GLOBAL ARCHITECTURE-BASED SIMULATION OBJECT MANAGEMENT WITH INTEGRATION OF LOCAL TOOLCHAINS

Christian Gnandt
TESIS DYNAware GmbH, Germany

Dr. Clemens Hepperle
TESIS DYNAware GmbH, Germany

1. Introduction

Today's cars and those of the future contain a large number of highly interconnected electronic vehicle functions. The rapid progress in assisted and autonomous driving and the demand for increased energy efficiency make the connection between the electronic control units even more complex. They are developed and tested in several steps: Model-in-the-Loop (MiL), Software-in-the-Loop (SiL) and Hardware-in-the-Loop (HiL) via a virtual vehicle approach. One such example is a Multi-ECU HiL-system consisting of a damper control unit, a Haldex-clutch control unit and a ESC control unit for virtual characteristic rating of vehicle dynamics control systems [1].

With the techniques of dynamic simulation, the increasing variant diversity and complexity can be managed. Within a development process in which the product maturity evolves and the development scope changes over the time, different levels of detail concerning simulation models are necessary. Various departments need a complete virtual vehicle model or model components depending on their respective development progress. The separate development of individual components and their related models, as well as their subsequent integration, has become unthinkable. Instead, an extensive collaboration throughout the complete development cycle is necessary, including the data exchange with technology partners.

This remains a challenging issue, in particular concerning the use of a common knowledge base and data base for efficient cross-department and cross-domain collaborations [2]. Thereby, the data of an individual development department coming from different and mostly individual users' simulation systems, as well as toolchains and working processes all have to be integrated. Available systems used for collaborative engineering have to provide *"…seamless connections and data exchanges with each other…"* on the basis of a standardized user interface [3]. Consequently, users are enabled to systematically exchange the increasing amount of data and work together on their simulation models in a synchronized way.

In many fields that are dependent on the verification of functional safety, legislative requirements need to be fulfilled concerning the approval of work products. In these cases, all data including model data and simulation results related to the simulation models have to be managed and made traceable. According to ISO 26262, a demanded standard in development of functional safety relevant systems of road vehicles, one *"…objective is to ensure that the work products, and the principles and general conditions of their creation, can be uniquely identified and reproduced in a controlled manner at any time"* [4].

A powerful instrument for providing data traceability with respect to recovering certain states of work products is a version control system. Already introduced some decades ago by Rochkind [5], version control systems have evolved and are widely accepted and used in the industry [6]. From a data traceability point of view, these systems are quite mature and are easily embeddable within currently used working environments.

Still, in each simulation environment data has its own meaning, holds additional properties (meta-information), and contains information on how it is related to other data. Until now in the field of functional development simulation, data is often only stored, versioned and managed in a file-based manner within a single simulation system. The necessity of introducing a comprehensive data management system has been recognized, but the implementation is challenging. Furthermore, an approach should be given to capture and version this kind of information, as well as be able to reproduce the meaning of a certain work product. Consequently, there is also great potential in being able to enhance how one searches, finds and easily accesses relevant existing, reusable data and knowledge.

This again means that simulation components and their related data are no longer developed redundantly. Similar data can easily be identified and respective abstracted architectures can be derived. These can consequently be used as a basis for further work products, resulting in reduced cost and increased productivity of development capabilities. Efficient exchange of available information and solutions (e.g. available parameterized engine model) between departments and technology partners helps to speed up the overall development.

Furthermore, the quality of the respective simulation artifacts can be increased if these are provided by the specialists of the department responsible for the component. In this context it is important to provide a controlled way of accessing data which is often restricted to a certain group of people. Besides controlling who is allowed to see which data, it is also important to regulate what users are able to do with this data. In this context, different procedures regarding permission handling have been discussed in the past for collaborative engineering [7, 8].

Against this background this paper presents a data management solution which represents and stores data as objects. The objects' meaning and the relationships between objects are described via a meta-model definition. Because of its abstract and high-level nature the so-called *simulation object manager* provides a cross-department and cross-domain collaboration of the data that is embedded in objects.

Although the creation of an overall object-oriented data management platform for model-based development faces the above mentioned challenges, it nevertheless provides several distinct advantages for individual users, as well as for the organizations where it is introduced. For example, context specific data settings can be configured for both individual users working in local toolchains, as well as for interconnected teams on a common data base. Thereby users are able to maintain their local toolchains and set up their own simulation environment while still contributing to a comprehensive data base.

In Section 2 of this article, the technical concept for the object-oriented and architecture-based data management system is described and its implementation as a client-server system is explained. Section 3 provides details concerning the connection of the object management and local toolchains. In both sections the developed solution is validated using common use-cases. In Section 4, conclusions on architecture-based simulation object management are drawn.

2. Architecture-based simulation object management

Objects

Our answer to the challenges identified above starts by establishing a generic approach. We incorporate every piece of data into an object. Each individual object bundles the data itself and all its related meta-information. For each data type, a different object type definition is used. The approach used for defining object types and objects instances is closely related to the UML class and object definition (see [9, 10] for details). Figure 1 illustrates the common object type definition and its instances as data objects.

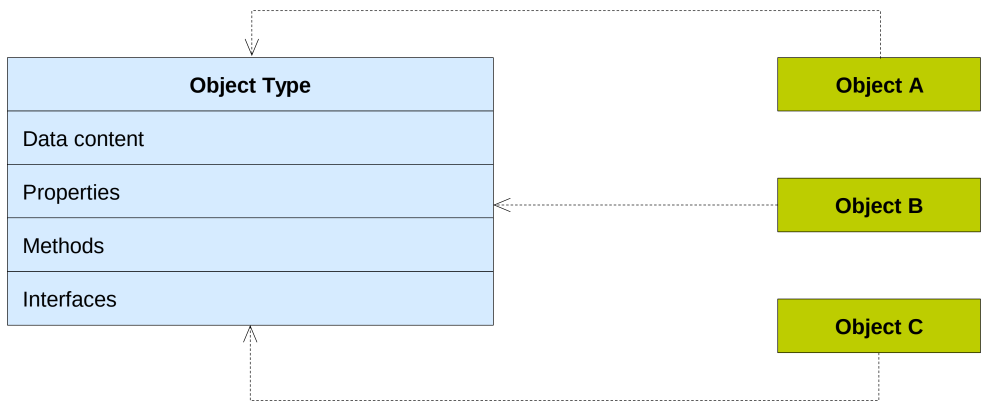


Figure 1: Generic object type definition that encapsulates data and meta-information to ensure sustainability

Each object type defines the type and structure of the data it can contain. If the data is available as binary information, it is also possible to simply add the data as file attachment to an object.

Properties provide additional meta-information for the object. Within the object management system, two types of properties are available. System properties are provided by the system and deliver information that is related to the life-cycle of the object, e.g. author or last modified information. User-defined properties can be configured by the user for each object type. The structure of a property can also be specified. This provides a convenient way for an organization to provide a common framework for users to use within a collaborative working environment. Objects can also provide certain functionalities via the use of methods. Common functionality includes e.g. the modification of object data and properties. Dependent on the object type, special methods can be specified, e.g. export of data to a local toolchain or generation of an executable for a model component. The interface specification is necessary when objects interact with each other.

Figure 2 shows a typical example for the definition of an object type. The object *DataValueSet* is common in the field of model based development. It specifies the data used to parameterize a model component.

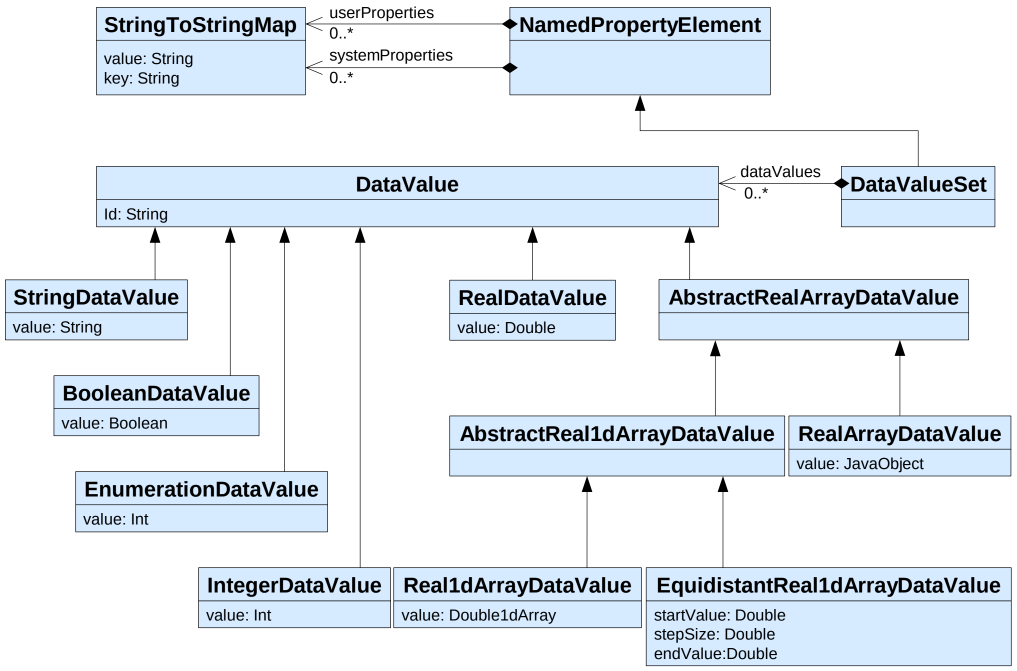
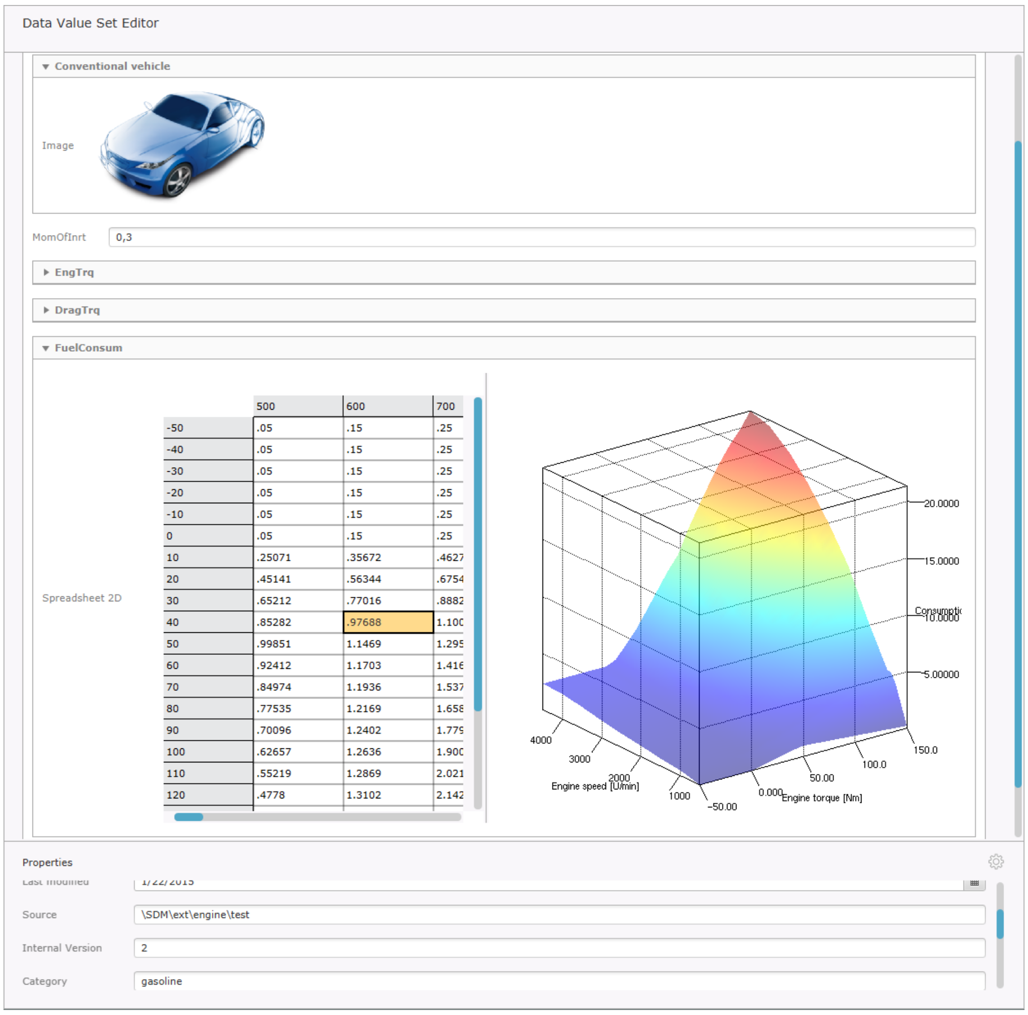


Figure 2: Object type definition of a DataValueSet

When the user creates or modifies a Data Value Set, the data for the individual data values that are part of the set are specified. The user also provides additional information via user-defined properties. Figure 3 shows the corresponding user interface. It is possible to define the individual data values (e.g. drag torque or fuel consumption) for the complete data set used to parameterize the fuel consumption model. Changes can be directly inspected via the appropriate visualization. The bottom of the screenshot shows the different (user-defined) object properties. They classify the Data Value Sets for a specific usage (e.g. fuel consumption model data for gasoline vehicle) and help users to find the proper data for their application.



Data Value Set Editor

Conventional vehicle

Image

MomOfInrt 0,3

EngTrq

DragTrq

FuelConsum

Spreadsheet 2D

	500	600	700
-50	.05	.15	.25
-40	.05	.15	.25
-30	.05	.15	.25
-20	.05	.15	.25
-10	.05	.15	.25
0	.05	.15	.25
10	.25071	.35672	.4627
20	.45141	.56344	.6754
30	.65212	.77016	.8882
40	.85282	.97688	1.100
50	.99851	1.1469	1.295
60	.92412	1.1703	1.416
70	.84974	1.1936	1.537
80	.77535	1.2169	1.658
90	.70096	1.2402	1.779
100	.62657	1.2636	1.900
110	.55219	1.2869	2.021
120	.4778	1.3102	2.142

Properties

Last modified	1/22/2015
Source	\SDM\ext\engine\test
Internal Version	2
Category	gasoline

Figure 3: User interface for the Data Value Set Editor

Meta-model approach

Looking at real simulation environments, it becomes clear that different Data Value Sets may refer to the same desired parameter types. In this context, it is not reasonable to start defining Data Value Sets from scratch. They should be built using a common parameter base. Instead of creating various Data Value Sets with different parameter names and units, for example, it is important to provide a standardized base set for related Data Value Sets from which different sets can be derived. This helps to organize collaborative engineering from a data point of view and allows for increased traceability of the simulation elements. At the same time, it prevents unnecessary redundancy in data artifacts, and in error-prone work e.g. defining Data Value Sets.

Thereby the given architecture-based approach provides a standardized foundation for the instantiation of Data Value Sets. Within that approach, the “Parameter Type Set” object type is introduced. Before going further into the details of this object type, it is important to note that this general approach of using an architecture-based Data Value Set definition is not restricted to Parameter Type Set. It can also be applied to various types of data, such as different model components derived from a common model architecture.

The concept of defining an architecture-based Data Value Set is illustrated in Figure 4. Here, one can see the different objects, as well as their relevant class diagrams describing the object types “Data Value Set” and “Parameter Type Set”. The figure shows how a concrete Data Value Set for the fuel consumption of a specific conventional passenger car model is based on the Parameter Type Set for fuel consumption. In addition, it is also implemented based on the class diagram of the Data Value Set already shown in Figure 2.

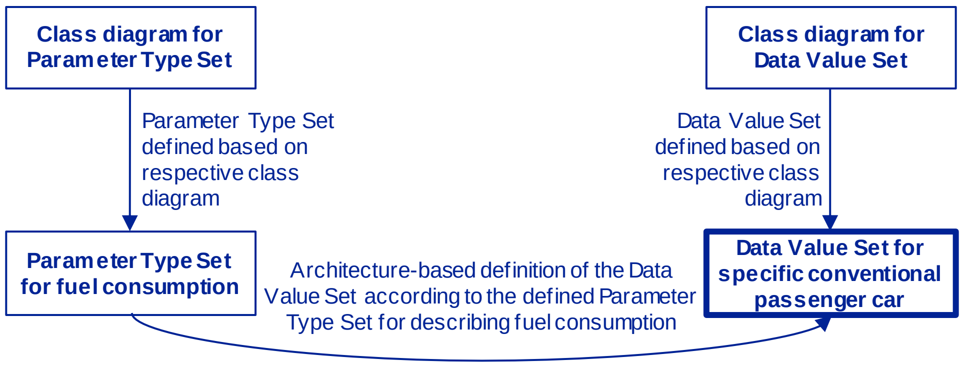

Class diagram for Parameter Type Set
Parameter Type Set defined based on respective class diagram
Class diagram for Data Value Set
Data Value Set defined based on respective class diagram
Parameter Type Set for fuel consumption
Architecture-based definition of the Data Value Set according to the defined Parameter Type Set for describing fuel consumption
Data Value Set for specific conventional passenger car

Figure 4: Concept of architecture-based data value set definition

As a Parameter Type Set is the basis for a Data Value Set, it holds information that helps the simulation engineer fill in the respective values leading to a concrete Data Value Set. The elements found in a Parameter Type Set are shown in Figure 5. Thereby the same data types as found in the class diagram for Data Value Sets (such as String, Boolean, etc.) are present so that a corresponding Data Value Set can be defined. Within the Parameter Type Set, the units of the Data Value Set can also be defined. Furthermore, the Parameter Type Set provides a method of validating if an entry destined for the Data Value Set is reasonable or not. This is also shown in the class diagram, e.g. entries can be verified against the optional minValue and maxValue specified for a given RealParameterType. Furthermore, it is also possible to define a default value for the different data types. As the Parameter Type Set is itself an object, it also consists of system and user properties.

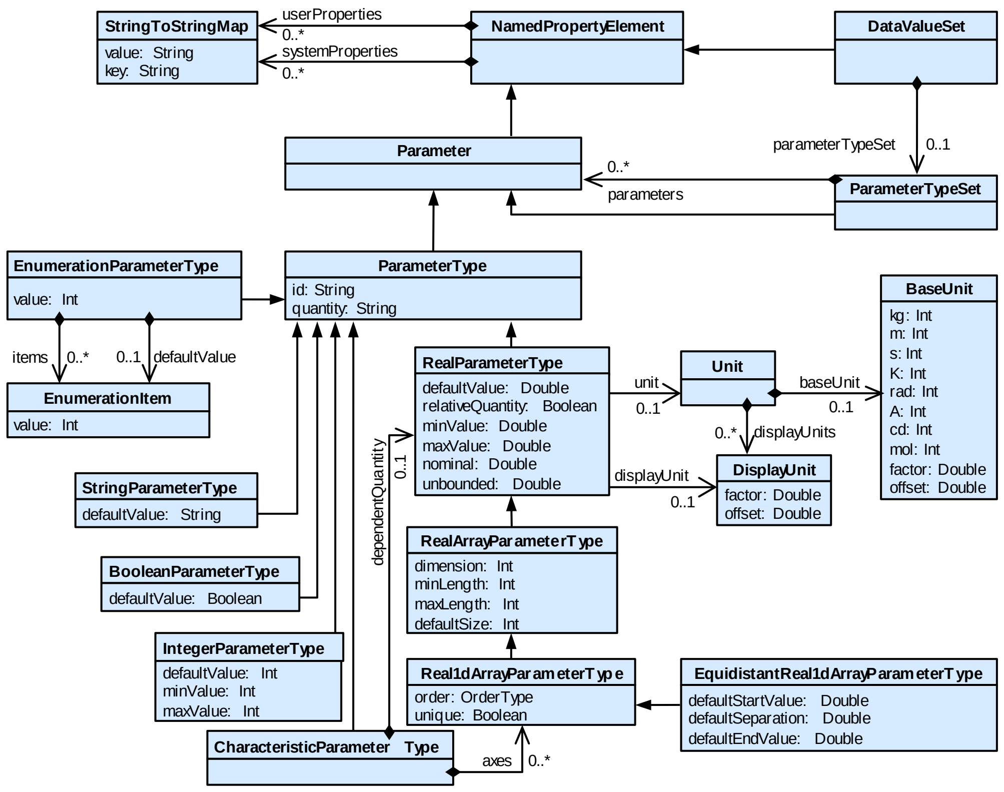
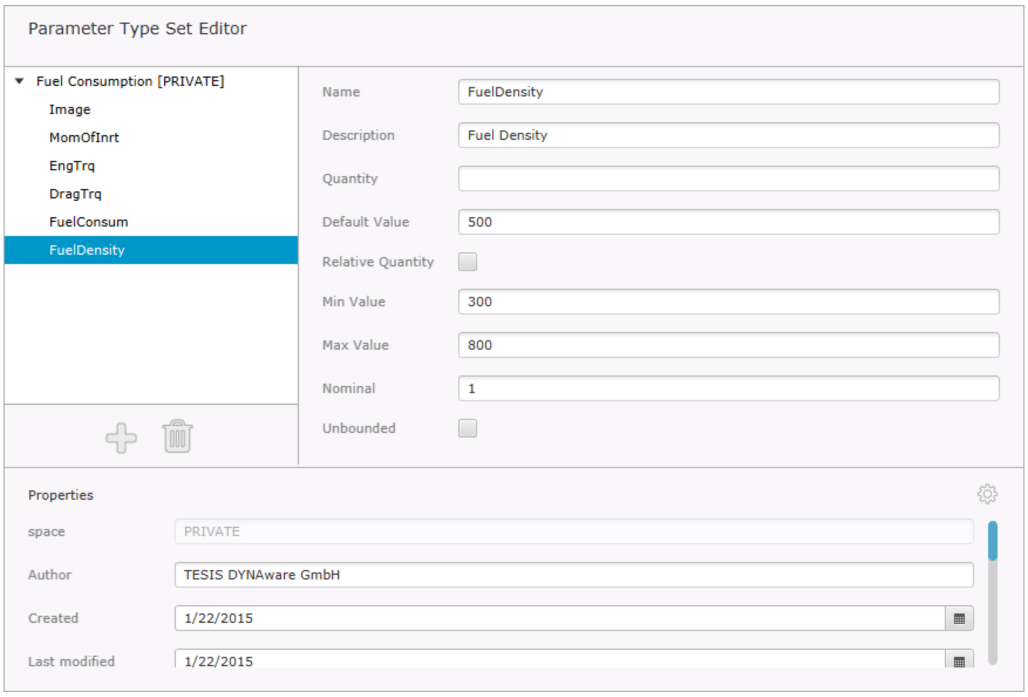


Figure 5: Definition of object type ParameterTypeSet

The user interface provides a Parameter Type Set editor to create and modify a respective Parameter Type Set according to the shown object type definition. In Figure 6, a screenshot illustrates the definition of a parameter 'Fuel Density' with a default value of 500, a Min Value of 300 and a Max Value of 800. At the bottom of the screenshot, different object properties of the Parameter Type Set are shown.

The common base provided by the use of a Parameter Type Set as the basis for a Data Value Set supports a structured procedure in defining and parameterizing simulation models. This helps in working efficiently with specific data sets while at the same time enhancing the transparency of the data manipulations.


Parameter Type Set Editor

Fuel Consumption [PRIVATE]
- Image
- MomOfInrt
- EngTrq
- DragTrq
- FuelConsum
- FuelDensity

Name	FuelDensity
Description	Fuel Density
Quantity	
Default Value	500
Relative Quantity	☐
Min Value	300
Max Value	800
Nominal	1
Unbounded	☐

Properties

space	PRIVATE
Author	TESIS DYNAware GmbH
Created	1/22/2015
Last modified	1/22/2015

Figure 6: User interface for the Parameter Type Set Editor

Collaborative Engineering

Objects evolve within simulation environments over time due to changing surrounding conditions and the accumulation of more detailed information throughout the development process. Therefore, methods for reconstructing the knowledge base to accommodate certain decisions have to be provided.

Thereby, the first such method is a common “data storage” which holds the objects that are shared within a collaborative engineering environment. This is realized by a Public Space, which can be seen as a large “repository”. Users can see objects in the Public Space for which they have access according to their permission management settings, which is discussed later as an important measure for supporting object management in collaborative engineering.

If an object needs to be modified by users, they must check out the object to their Private Space. Similarly, the creation of a new object also takes place in the Private Space. If an object is only used within the respective simulation context and is not modified, the user can apply the object directly from the Public Space. After an object is created and/or modified, it can be published to the Public Space.

From a technical implementation point of view, working with Public and Private Spaces as well as using objects within local systems is enabled by a client-server architecture. We follow a modular approach with state-of-the-art software technology and well documented APIs. Different database systems are supported for a comprehensive handling of objects and their relations.

Versioning

An important issue is the traceability of objects when they are created, modified, and used. The objects' history can be reconstructed and reapplied using the version control system, which is able to correlate objects' data and revision history between the Public and the Private Space. The version control system is comparable to existing version control systems like IBM Clearcase [11] or Apache Subversion (SVN) [12] and supports the following version control actions: branching, check-out, diff, freeze, lock, merge, modify, publish, release, revert, save, unfreeze, update. The following paragraphs explain the core version control elements (version, branch, release, and label) in detail.

A version refers to a unique state of a simulation object. It contains:

- Data content specific to the object type
- Properties
- Methods
- Interfaces

For every version in the Public Space the user can add a comment which refers to the specific version. Each modification of an object in the Public Space leads to a new version. The "latest" version of an object always refers to the most up-to-date version of the object in a specific branch.

A branch represents the parallel development of one simulation object. Branches are generated automatically when two or more users are working on one object at the same time. Additionally, branches can also be generated manually by the user.

A release connects different versions of a collection of simulation objects to one logical element within the Public Space. There can be a multitude of objects with different versions in the Public Space. For a given project, the user(s) will need a specific set of objects. Releases not only help the users keep an overview over the individual versions, but also over specific content that belongs together. The contents of releases have to be consistent: all its objects and the objects that they reference internally are part of the release.

Labels help the users to group versions of objects. Labels can be defined for a specific version of an object. The same label can only be used once within the version tree of one object. Labels can be used to define the content of releases.

Unlike many existing file-based version control systems, the suggested version control is also capable of versioning the objects' meta-data, which leads to multiple advantages in data management:

- Searching for, or filtering, objects related to a specific object version according to their meta-data (properties) is possible.
- Being able to work with a plain object/data structure and at the same time providing the possibility of flexibly grouping the objects according to their respective simulation context.

As simulation environments deal with a large amount of data and files, users have to be helped in defining their context-specific working environment and its associated set of objects. Thus, so-called Configuration Rule Sets (CRSs) are provided. They help form the basis of what Public Space objects are shown on the user's screen. The CRSs filter objects and only show those matching the filter criteria of the CRSs. Thereby a CRS is defined using structured query language (SQL) statements and thus provides powerful search options. Consequently, the version used for creating simulation results is clearly defined. This enables transparency in defining collaborative working scenarios, as well as traceability when handling data.

Figure 7 illustrates the concept of using Configuration Rule Sets in combination with the Public Space as a large repository, and the Private Space as an area for creating and modifying objects. For object O1 and O3, three versions already exist, while for O2, only two versions have been documented within the Public Space. Within his/her simulation context, the User A needs to always work with the latest version of the objects. Therefore he/she uses a CRS that filters the repository and shows only the latest version of objects. User B, on the other hand, has a different goal within his/her simulation context and thus needs different object versions than those of User A. For this reason, he/she has assigned the "Label X" to these specific object versions (O1v2, O2v1, O3v3). Within the CRS Y it is defined that all object versions being labelled with "Label X" should be filtered from Public Space.

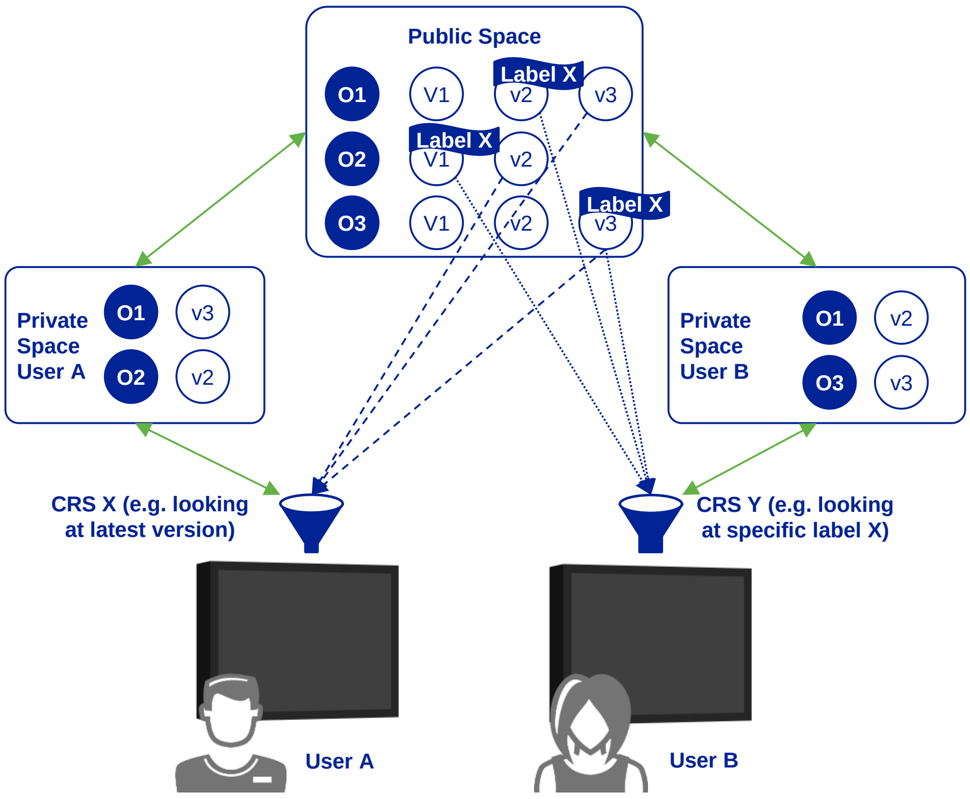


Figure 7: Working with Configuration Rule Sets and Spaces

Based on the view of the Public Space resulting from CRS X, User A can, for example, check out O1v3 for modification to his/her Private Space. If he/she would check in a modified version of these objects, they would get a new version number. Based on the view of the Public Space resulting from CRS Y, User B is similarly able to check out, for example, O1v2 for modification in his/her Private Space. Checking the modified version in again could be realized, for example, by starting a new branch of object 1.

If the simulation context of User B changes and, for example, becomes the same as that of User A, he/she might simply change his/her applied CRS to CRS X. This way, he/she would also see the latest version of each object. Thus, sharing a CRS with another user provides a common object view, and therefore a common view of the database, for the respective users.

Authentication and authorization

The act of sharing a common data platform allows for better efficiency and traceability when handling data. But if everybody has full access to objects and is able to use them however they want, it also means that information is spread throughout a company, and even over the company’s borders. To control the distribution of data, two elements that build on each other are considered: authentication and authorization. Accordingly, implementing an appropriate permission management system is motivated by different factors.

In particular authentication allows the tracking of which simulation data is used and processed by which user. On the one hand, this enhances knowledge management, as other users know whom to ask for more information in case something needs to be clarified. On the other hand, in correspondence with the documentation of versioning information, it becomes clear who created certain (intermediate) results and how they were created. In the case where errors occur, the company can react quickly and implement bug-fixing measures, since the user responsible for this issue is known.

Furthermore, a permission-based access to data enhances intellectual property protection, which also concerns cross-company collaboration. If the access to objects is controlled, confidential information which should not be available to all users can be protected. One example is the access to objects depending on their maturity level (e.g. only accessible for users within preliminary development departments and not series development). From a performance point of view, the creation of a well thought out authorization scheme for certain service methods, which are to be applied to the objects, can increase efficiency in daily work. As certain tasks should only be fulfilled by certain roles within a company, the access to role-dependent activities can be limited. One such example is the simulation-based execution of tests by the “testing expert” role. In this scenario, the testing expert should not be able to apply service methods that can change the simulation model, since this is only allowed to be done by the “simulation model engineer” role in earlier stages of the test cycle.

For providing both control over objects and service methods dealing with these objects, an approach combining Access Control Lists (ACL) for object permission handling and a Role Based Access Control (RBAC) for service methods is used (see Figure 8). On the object permission side a user, creating an object can decide which other users get the permission to read/modify and publish the respective object (e.g. publish permission concerning specific “Data Value Set for fuel consumption of a specific conventional passenger car”). Besides individual users, user groups consisting of multiple users are considered for permission assignment.

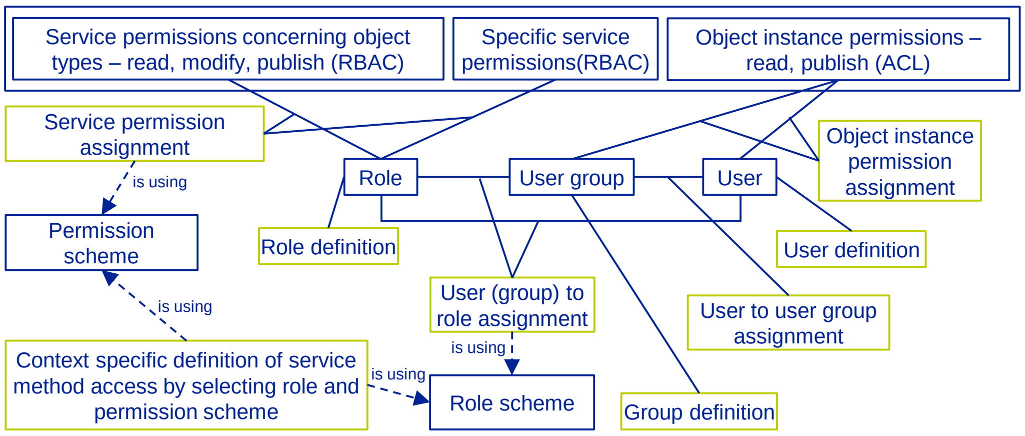

Service permissions concerning object types – read, modify, publish (RBAC)
Specific service permissions(RBAC)
Object instance permissions – read, publish (ACL)
Service permission assignment
Role
User group
User
Object instance permission assignment
is using
Permission scheme
Role definition
User definition
User (group) to role assignment
User to user group assignment
is using
is using
Context specific definition of service method access by selecting role and permission scheme
is using
Role scheme
Group definition

Figure 8: *Permission management for architecture based object management*

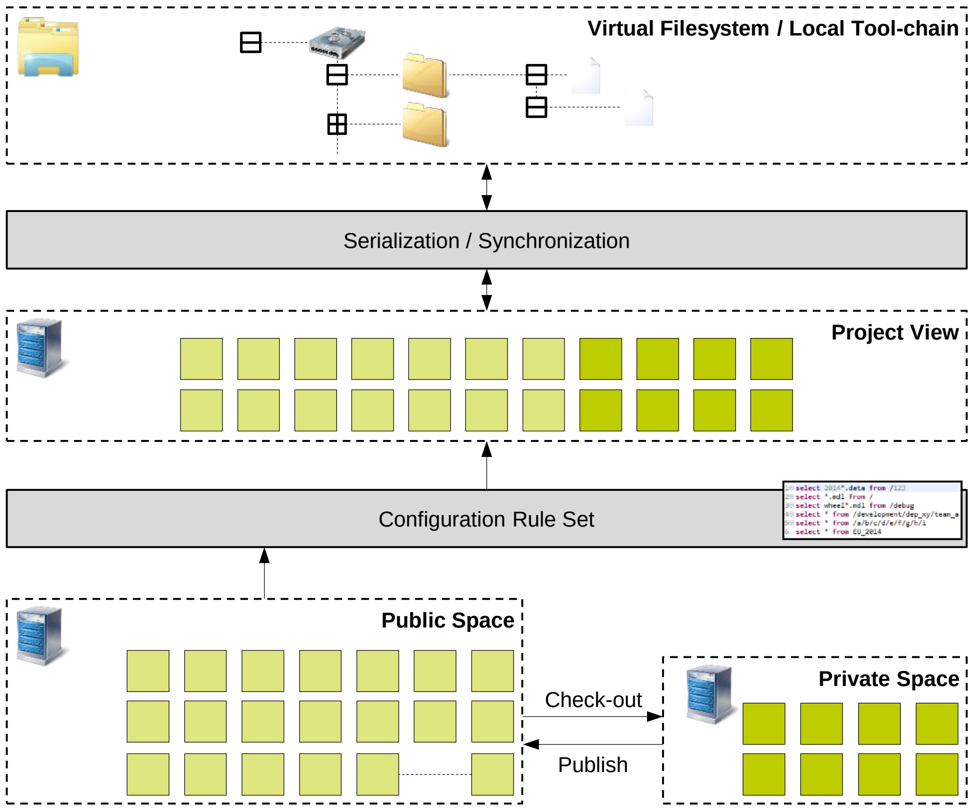
Service methods are protected by service permissions. Thereby each role is given a certain specific set of service permissions. This is due to the fact that role definitions can be systematically defined. In other words, each role is associated with certain privileges and duties, which correlate to a limited set of actions that can be performed using that role. As users act within certain roles, the users can be assigned roles depending on the respective simulation context. The service methods are divided into two general categories: the object type service permissions and further specific service permissions.

- Object type permissions control if a role is allowed to apply methods to read, modify, or publish specific objects classified as a certain object type (e.g. modify all objects of object type "Data Value Set").
- Specific service permissions control access to further services other than object type permissions, e.g. if a role is allowed to execute a certain software.

Assigning users to roles within role schemes and of service permissions to roles in permission schemes increases the flexibility and reusability in realizing context specific permission scenarios. This increases the flexibility and reusability in realizing context specific permission scenarios. Consequently, the decision whether a user is allowed to work with a certain object or not is based on the combination of specific object instances and service permissions present in a given scenario. Thereby the approach of using objects and according versioning mechanisms provides opportunities of permission handling also addressing specific object versions with respect to their correspondent meta-information.

3. Connection to local simulation systems

Virtual file system

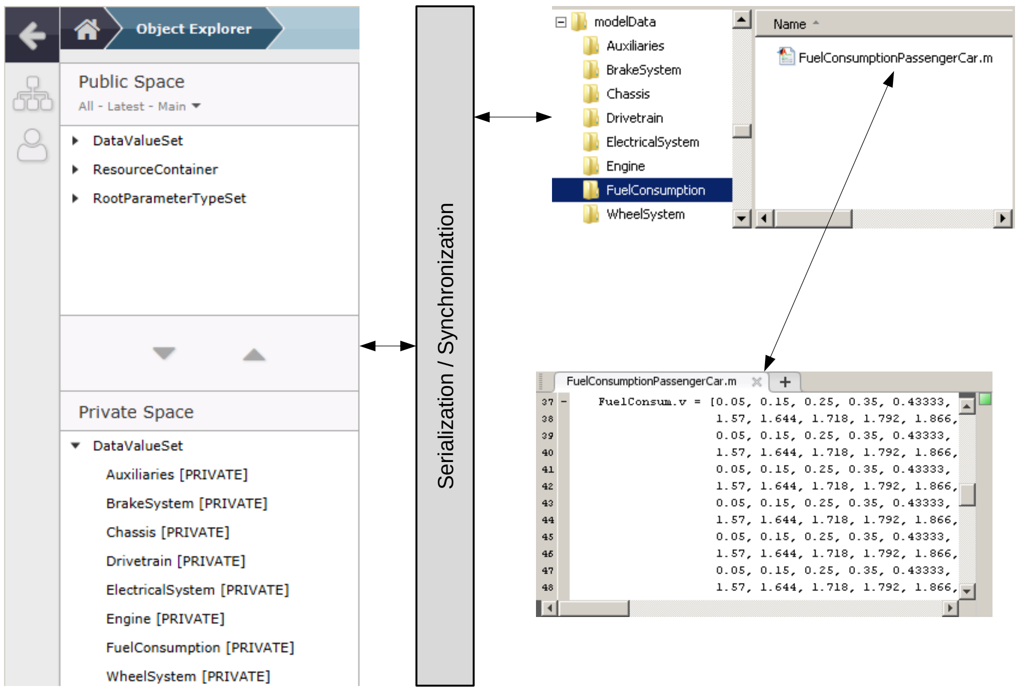


```
select 2014*.data from /123
select *.mdl from /
select wheel*.mdl from /debug
select * from /development/dep_xy/team_a
select * from /a/b/c/d/e/f/g/h/i
select * from EU_2014
```

Figure 9: Bidirectional connection between virtual file systems and object management system

The approach described in Section 2 is used to administer and share data in a collaborative working environment. For example, consider the scenario where the user wants to use the data located in the data management system within a simulation in his/her local toolchain. A common method of moving data to and from a simulation system is to use an import/export mechanism. On the one hand, this is very time consuming due to its manual nature and to the fact that the data usually has to be transformed. On the other hand, the required traceability is lost. Therefore, we developed a direct connection between the object management system located in a database on the server and the toolchains located on local machines. Because most simulation systems are organized using files on the file system, our approach synchronizes the data content of the objects with files and folders in a virtual file system. Figure 9 shows the main functional concept.

Initially, the user configures the objects together with the desired version via the Configuration Rule Set. The project view shows all objects together including their meta-data. The individual objects are stored in a virtual file system using serialization. The connection is bi-directional and configurable. The method used to serialize each object, or object type, is specified in the corresponding meta-model. Note that there is no pre-defined structure enforced by the object management system, so it is possible to define your own structure in the form of a file system containing files and folders. Figure 10 shows an example of a project view together with the corresponding serialization with the virtual file system.



```
FuelConsum.v = [0.05, 0.15, 0.25, 0.35, 0.43333,
                1.57, 1.644, 1.718, 1.792, 1.866,
                0.05, 0.15, 0.25, 0.35, 0.43333,
                1.57, 1.644, 1.718, 1.792, 1.866,
                0.05, 0.15, 0.25, 0.35, 0.43333,
                1.57, 1.644, 1.718, 1.792, 1.866,
                0.05, 0.15, 0.25, 0.35, 0.43333,
                1.57, 1.644, 1.718, 1.792, 1.866,
                0.05, 0.15, 0.25, 0.35, 0.43333,
                1.57, 1.644, 1.718, 1.792, 1.866,
                0.05, 0.15, 0.25, 0.35, 0.43333,
                1.57, 1.644, 1.718, 1.792, 1.866,
```

Figure 10: Example of the file-based serialization of a complete MATLAB simulation environment

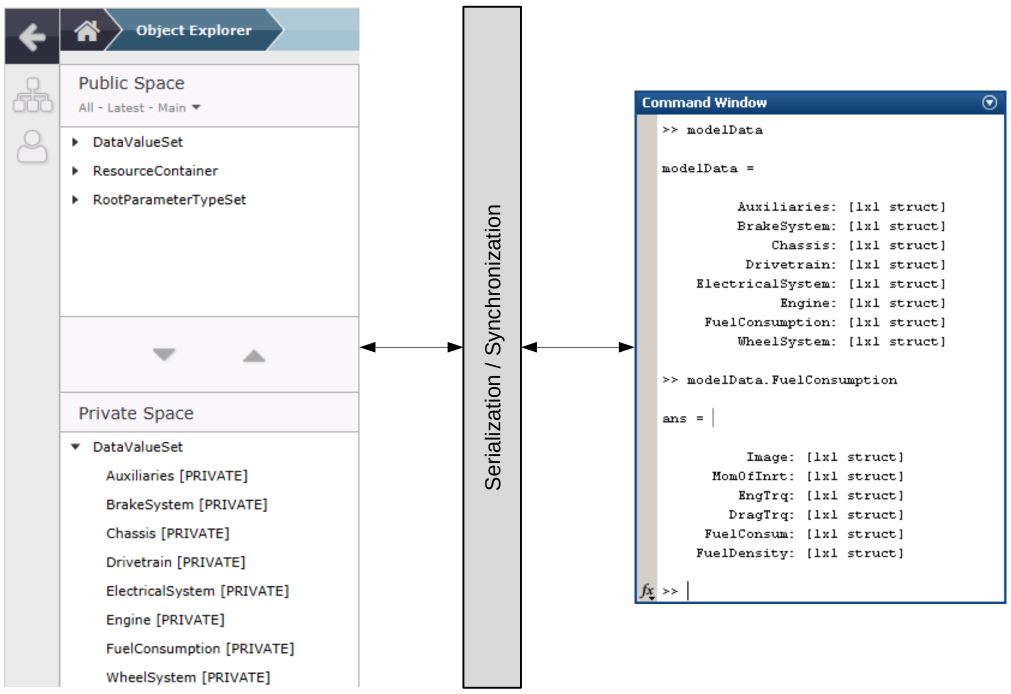
In this example, the project view shows different Data Value Sets that have been selected by the user. They represent a virtual vehicle approach to functional development. The local toolchain is executed in MATLAB. Therefore, the data content for all objects is serialized as MATLAB script files for parameterization in a specific folder structure. Note that the structure, as well as the serialization itself, is toolchain dependent and is therefore also adaptable to other file system conventions. It is hence possible to rely on the same data base to generate data specifically formatted for different local toolchains.

This approach allows the user to work with both the data in the object management system, as well as the one in his/her local toolchain. If the user modifies a script file, the object management recognizes the change and stores the modification as a new version in the user's Private Space. If applicable, the user can add meta-data to the new version and publish it so that it can be shared with other users. If he/she changes the objects or the object's version he/she is working with via the object management this also affects the files for his/her local toolchain.

The virtual file system is created by using the well-known *Web Distributed Authoring and Versioning (WebDAV)* technology. It is a set of extensions to the HTTP protocol that provide the capability of managing files on remote web servers in a collaborative manner. The actual specification can be found here [13], for further details we recommend [14]. In our example, we mount the virtual file system as a common web folder in Windows Explorer, which allows the user to have direct access to it on his/her local machine. Since the user has to authenticate him or herself when connecting to WebDAV, the virtual file system holds the same authorization settings as the objects within the object management system. Therefore, it is possible to individually configure the resulting file system based on the user's requirements. E.g. it is possible to restrict the attributes of certain files so that they are read-only. This will guarantee that the user does not accidently modify parameters for a predefined simulation test environment when executing functional safety tests.

Common API

Another way of connecting the data management system to the local toolchains can be realized through the use of a common API. It is possible to access the object's data content and meta-data via this API's methods. Access is controlled via authentication and authorization. The interface considers the common standards OSLC [15] and ASAM XiL-API [16]. As seen in the example above it is possible and reasonable to directly serialize the data within the simulation system. In this example, we provide the data in MATLAB's workspace (see Figure 11) as variables. Thus they can directly be used and modified within that environment.



```
>> modelData

modelData =

         Auxiliaries: [1x1 struct]
         BrakeSystem: [1x1 struct]
             Chassis: [1x1 struct]
          Drivetrain: [1x1 struct]
    ElectricalSystem: [1x1 struct]
              Engine: [1x1 struct]
     FuelConsumption: [1x1 struct]
         WheelSystem: [1x1 struct]

>> modelData.FuelConsumption

ans =

          Image: [1x1 struct]
      MomOfInrt: [1x1 struct]
         EngTrq: [1x1 struct]
        DragTrq: [1x1 struct]
      FuelConsum: [1x1 struct]
    FuelDensity: [1x1 struct]

>>
```

Figure 11: Example of the serialization of a complete MATLAB simulation environment via an API and workspace allocation

4. Conclusions

The simulation object management solution presented in this paper provides an innovative approach to enable cross-department and cross-domain collaboration. It also fulfills the requirements for the verification of functional safety specified by ISO 26262 and helps handle the growing data overload found in X-in-the-Loop development environments.

The architecture of the data backbone can be adapted, extended and configured to support internal development standards and existing toolchains. The resulting unified data base constitutes as a foundation for cross-disciplinary collaboration. Since the object management platform has been developed independent of any specific simulation toolchain, in accordance with CPO (Code of PLM Openness) [17], past and future investment protection is assured.

A major contribution to higher acceptance among simulation engineers for an overall simulation data management is done by the direct connection and support of local simulation systems without the need of data transformation or import / export interactions. As a result development projects are executed more efficiently and providing higher quality results.

5. References

[1] Dessort, R., Simon, P., and Pfau, J. 2013. Multi-ECU HiL-Systems for Virtual Characteristic Rating of Vehicle Dynamics Control Systems. In *VDI-Conference Simvec Spezial Simulation Fahrdynamik 2013*, Baden-Baden.

[2] Chucholowski, C. 2014. *The challenge of vehicle dynamics simulation – a critical assessment. ATZ Extra,* 2014/01. http://www.tesis-dynaware.com/fileadmin/Downloads/Presse/vehicle_dynamics_simulation_challenges.pdf. Accessed 19 January 2015.

[3] ElMaraghy, W. 2009. Knowledge Management in Collaborative Engineering. *International Journal of Collaborative Engineering* 1, 1, 114–124.

[4] International Organization for Standardization. 2011. *Road vehicles - Functional safety - Part 8: Supporting processes,* ISO 26262-8:2011-11.

[5] Rochkind, M. J. 1975. The source code control system. *IEEE Transactions on Software Engineering,* 4, 364–370.

[6] Koc, A. and Tansel, A. U. 2011. A survey of version control systems. In *The 2nd International Conference on Engineering and Meta-Engineering: ICEME 2011*, Orlando.

[7] Tolone, W., Ahn, G.-J., Pai, T., and Hong, S.-P. 2005. Access control in collaborative systems. *ACM Computing Surveys (CSUR)* 37, 1, 29–41.

[8] Sandhu, R., Ferraiolo, D., and Kuhn, R. 2000. *The NIST model for role-based access control: towards a unified standard*. National Institute of Standards and Technology.

[9] Object Management Group. *OMG Unified Modeling Language (OMG UML),* formal/2011-08-05. http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF. Accessed 19 January 2015.

[10] Lahres, B. and Rayman, G. 2009. *Objektorientierte Programmierung*. Galileo Press, Bonn.

[11] Girod, M. and Shpichko, T. 2011. *IBM Rational ClearCase 7.0: Master the Tools that Monitor, Analyze, and Manage Software Configurations*. Packt Publishing Ltd, Birmingham.

[12] Collins-Sussman, B., Fitzpatrick, B. W., and Pilato, C. M. 2011. *Version Control with Subversion: For Subversion 1.7*.

[13] Network Working Group. 2007. *HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV)*. The IETF Trust. http://www.webdav.org/specs/rfc4918.html. Accessed 20 January 2015.

[14] Dusseault, L. B. 2003. *WebDAV: Next Generation Collaborative Web Authoring*. Prentice Hall Professional Technical Reference.

[15] OSLC Core Specification Workgroup. 2013. *Open Services for Lifecycle Collaboration Core Specification Version 2.0*. http://open-services.net/bin/view/Main/OslcCoreSpecification. Accessed 20 January 2015.

[16] ASAM. 2013. *ASAM XIL V2.0.0*. http://www.asam.net/nc/home/standards/standard-detail.html?tx_rbwbmasamstandards_pi1%5BshowUid%5D=2529. Accessed 20 January 2015.

[17] ProSTEP iViP. *Code of PLM Openness (CPO)*. http://www.prostep.org/de/cpo.html. Accessed 22 January 2015.