# Web Application Specification Language (WASL)

Ulrich Wolffgang
European Research Center for Information Systems (ERCIS)
University of Münster

2009-09-22
Document version 1.0

## Contents

## Introduction

The *Web Application Specification Language* (WASL) family is a set of semi formal modeling languages for the model-driven development of web applications. The languages are interconnected by transformation definitions, which support the transformation of platform-independent WASL models to platform-specific WASL models and from platform-specific WASL models to source code.

The WASL language family is an implementation of MDA vision introduced by the Object Management Group (OMG), which classifies models in a MDA-based software development process as computation-independent models (CIM), platform-independent models (PIM) and platform-specific models (PSM) [1]. As a common metamodeling language the OMG proposes the meta-object facility (MOF), for which an implementation exists with the Eclipse Modeling Framework (EMF) and its meta metamodel Ecore that slightly differs from MOF in some details [2]. All WASL languages are based on the Ecore meta metamodel because it provides large tool support through the Eclipse Modeling Framework (EMF), several model-to-model transformation languages, code generators and standardized file formats such as XMI.

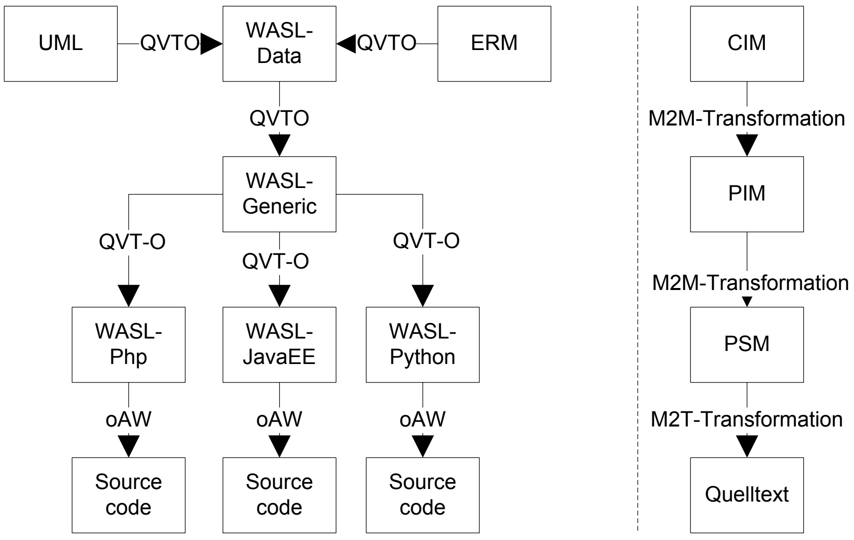

*Fig. 1: WASL generator framework*

In WASL the web application development process starts by creating a conceptual data model, which represents the structure of the domain as a CIM. For this step the modeling languages ERM, UML and WASL Data can be used. As WASL Data is the source language for further transformations, models written in ERM or UML have to be transformed to WASL Data models first. This is achieved with a transformation definition, which is written in the standardized transformation language QVT Operational.

PIM layer models are specified with the semi formal modeling language WASL Generic, which offers language elements for modeling the data structure, navigation structure, business logic and presentation structure of a web application. As described WASL

Generic models can be generated from WASL Data models with a transformation definition using QVT Operational. In the transformation step the data model is copied from the WASL Data model to the WASL Generic model as both languages represent data models in the same way. Additionally the contents of the navigation, logic and presentation models are generated from the contents of the WASL Data model automatically. The generated business logic and presentation aspects represent the functionality of the web application for *create*, *read*, *update* and *delete* (CRUD) operations on the entities described by the data model. The generated WASL Generic model serves as an initial point for modifications on the generated model elements and enrichments by adding new model elements for additional functionality.

In contrast to the PIM language WASL Generic, platform-specific implementation details are modeled with the three semi-formal modeling languages WASL Php, WASL Python and WASL JavaEE. WASL Generic models can be transformed to models written in these three languages by three QVT-Operational transformation definitions, which map the Generic PIM concepts to detailed technical PSM realizations. Finally the transformation from the PSM level to the source code level is implemented with three individual code generators based on the openArchitecture (oAW) code generator framework.

The goal of the WASL MDSD approach is to provide a fast and reliable way of generating web applications based on proven design patterns and thus support the usage of those patterns in the web development community. However this does not include recreating detailed program logic in models. The approach of imitating source code in models is not considered useful because MDSD should reduce error-prone software engineering practices and is not intended to introduce another layer of high complexity. Thus a major principle of the WASL family is that method bodies of object-oriented models are not modeled explicitly. In contrast methods are declared and enriched with additional required information for generating the complete source code of the method’s body automatically.

## Platform-independent languages

### WASL Data 1.0

WASL Data is a semiformal modeling language for CIM modeling in the WASL MDSD stack and offers language elements for representing the structural domain aspects of a web application in the form of a data model.

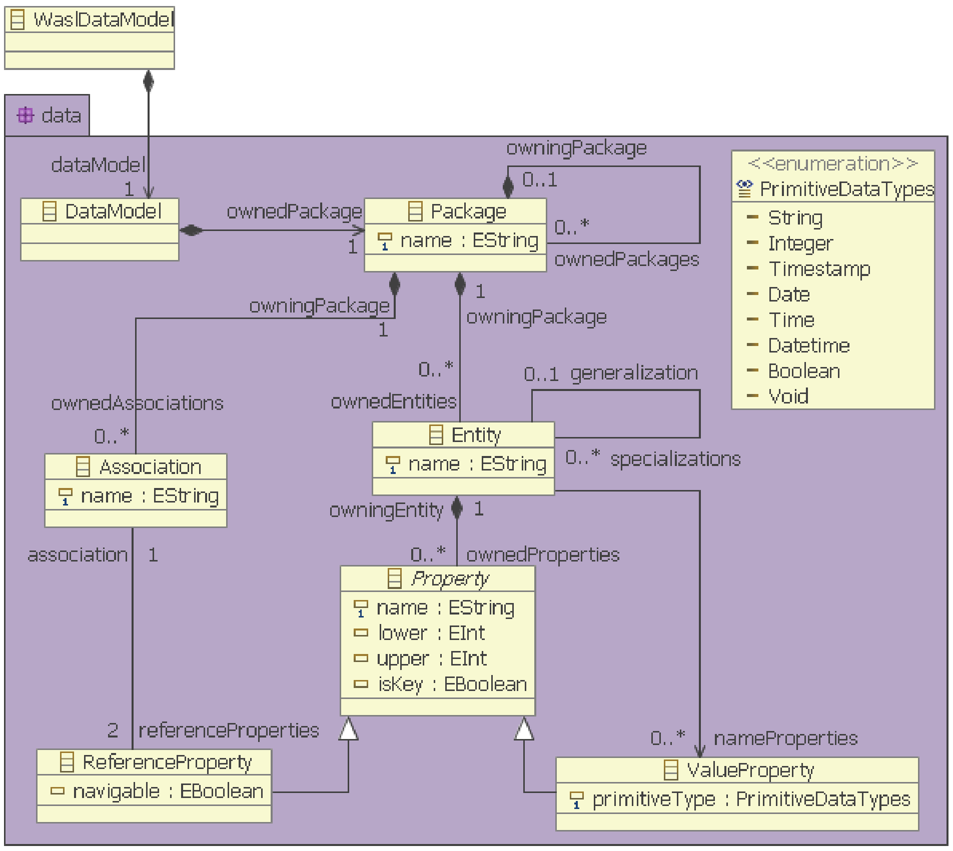

*Fig. 2: Metamodel of WASL Data*

The metamodel of WASL Data resembles the UML metamodel for class diagrams and generally offers language elements for specifying packaged entity types, typed properties and associations between these properties. The central language element *Entity* represents an entity type of the conceptual domain. A type hierarchy can be built with the EReference *generalization* for defining the super type of an entity and with the EReference *specializations* for multiple opposite subtypes. An entity can possess properties, which are subdivided into *ValueProperties* and *ReferenceProperties*. The super type *Property* allows merging collections of both *Property* types in one single collection. This is important for specifying an ordered set of properties mixed of *ReferenceProperties* and *ValueProperties*, which for instance is represented in the resulting web application as an ordered list of input fields and select boxes contained in a HTML form.

Each *ValueProperty* is typed with a value from the EEnum *PrimitiveDataTypes*, which is a list of all provided platform-independent primitive data types. For example the type *PrimitiveDataType::UnsignedInteger* can be assigned to a *ValueProperty* instance named *age* for modeling the age of a person. Using an enumeration of fixed primitive types offers the advantage of type-specific and type-safe mappings in subsequent model-to-model-transformations. E. g. it can be specified that both the primitive PIM types *Integer* and *UnsignedInteger* should be transformed to the PSM primitive SQL type *BIGINT*.

Each *Entity* is contained in a *Package* which itself can be nested in a super package. This allows structuring *Entities* in a hierarchy grouped by domain concepts. Associations between *ReferenceProperties* are modeled by the metamodel element *Association*, which provides bidirectional navigable relationships between *Properties*. An association can reference the same *Entity* or two different *Entities*, so that the question arises where to store the association instance in the model. Similarly to UML an *Association* instance can be contained in any *Package* instance without any restriction regarding the packages, which are containing the *Entity* instances referenced by the *Association*.

Contrary to ERM in WASL Data entities can be packaged and properties are typed with a preset primitive type system. In contrast to the UML metamodel the WASL Data metamodel is reduced to a minimal feature set and provides special EAttributes for some metamodel elements such as the *Entity's nameProperties*. Additionally in UML the primitive types are not preset by the metamodel but are specified on the model layer with instances of the metamodel element *PrimitiveType*. With this approach a *ValueProperty* instance would be typed by assigning one of those *PrimitiveType* instances to the *ValueProperty* instance. The UML approach has the advantage of higher flexibility as the language can be configured while modeling by modifying the primitive type system. The downside of the approach is the ambiguity of primitive types that leads to the problem that primitive types can only be identified through textual comparison by their name and thus not be transformed type-safe in a type-to-type-mapping. Because of this issue the approach of using an enumeration of fixed primitive types is chosen for WASL Data.

## WASL Generic 1.0

WASL Generic is a platform-independent semi-formal modeling language, which offers language elements for modeling data structures, navigational structures, presentation structures and business logic inducing a separation of concerns. The language elements for the data model comply with WASL Data so that WASL Generic can be interpreted as an extension of WASL Data.

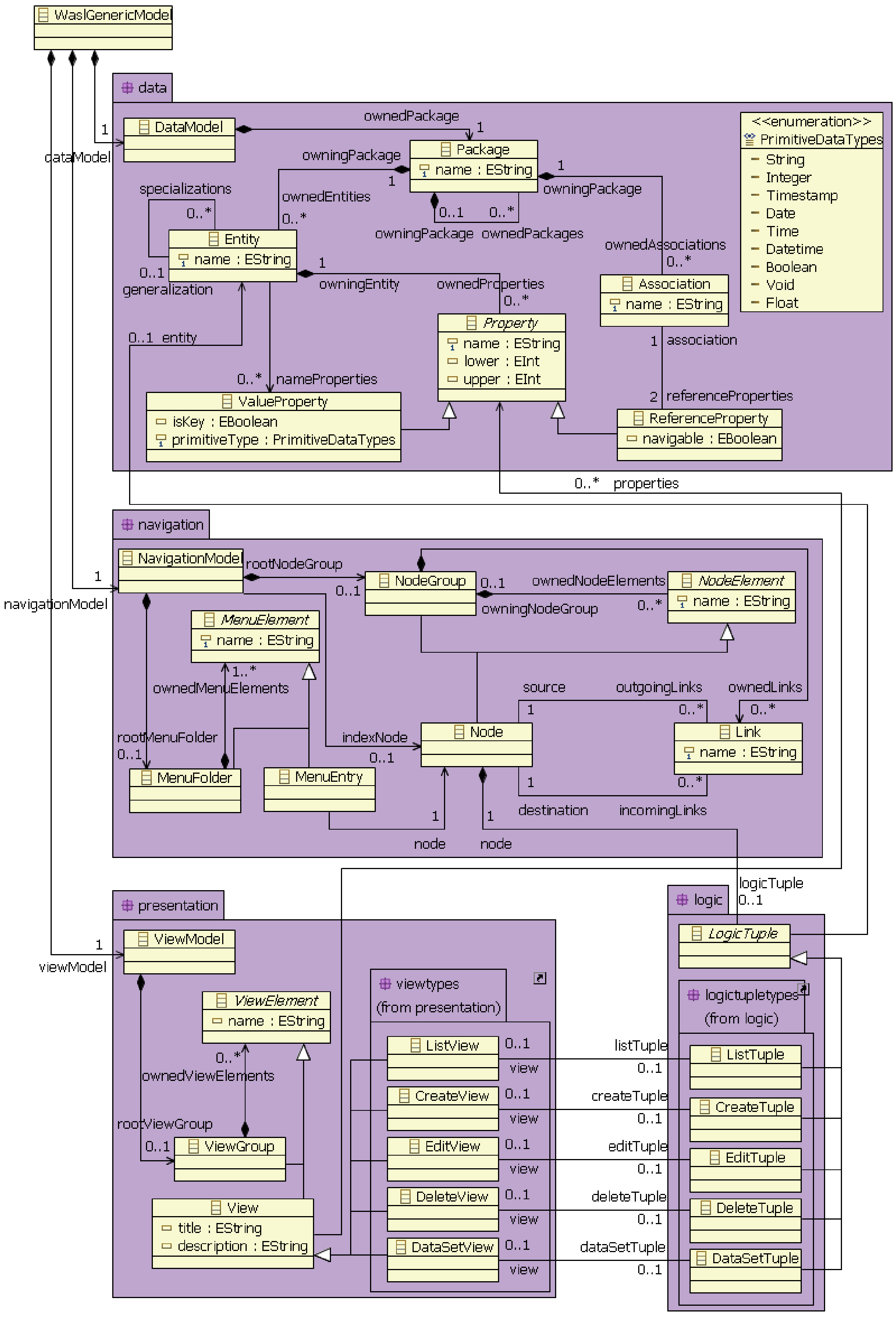

*Fig. 3: Metamodel of WASL Generic*

The language elements for the navigational model are contained in the EPackage n*avigation* and represent the web application's navigation structure in the form of a directed multigraph. Web applications consist of pages or respectively views between which the user interactively can navigate with the browser e.g. by clicking links and buttons. In WASL Generic the term *Node* is defined as a part of a web application covering delimited specific functionality, addressed by a unique URL and navigable by the user with the browser´s controls such as back and forth buttons as well as bookmarks. Normally a *Node* corresponds to one single web page; however other implementations with multiple pages are possible as long as only one URL is presented to the user for these pages. Also platform-specific technologies like AJAX allow the content of a web page to be replaced after initially rendering it within the browser, leading to a multi-page approach based on a single page and URL.

The navigation path between two *Nodes* is expressed by a *Link*, which corresponds to a directed edge of the multigraph and is implemented platform-specifically e.g. by a hypertext anchor or a HTML form's *action* attribute. Each *Node* is contained in a *NodeGroup*, which itself can be nested in a super *NodeGroup* thus forming a hierarchy of *NodeGroups*. Usually the structure of the hierarchy is structured along the functional areas of the modeled web application.

The menu structure of the web application is modeled without using *Links* with the purpose to reduce the number of links in the navigational model and thus to reduce the complexity. Modeling the menu structure in the navigational model by links with a number of $n$ nodes would lead to a number of $n^2$ links as for each node a link to all nodes would have to be added. This does not comply with the requirement of clarity a model should satisfy. Also typically the menu of a web application is visualized and implemented separately from the rest of the page contents, so that a separated representation in the model simplifies the implementation of transformation definitions. Correspondingly to *NodeGroups* the menu can be structured hierarchically by *MenuFolders*.

In WASL Generic standard business logic pattern are represented separately from the *Node* concept as specializations of the metamodel element *LogicTuple*. A *LogicTuple* is assigned to a *Node* and aggregates the information, which *Node* operates on what *Entity* visualized by which *View*. The approach is flexible as additional business logic types can be appended to the metamodel as additional specializations of the element *LogicTuple*.

The business logic is represented to the user by one or more *Views* allowing modeling simple CRUD operations as well as for example more complex multi-page wizards. In contrast to *Nodes*, multiple *Views* owned by one *LogicTuple* must not be distinguishable for the user by individual URLs and as described above thus do not have to be navigable by the browser´s back and forward buttons or bookmarks. As navigation and presentation structure are separated from each other and AJAX functionality is represented only in the *views* of the presentation model the page-driven paradigm typical for web applications is ensured. AJAX introduces the problem that the functionality of the browser´s page-driven URL addressing mechanism is broken due to the fact, that with AJAX the page content is just replaced and no transition between two web pages distinguishable by two URLs is made. The separation of navigation and (AJAX-based) presentation in WASL Generic assures that web applications derived from WASL Generic do not suffer typical AJAX symptoms such as a missing browser history, broken back and forward buttons and indistinguishable URLs prohibiting the bookmarking of specific parts of the web application.

## Platform-specific languages

The platform-specific model languages presented in this chapter are intended to enable a direct representation of a platform's relevant system elements in the model for subsequent code generation. Thereby for WASL a translational approach is chosen instead of constructing virtual machines that execute the models directly. The terminology used in the PSM metamodels equates to the terminology of the respective used target languages and frameworks and thus differs from terminology utilized in the WASL PIM metamodels.

The metamodels are not graphically depicted in this document because the platform-specific WASL metamodels are larger and more complex than the metamodel of WASL Generic. For illustration purposes the corresponding Ecore file of the metamodel can be viewed instead.

### WASL Php 1.0

The metamodel of WASL Php for the scripting language PHP is organized accordingly to the object oriented language features of PHP. WASL Php offers language elements for modeling entity classes and data access objects (DAO) for the object-relational mapping (ORM) framework Doctrine. No further frameworks such as a model-view-controller (MVC) framework or a template engine are used to exemplify, that also an implementation following low-level pattern approach can be generated from WASL Generic via WASL Php.

The main container of a WASL Php model is the EClass *PhpModel*, containing a class model, a page model and a relational schema.

The metamodel elements in the EPackage *classes* are used for developing class models and follow the object-oriented paradigm of PHP under usage of the namespace feature introduced with PHP 5.3. Each *ClassModel* can contain multiple *Namespaces*, which themselves can nest further *Namespaces* thus building a namespace hierarchy. Correspondingly to the packaged associations in WASL Data, *Associations* in WASL Php are contained in one of those *Namespaces*. Similarly to Java each *Namespace* corresponds to a folder in the file system when generating source code from a WASL Php model. This is done for clarity reasons although PHP does not force it. The generic abstract metamodel element *Type* is specialized into the non-abstract subtypes *Class*, *Interface* and *PrimitiveDataType*, which can be structured by single class inheritance and multiple interface inheritance with the EReferences *superClass*, *subClasses* and *implementedInterfaces*. Additionally a *Class* can contain *Methods*, which possess information regarding *visibility*, *parameters* and by *isStatic* the distinction as a class or member method. In contrast to the UML metamodel a PHP method does not have a static return type. *Parameters* can be annotated with a *ParameterDirectionKind* to model the concepts call-by-value and call-by-reference, which both are supported by PHP. Unlike properties in UML, properties in WASL Php do not possess type information as PHP uses dynamic typing and determines the type of a property at runtime.

In the Doctrine ORM framework a relational schema is represented by a set of PHP classes, which map the schema metadata for the database tables managed by Doctrine. Both theses classes and database tables automatically can be generated by Doctrine from an YML schema file. Platform-identical metamodel elements for describing such a schema are provided with the EPackage *doctrine* and its EClass *Schema*. An YML *Schema* contains *Models* that represent database tables and *Connections* that can be associated to *Models* to specify, how Doctrine should access the database. With the option *detect_relations* Doctrine can be instructed to guess relationships between *Models* based on *Models'* column names.

As described a *Model* is implemented by a PHP entity class as well as a database table and can contain multiple *Columns* for representing value properties, multiple *Relations* to other models, a database *Connection* and multiple *Option* information for charset and collate settings. A value attribute is expressed as a single *Column* correspondingly to the element *ValueProperty* of WASL Generic. In contrast correspondingly to a *ReferenceProperty* an association between a *Model* A and a *Model* B is expressed through a *Column* owned by A plus a *Relation* owned by B referencing the *Column* in A. Each *Column* can be marked as a *primary* key, whose value can be generated automatically when inserting a new row. The column's type has to be selected from the EEnum *PrimitiveTypes*. Correspondingly to the Doctrine manual each *relation* references its local *column* as well as a key *column* of the referenced model. With the EAttribute *foreigntype* the cardinality of the relation is specified, allowing the combinations One:One, Many:One, One:Many and Many:Many. In the case of a relation with a cardinality of *n:n* a cross table *Model* has to be added, which is referenced by the two associated *Models* through the EReference *refClass*.

For each *Model* the corresponding *DoctrineClass* can be specified additionally. WASL Php offers an explicit approach of modeling and generating both the low-level database YML schema file as well as the high-level Doctrine model classes for reasons of (1) complete explication, (2) integration into the build process and (3) controlling how the generated source code should look. Another approach not chosen in WASL would be to model just the YML schema file, to generate it with the MDA generator workflow and separately to generate the Doctrine PHP classes based on the YML file with the generator script provided by Doctrine. Such an approach has the problem, that (1) in the model other PHP classes cannot reference methods of the model classes as they do not exist in the model, (2) the build process is split into two generator steps and (3) the external generator process cannot be customized.

Every Doctrine entity class owns two methods for the declaration of columns, relations, options etc. named *setTableDefinition* and *setUp*. The WASL Php metamodel provides elements for those method types as specializations of the more generic metamodel element *Method*. Additionally to the EAttributes and EReferences provided by *Method* these both methods offer the properties *columns*, *options*, *tableName* and *relations*, which reference the corresponding elements contained in the YML schema.

The retrieval of datasets from Doctrine is encapsulated by data access objects (DAO), which are marked as such by the *Class* specialization *DataAccessObject*. DAOs can contain the *DaoMethods Fetch* for retrieving one row from the database and *FetchAll* for collecting an array of multiple records. Every *DaoMethod* is annotated with a *managedDoctrineClass* to specify the returned model class and enable the automatic generation of the method's body. Additionally all model classes referenced from the managed model class have to be defined with the EReference *leftJoins* because a lazy evaluation of relations is not available when navigating through the object graph after retrieving the objects from Doctrine.

PHP used in the web context usually generates web pages through server-side PHP scripts, which are writing HTML markup code directly into a requests' HTTP response stream. In a script file PHP code can be inserted into surrounding HTML text as inline commands thus allowing a direct output of HTML markup code without using a template engine. In WASL Php such a script file is represented in a low-level approach as a *Page*, which is contained in a *Folder* that corresponds to a file system folder. Each *Page* owns an HTML *title* property and is containing HTML *Elements*. In contrast to WASL Generic in WASL Php most details of the pages' contents are modeled explicitly. Exemplary WASL Php provides typical HTML elements such as *Header*, *Paragraph* and *Anchor* tags. Additionally special metamodel elements for PHP code inlining are provided for modeling tables and HTML forms managed and filled with values by PHP code. The code areas managed by PHP are encapsulated in so called *PhpComponents* such as the *PhpListComponent*, *PhpCreateComponent* and others, which give a context to the contained PHP model elements by storing contextual information as e.g. references to DAO methods to populate managed tables and forms with values and to save submitted form values. Similarly to WASL Generic the menu structure is modeled with *MenuFolders* and *MenuEntries* separately for clarity reasons.

All PHP page scripts are maintained by a central *index.php* script, which behaves as a typical simple MVC controller and has to be called every time a PHP page script is invocated. On every invocation of the index file a page id parameter named *pid* has to be submitted, which is passed to the MVC controller method *GetTargetPath*. This method returns the path of the PHP page script associated with the delivered *pid* for inclusion through the index file. In the case of an empty *pid* the path of the default page stored in the method's EReference *indexPage* is returned.

With the described metamodel a platform-specific representation of a CRUD-oriented PHP web application can be built. All model elements can be generated from a WASL Generic model by a transformation definition written in QVT Operational.

## WASL Python 1.0

Python offers support for web engineering based on a MVC pattern with web frameworks such as Django, Zope and TurboGears, allowing the development of well-structured architectures for web applications. With WASL Python a semi-formal modeling language is provided covering Google AppEngine as the platform. Google AppEngine is based on Python and integrates the Django MVC-framework including a template engine for the frontend and Google's BigTable API for the backend implementation in the form of a schemaless object datastore.

Each *PythonModel* can contain a class model for Python's basic object-oriented language features, a template model for representing templates written in Django's template language, a model of the web application's main configuration file and a handler model for mapping HTTP requests to Python classes.

The WASL Python metamodel includes an EPackage *classes*, which correspondingly to the UML metamodel covers elements for class diagrams. Each *class* is nested in a *package*, which itself can be contained in another nesting *package*. Similarly to WASL Generic an association between classes can be stored in any package. As Python supports multiple inheritance the metamodel element *Class* can be modeled to reference multiple *superClasses*. Each class can own multiple methods and properties. The metamodel elements *Method* and *Property* do not possess a type because python uses a dynamic type system. The metamodel element *FrameworkClass* is needed for modeling framework specific classes such as the class *db.Model,* which are only intended to be referenced but not to be generated by the MDSD stack.

Entity classes for the datastore API are derived from the framework class *db.Model* and specialized in WASL Python in the form of the metamodel element *ModelClass*. A distinction is made between ordinary class *Properties* and *datastore Properties* defined for the datastore ORM framework. Both of them are generalized by the language element *AbstractProperty*. Google's BigTable API provides several types of datastore properties such as *StringProperty* for Strings and *UserProperty* for Google user accounts, which can be assigned to entity classes and partly are represented in WASL Python as *StringProperty*, *IntegerProperty*, *ListProperty* and *ReferenceProperty*. For *ListProperty* a primitive type has to be specified with the EReference *itemType* restricting, what type of elements should be stored in the list. For *ReferenceProperty* the *referencedModelClass* has to be specified, also restricting the set of possible values to elements with that M*odelClass* type.

Each web application based on Google's AppEngine is configured by a central configuration YML file called *app.yaml* represented by the language element *ApplicationConfig*, which contains the web application's name and version number and sets up the HTTP request routing handler of the web application. Corresponding to the MVC pattern Google AppEngine integrates a request controller mechanism that is configured by this application configuration file and is containing mappings from URLs to handler scripts. Each HTTP request to the web application is sequentially matched against all URL patterns provided by the configuration's *RoutingHandlerConfigEntries* until a matching entry is found. By distinguishing between script handler entries and directory handler entries different types of *RoutingHandlerConfigEntries exist.*

Script handlers execute a Python handler script for requests matching a given URL regular expression pattern. The mapping can be modeled as a *RoutingHandlerScriptConfigEntry* and defines the URL pattern to match and the *handlerScript* to be executed. In contrast to a script handler, a directory handler serves a directory directly in a static way to the client without using custom handler scripts. Particularly this is useful in the case of directories containing resources like images, CSS stylesheets and JavaScript files. With the metamodel element *RoutingDirectoryConfigEntry* such directories can be mapped to URL patterns. In the case of overlapping URL patterns the first *RoutingHandlerConfigEntry* with a matching URL pattern is chosen automatically.

Each *RoutingHandlerScriptConfigEntry* references a *HandlerScript* that is contained in the web application's *HandlerModel* and owns multiple *HandlerClasses*. Additionally to the mapping mechanism covered by the application's YML configuration each

*HandlerScript* provides its own routing mechanism by *UrlMappings*, which are mapping HTTP request URLs to *HandlerClasses* contained in the *HandlerScript*. This allows keeping the *app.yaml* file small and storing the detailed routing directives in the same handler script file as their assigned *HandlerClasses*. Correspondingly to the HTTP request methods GET and POST each *HandlerClass* can implement *HandlerMethods* named *get* and *post*, which are invocated by the handler mechanism dependent on the HTTP request type. WASL Python offers different types of *HandlerMethods* for different types of business logic. E.g. *HandlerMethodList* covers business logic for fetching objects for a *managedModelClass* from the datastore and visualizing the fetched data entries through a Django *pageTemplate*.

AppEngine offers Django's templating engine for storing the HTML code separately from the Python code and thus keeping the code clean and maintainable. Django's template language includes keywords to indicate where and how the data provided by the business logic should be displayed. With WASL Python *Templates* are stored in a central *TemplateModel* structured by *Folders*, which correspond to file system folders contained in the main template folder. Similarly to WASL Generic *MenuFolders* and *MenuEntries* are modeled explicitly for including them in the form of a consistent menu on every page. Django includes an extension mechanism with the keyword *extends*, by which page templates can extend a global frame template containing the web application's page header layer, page footer layer as well as the global menu.

Similarly to WASL Php the language elements for modeling the templates are contained in an EPackage *html* and represent HTML markup tags as well as Django directives on a low abstraction level. Each template *Element* can be nested in its *owningElement* and possess *ownedElements*, thus creating the typical DOM tree hierarchy. A HTML anchor is represented by the *A* element, which owns a reference to one of the handler classes' URL mappings through the EReference *hrefUrl*. In the generated implementation the value of the EAttribute *url* owned by the referenced *UrlMapping* is used as the generated anchor's *href* value. With a given set of *DjangoComponents* predefined functionality can be added to the template model, which is expanded by the model-to-text generator to more complex Django directives.

Structurally a WASL Python model varies from a Python web application by the fact that packages are modeled differently than the implementation due to simplification reasons. Generally in WASL Python packaged classes are modeled following the structure *superpackage/package/class*. In contrast Python expects such a class to be placed in a module file named like the *package*, located in a package folder named like the *superpackage* that additionally has to contain a file named *__init__.py* to indicate, that the folder should be treated as a package directory. This introduces the problem that potentially multiple classes have to be generated into one single module file, inducing implications on the transformation definition of the code generator.

### WASL JavaEE 1.0

The development of web applications for Java runtime environments is supported by several specifications and frameworks, such as Enterprise JavaBeans (EJB), Hibernate, Servlets, JavaServer Pages (JSP), JavaServer Faces (JSF) and Spring. For WASL JavaEE the two frameworks Hibernate and JSF were chosen because both are feature stable and widely used in the Java web development community.

The core metamodel of WASL JavaEE is constituted by elements for modeling UML-like class diagrams. A *JavaEEModel* contains one such *ClassModel* for modeling classes including MVC model classes and one *JsfModel* covering MVC views and MVC controllers. Similarly to UML primitive data types are not provided by the metamodel but have to be added by the modeler as instances of *PrimitiveDataType* to each WASL Generic model in a prior language configuration step. By the EReference *primitiveDataTypes* each *ClassModel* offers one central location for gathering all those instances of *PrimitiveDataType*. A class model can contain nested *Packages* forming a package hierarchy. *Classes* and *Interfaces* are generalized to *Classifiers* and can be placed in *Packages* together with instances of the type *FrameworkClass*, which represent classes provided by the runtime environment and which should not be created by the code generator. A class can possess one *superClass*, multiple *subClasses* and can implement multiple *Interfaces*. *Classes* as well as *Interfaces* can own multiple *Methods* and *Properties* as well as *Annotations* accordingly to the Java Specification Request (JSR) 175. Correspondingly to UML *Associations* provide bidirectional mappings between *Properties*. A *Method* owns a preset *visibility*, *parameters*, *annotations* and can be marked as being *static*. With the EAttribute *isDerived* it can be expressed that a method consists only of a method call to another method and thus is only a trivial proxy without further program logic. As a specific *Method* type the metamodel element *Constructor* classifies Java constructor methods with the restriction that the constructor's name has to be the same as the name of the containing class. Theoretically it could be determined by name comparison if a method is a constructor, but by typing them explicitly the selection mechanism is simplified and constructor-specific EAttributes and EReferences can be added in future versions of WASL JavaEE. A *Property* can be marked as being *readable* and *writeable* for keeping the model clean from accessors following the C# property concept and letting the code generator fabricate those trivial methods. With the EAttribute *isInitialzed* it can be determined, if the declaration of a *Property's* variable should be followed by an initialization storing a default value in the variable.

The Hibernate object-relational mapper (ORM) framework uses plain-old java objects (POJO) and offers adding meta information to those POJOs through the Java annotation API. In the context of Enterprise JavaBeans (EJB) a persistent POJO is also called *entity bean*. In WASL Generic a *PersistentPojo* is derived from the metamodel element *Class*. It owns *PersistentPojoProperties* which can be marked as keys with the EAttribute *isKey*. For each POJO one data access object (DAO) class is provided that encapsulates all method calls to the Hibernate API. A DAO class is modeled by the metamodel element *DataAccessObject*, which possesses *DAOMethods* for load, delete, save, list and search operations represented by the metamodel elements *DAOLoad*, *DAODelete* and so forth. All of these methods reference the method *getCurrentSessionMethod*, which returns the current Hibernate database session. With WASL JavaEE DAOs are implemented accordingly to the singleton pattern. Usually singleton classes possess (1) a private constructor, (2) a private static variable for referencing the singleton instance and (3) a public static method for initializing this variable with a singleton instance and returning a reference of this instance to method callers. The first two requirements can be modeled with standard language elements of WASL JavaEE, whereas for the third aspect the metamodel element *GetInstance* is provided as a specialization of *Method*, which indicates the required logic. Thereby WASL Generic follows the approach of classifying methods and their bodies by metamodel elements instead of representing program logic in the model by cumbersome program logic diagrams.

The implementation of the web application's presentation layer is based on the JSF specification and modeled in a *JsfModel*, which contains a model of the *faces-config.xml* descriptor file, a view model, a representation of the menu structure and a class model covering MVC controller classes.

The JSF faces-config descriptor file is a XML document that configures the central aspects of a JSF application such as *ManagedBeans* and *NavigationRules*. A faces-config *ManagedBean* entry is offered under its *name* to JSF views, references a *managed (MVC-model) class*, has a *scope* w.r.t. to the lifespan of the managed bean instance and can maintain managed properties in the case of property values which should be set by the JSF controller. Also *FacesConfig* covers *NavigationRules* containing *NavigationCases* to describe, which *View* should be displayed by the JSF controller depending on the outcome of preceding user requests and methods e.g. for validating user inputs.

Similarly to WASL Generic the presentation layer is modeled in WASL JavaEE as views in the form of different *viewtypes*. Typical CRUD-operations are covered by the viewtypes *EditView*, *ListView*, *DataSetView*, *CreateView* and *DeleteView* derived from the generalized language element *View*. In the case of views containing forms and tables a set of *ViewProperties* can be composed and added to such a *View*, enabling the selection of a subset of relevant persistent POJO properties to be displayed as columns or input elements in those tables and forms. The approach is modular so that more specific view types can be added in a language configuration step. *Views* are structured in *ViewGroups* and should be grouped w.r.t. general structure of the frontend as presented to the user. With JSF MVC controller methods are registered to *View* input elements such as buttons and links as action listeners following an observer design pattern. If the user clicks a button a preset *ControllerMethod* of a *Controller* class is called. In case of the provided *controllermethods CLoad*, *CDelete*, *CSave* and *CReset* covering basic CRUD functionality the DAO methods of the backend are utilized for retrieving and storing entities. The localization of the corresponding backend DAO is done with a preset reference to the DAO stored in the *CPrimaryDAO* property possessed by each controller class. Also every controller can possess additional properties typed by the metamodel element *CManagedPersistencePojo*, in which the controller can store POJOs for multi-page sessions e.g. in the case of page-spanning wizards.

With the described metamodel a platform-specific representation of a CRUD-oriented JavaEE web application can be built. All model elements can be generated from a WASL Generic model by a transformation definition written in QVT Operational.

## Bibliography

[1] "MDA Guide Version 1.0.1," OMG, 2003.

[2] Anna Gerber and Kerry Raymond, "MOF to EMF: There and Back Again," in *OOPSLA Workshop on Eclipse Technology eXchange (OOPSLA2003)*, Anaheim, California, 2003, pp. 60-64.

## Changelog

| | |
|---|---|
| Version 1.0 | Initial version |