The transactional, queued and asynchronous RFC are abstracted to `AsynchronCall` at the PIMM level.

XSD Lite	⇒	PIMM
Type	⇒	Type
XsdBaseType	⇒	SimpleType
'string'	⇒	String
'int'	⇒	Number
'short'	⇒	Number
'unsignedShort'	⇒	Number
'unsignedInt'	⇒	Number
'unsignedLong'	⇒	Number
'positiveInteger'	⇒	Number
'negativeInteger'	⇒	Number
'nonPositiveInteger'	⇒	Number
'nonNegativeInteger'	⇒	Number
'float'	⇒	Number
'long'	⇒	Number
'double'	⇒	Number
'decimal'	⇒	Number
'date'	⇒	PointInTime
'dateTime'	⇒	PointInTime
'time'	⇒	PointInTime
'gDay'	⇒	PointInTime
'gMonth'	⇒	PointInTime
'gMonthDay'	⇒	PointInTime
'gYear'	⇒	PointInTime
'gYearMonth'	⇒	PointInTime
'byte'	⇒	Opaque
'base64Binary'	⇒	Opaque
'anyURI'	⇒	String
'QName'	⇒	String
'Name'	⇒	String
'NCName'	⇒	String
'language'	⇒	String
'token'	⇒	String
'NMTOKEN'	⇒	String
'NMTOKENS'	⇒	String
'normalizedString'	⇒	String
ComplexType	⇒	ComplexType
... + Container + ElementRef	⇒	Field reference
Element	⇒	Field
Attribute	⇒	Field

Table 18: Transformation of XSD structure package to PIMM

PSMMSAPJCO	⇒	PIMM
SAP R3 Interface + CommunicationChannel + sRFC	⇒	CommunicationRoot + SynchronCall
SAP R3 Interface + CommunicationChannel + aRFC	⇒	CommunicationRoot + AsynchronCall
SAP R3 Interface + CommunicationChannel + tRFC	⇒	CommunicationRoot + AsynchronCall
SAP R3 Interface + CommunicationChannel + qRFC	⇒	CommunicationRoot + AsynchronCall

Table 19: Transformation of SAP R/3 communication package to PIMM

9.3.2 Relational Database Systems

The SQL queries from PSMM level have either the communication channel JDBC or ODBC and are abstracted to `SynchronCalls` within the communication package at the PIMM level. Flat files have `AsynchronCall` at the PIMM level.

PSMMRELDBSQLJDBC	⇒	PIMM
SQL Interface + CommunicationChannel + JDBC	⇒	CommunicationRoot + SynchronCall
SQL Interface + CommunicationChannel + ODBC	⇒	CommunicationRoot + SynchronCall
SQL Interface + CommunicationChannel + Flatfile	⇒	CommunicationRoot + AsynchronCall

Table 20: Transformation of relational database communication package to PIMM

9.3.3 Java 2 Platform, Enterprise Edition(J2EE)

J2EE components use RMI IIOP as the `CommunicationChannel`. It is abstracted to a `SynchronCall` at the PIMM level.

J2EE	⇒	PIMM
J2EE + RMI IIOP	⇒	CommunicationRoot + SynchronCall

Table 21: Transformation of J2EE communication package to PIMM

9.3.4 Web Services

The Web services metamodel differentiates between `MessageExchangePatterns` as part of the `communication` package. Every pattern belongs to exact one `Operation`. Those patterns are general enough to be part of the PIMM. The transformation rules are given in Table 22.

PSMMWSDL	⇒	PIMM
Operation +	⇒	CommunicationRoot +
...InOnly	⇒	...InOnly
...RobustInOnly	⇒	...RobustInOnly
...InOut	⇒	...InOut
...InOptionalOut	⇒	...InOptionalOut
...OutOnly	⇒	...OutOnly
...RobustOutOnly	⇒	...RobustOutOnly
...OutIn	⇒	...OutIn
...OutOptionalIn	⇒	...OutOptionalIn

Table 22: Transformation of Web services communication package to PIMM

9.3.5 Extensible Markup Language (XML)

The XML metamodel differentiates between three `AccessPatterns`, each represents a container for XML-based data: (1) XML wrapped by an Email, (2) XML accessible by web server and (3) file-based XML. Transformations are given in Table 23.

XML	⇒	PIMM
CommunicationRoot + EmailBoxAccess	⇒	CommunicationRoot + SynchronCall
CommunicationRoot + WebAccess	⇒	CommunicationRoot + SynchronCall
CommunicationRoot + FileAccess	⇒	CommunicationRoot + SynchronCall

Table 23: Transformation of XML communication package to PIMM

9.4 Semantic Package Transformations

9.4.1 SAP R/3

The annotations (AnnotatableElement) of `ImportParameter`, `ExportParameter` and `Segment` elements are abstracted to annotations of `ImportParameter`, `ExportParameter` and `DocumentElement` elements at the PIMM level. The annotations (AnnotatableFunction) of the elements `SAP_R/3_Interface` and `Method` are abstracted to annotations of `IEModule` and `FunctionInterface`.

9.4.2 Relational Database Systems

The annotations (AnnotatableElement) of `OutputParameter` and `InputParameter` elements are abstracted to annotations of `ImportParameter` and `ExportParameter` at the PIMM level. The annotations (AnnotatableFunction) of the element `SQL_Interface` is abstracted to annotations of `FunctionInterface` on PIMM level.

9.4.3 Java 2 Platform, Enterprise Edition(J2EE)

The annotations (AnnotatableElement) of `ImportParameter`, `Export Parameter`, `Field` of `ComplexType` and `CollectionElements` of `Collections` are abstracted to annotations of `ImportParameter`, `ExportParameter`, `Fields` of `ComplexTypes` and `ListElements` of `AbstractCollections` at the PIMM level.

9.4.4 Web Services

The annotations (AnnotatableElement) of `ImportParameter`, `ExportParameter`, `FaultParameter`, `Message` and `MessagePart` are abstracted to annotations of `ImportParameter`, `ExportParameter` and `Fields`. The annotations (AnnotatableElement) of `Operations` are abstracted to annotations of `FunctionInterfaces`.

9.4.5 XSD Lite

The annotations (AnnotatableElement) of `Attribute` and `Element` are abstracted to annotations of `Fields`.

9.5 Transformation Examples

9.5.1 SAP R/3

The example shows exemplary the transformation of the PSM model from chapter 3.1.5 to the PIM level (see figure 62).

9.5.2 Relational Database Systems

The example shows exemplary the transformation of the PSM model from chapter 3.2.5 to the PIM level (see figure 63).

9.5.3 Java 2 Platform, Enterprise Edition(J2EE)

The example shows exemplary the transformation of the PSM model from chapter 3.3.5 to the PIM level (see figure 64).

9.5.4 Web Services

In figure 65 the example web service from chapter 3.4.5 is transformed to the PIM level.

9.5.5 Extensible Markup Language (XML)

The XML-to-PIM transformation uses the structure rules from the Web service-to-PIM transformation (figure 65). These structure rules based on the shared XSDlite metamodel, which is used by the WSDL and the XML metamodel. Therefore additional XML transformation example is not provided here.

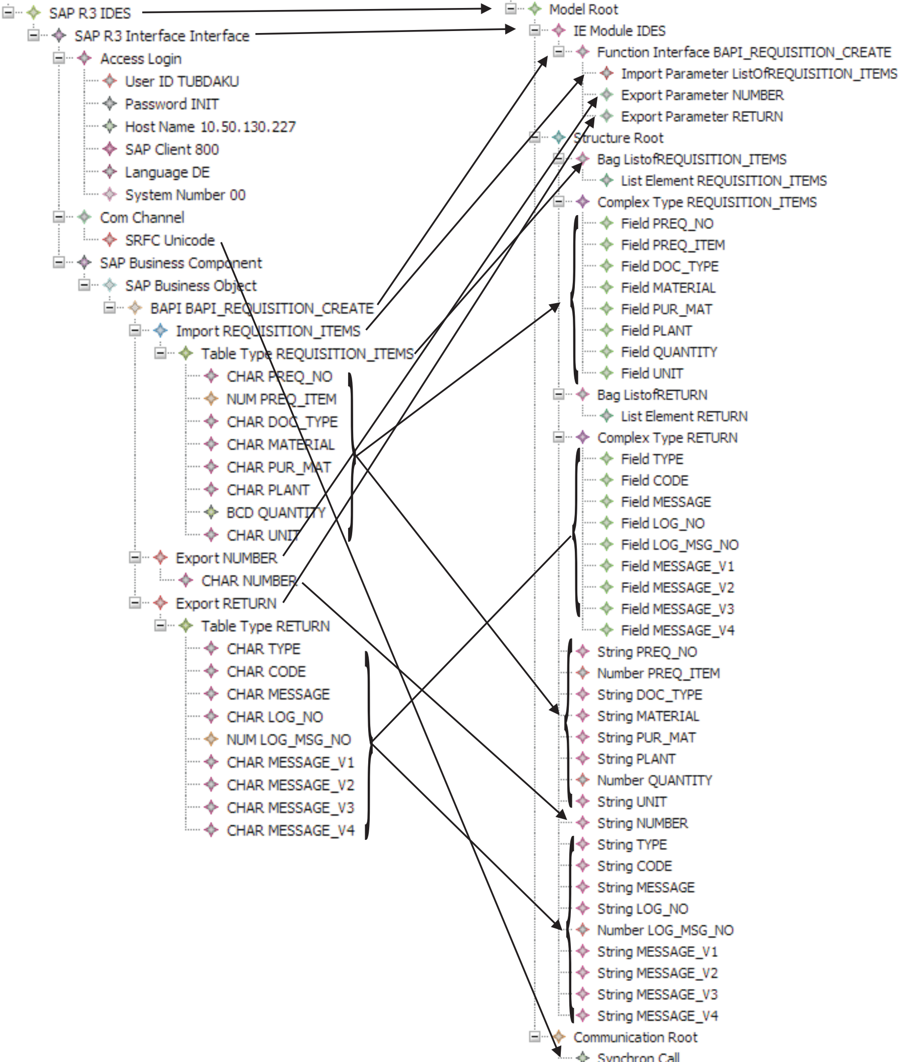


Figure 62: SAP R/3 transformation example

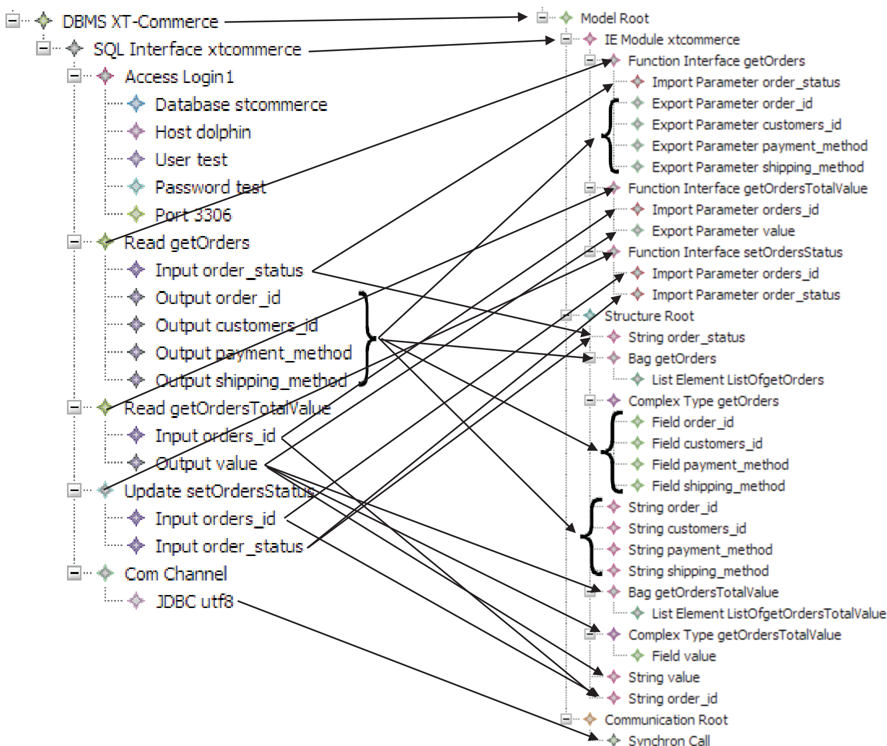


Figure 63: Relational database transformation example

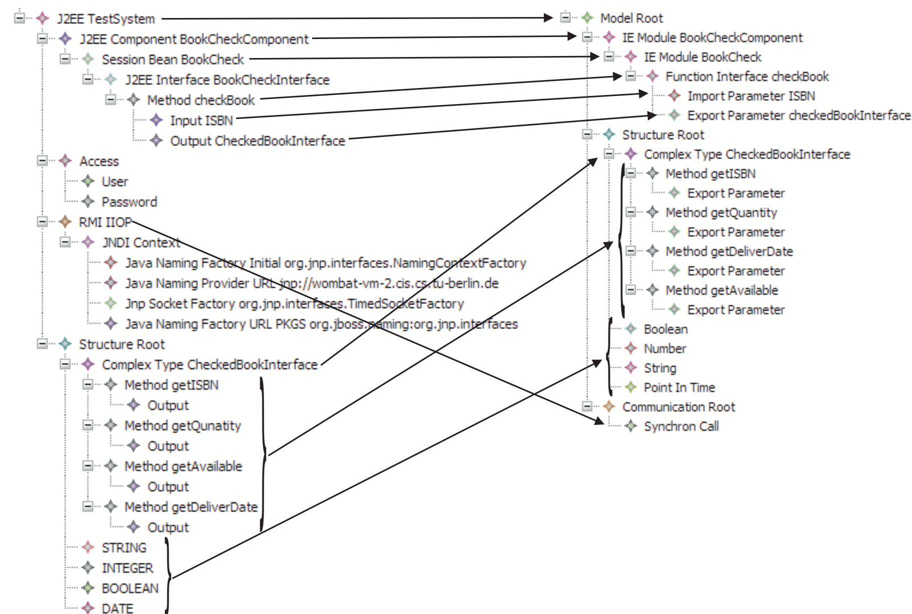


Figure 64: J2EE transformation example

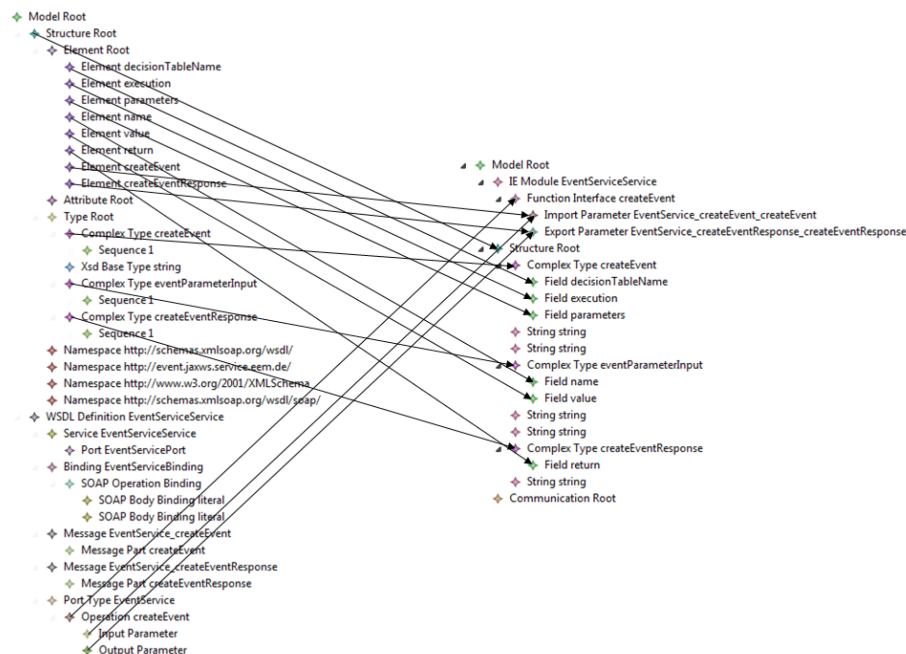


Figure 65: Web services transformation example

10 Summary

This technical report summarizes metamodels and transformations developed in the BIZYCLE project. In Chapter 1 basic information about the project are given, placing subsequent work in the context of model-based software integration. Chapter 2 provides information about computation independent metamodel, which is used for modeling of integration business processes. In Chapter 3 an overview of platform specific models is performed, including following platforms/systems: SAP R/3, relational databases, J2EE components, Web services and flat XML files. Chapter 4 introduces platform independent metamodel, which serves as a common abstraction level at which integration conflict analysis process can be performed. Semantics is crucial issue in software integration, therefore Chapters 5 and 6 propose a method for modeling of semantic knowledge (semantic metamodel) and semantic annotation (annotation metamodel) of integration business processes as well as platform specific interface descriptions. Property metamodel, presented in Chapter 7 enables modeling of integration specific non-functional properties, such as availability, reliability or security. Finally, the connector metamodel (Chapter 8) uses Enterprise Application Integration (EAI) patterns to capture structure and behavior of the connector component that serves as a mediator between systems targeted for integration.

Chapter 9 focuses on model transformations, which are an essential part of model-based integration process. Model transformations, that is, abstractions from platform specific (PSM) to platform independent (PIM) level are described in detail for all proposed metamodels, as they serve as the basis for the automated conflict analysis process.

References

[1] W3c web services description language(wsdl) 1.1, 2001. http://www.w3.org/TR/wsdl.

[2] ATL: Atlas Transformation Language User Manual. *http://www.eclipse.org/m2m/atl/doc/ATL_User_Manual[v0.7].pdf*, 2006.

[3] W3c xml schema, 2008. http://www.w3.org/XML/Schema.

[4] H. Agt, G. Bauhoff, M. Cartsburg, D. Kumpe, R. Kutsche, and N. Milanovic. Metamodeling Foundation for Software and Data Integration. In *Proc. ISTA*, 2009.

[5] H. Agt, J. Widiker, G. Bauhoff, R.-D. Kutsche, and N. Milanovic. Model-based semantic conflict analysis for software- and data-integration scenarios. Technical Report 2009/7, Technische Universität Berlin, 2009.

[6] atl. Atlas transformation language (atl). http://www.eclipse.org/m2m/atl/.

[7] N. Boudjlida and H. Panetto. Annotation of enterprise models for interoperability purposes. In *Proceedings of the IWAISE 2008*, 2008.

[8] EMF. *Eclipse Modeling Framework*. ?, 2008.

[9] M. D. D. Fabro, J. Bézivin, F. Jouault, E.Breton, and G. Gueltas. AMW: a generic model weaver. In *Proceedings of IDM05*, 2005.

[10] T. Hildenbr and R. Gitzel. A taxonomy of metamodel hierarchies.

[11] G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2003.

[12] A. Kiryakov, B. Popov, D. Ognyanoff, D. Manov, A. Kirilov, and M. Goranov. Semantic annotation, indexing, and retrieval. In *International Semantic Web Conference*, 2003.

[13] T. Kühne. Matters of (meta-)modeling. *Software and System Modeling*, 5(4):369–385, 2006.

[14] R. Kutsche and N. Milanovic. (Meta-)Models, Tools and Infrastructures for Business Application Integration. In *UNISCON 2008*. Springer Verlag, 2008.

[15] R. Kutsche, N. Milanovic, G. Bauhoff, T. Baum, M. Cartsburg, D. Kumpe, and J. Widiker. BIZYCLE: Model-based Interoperability Platform for Software and Data Integration. In *Proceedings of the MDTPI at ECMDA*, 2008.

[16] N. Milanovic, M. Cartsburg, R. Kutsche, J. Widiker, and F. Kschonsak. Model-based Interoperability of Heterogeneous Information Systems: An Industrial Case Study. In *Proceedings ECMDA*, 2009.

[17] N. Milanovic, R. Kutsche, T. Baum, M. Cartsburg, H. Elmasgunes, M. Pohl, and J. Widiker. Model & Metamodel, Metadata and Document Repository for Software and Data Integration. In *Proceedings of the ACM/IEEE MODELS*, 2008.

[18] E. Pulier and H. Taylor. *Understanding Enterprise SOA*. Manning, 2006.

[19] L. Reeve and H. Han. Survey of semantic annotation platforms. In *Proceedings of the 2005 ACM symposium on Applied computing*, NY, USA, 2005. ACM.

[20] M. Shtelma, M. Cartsburg, and N. Milanovic. Executable Domain Specific Language for Message-based System Integration. In *Proc. MoDELS*, 2009.

[21] H. Wada, E. M. M. Babu, A. Malinowski, J. Suzuki, and K. Oba. Design and implementation of the matilda distributed uml virtual machine. In *Proc. of the 10th IASTED SEA Conference*, 2006.

[22] H. Wada and J. Suzuki. Modeling turnpike: A model-driven framework for domain-specific software development. In *Proc. of the Doctoral Symposium at the 8th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS/UML 2005)*, 2005.