A Model-Driven Engineering Approach to Software Tool Interoperability based on Linked Data

Jad El-khoury, Didem Gurdur
Department of Machine Design
KTH Royal Institute of Technology
Stockholm, Sweden
email:{jad, dgurdur}@kth.se

Mattias Nyberg
Scania CV AB
Södertälje, Sweden
email: mattias.nyberg@scania.com

Abstract—**Product development environments need to shift from the current document-based, towards an information-based focus, in which the information from the various engineering software tools is well integrated and made digitally accessible throughout the development lifecycle. To meet this need, a Linked Data approach to software tool interoperability is being adopted, specifically through the Open Services for Lifecycle Collaboration (OSLC) interoperability standard. In this paper, we present a model-driven engineering approach to toolchain development that targets the specific challenges faced when adopting the technologies, standards and paradigm expected of Linked Data and the OSLC standard. We propose an integrated set of modelling views that supports the early specification phases of toolchain development, as well as its detailed design and implementation phases. An open-source modelling tool was developed to realize the proposed modelling views. The tool includes a code generator that synthesizes a toolchain model into almost-complete OSLC-compliant code. The study is based on a case study of developing a federated OSLC-based toolchain for the development environment at the truck manufacturer Scania AB.**

Keywords-Linked data modelling; OSLC; resource shapes; tool integration; tool interoperability, information modelling.

I. INTRODUCTION

This article is an extended version of [1], in which we expand the earlier focus on the specification phase, to present a more complete development approach to software tool interoperability. The new approach includes a tighter incorporation of the later phases of design and implementation of tool interfaces. Based on additional work on the case study, further refinements of the proposed models and supporting tools are also reflected in this article.

The heterogeneity and complexity of modern industrial products requires the use of many engineering software tools, needed by the different engineering disciplines (such as mechanical, electrical, embedded systems and software engineering), and throughout the entire development life cycle (requirements analysis, design, verification and validation, etc.). Each engineering tool handles product information that focuses on specific aspects of the product, yet such information may well be related or dependent on information handled by other tools in the development environment [2]. It is also the case that a tool normally manages its product information internally as artefacts stored on a file system or a database using a tool-specific format or schema. Therefore, unless interoperability mechanisms are developed to connect information across the engineering tools, isolated “islands of information” may result within the overall development environment. This in turn leads to an increased risk of inconsistencies, given the natural distribution of information across the many tools and data sources involved.

As an example from the automotive industry, the functional safety standard ISO 26262:2011 [3] mandates that requirements and design components are to be developed at several levels of abstraction; and that clear trace links exist between requirements from the different levels, as well as between requirements and system components. Such a demand on traceability implies that these development artifacts are readily and consistently accessible, even if they reside across different development tools. Naturally, the current industry practice, in which development artefacts are handled as text-based documentation, renders such traceability ineffective – if not impossible. The ongoing trend of adopting the Model-Driven Engineering (MDE) approach to product development is a step in the right direction, by moving away from text-based artefacts, towards models that are digitally accessible. This leads to an improvement in the quality and efficient access to product and process information. However, while MDE is more accepted in the academic research community, its complete adoption in an industrial context remains somewhat limited, where MDE is typically constrained to a subset of the development lifecycle [22]. Moreover, even where MDE is adopted, mechanisms are still needed to connect the artefacts being created by the various engineering tools, in order to comply with the standard.

In summary, current development practices need a faster shift from the localized document-based handling of artefacts, towards an **Information-based Development Environment (IDE)**, where the information from all development artefacts is made accessible, consistent and correct throughout the development phases, disciplines and tools.

One can avoid the need to integrate the information islands, by adopting a single platform (such as PTC Integrity [4] or MSR-Backbone [5]) through which product data is centrally managed. However, large organizations have specific development needs and approaches (processes, tools, workflow, in-house tools, etc.), which lead to a wide landscape of organization-specific and customized development environments. Moreover, this landscape needs to evolve organically over time, in order to adjust to future unpredictable needs of the industry. Contemporary platforms, however, offer limited customization capabilities to tailor for the organization-specific needs, requiring instead the organization to adjust itself to suite the platform. So, while they might be suitable at a smaller scale, such centralized platforms cannot scale to handle the complete heterogeneous set of data sources normally found in a large organization.

A more promising integration approach is to acknowledge the existence of distributed and independent data sources within the environment. To this end, OASIS OSLC [6] is an emerging interoperability open standard (see Section II for further details) that adopts the architecture of the Internet and its standard web technologies to integrate information from the different engineering tools - without relying on a centralized integration platform. This leads to low coupling between tools, by reducing the need for one tool to understand the deep data of another. Moreover – like the web – the approach is technology-agnostic, where tools can differ in the technologies they use to internally handle their data. That is, both the data as well as the technology is decentralized. Such an approach lends itself well to the distributed and organic nature of the IDE being desired - a Federated IDE (F-IDE), where the information from all development artefacts – across the different engineering tools – is made accessible, consistent and correct throughout the development phases, disciplines and tools.

In this paper, we advocate the use of OSLC and the Linked Data principles as a basis for such an F-IDE. Yet, when developing such a federated OSLC-based F-IDE for parts of the development environment at the truck manufacturer Scania AB, certain challenges were encountered that needed to be addressed. Put generally, there is an increased risk that one loses control over the overall product data structure that is now distributed and interrelated across the many tools. This risk is particularly aggravated if one needs to maintain changes in the F-IDE over time.

We here propose a model-driven engineering approach to F-IDE development that tries to deal with this risk. That is, how can a distributed architecture – as promoted by the Linked Data approach – be realized, while maintaining a somewhat centralized understanding and management of the overall information model handled within the F-IDE?

In the next section, we will first give some background information on Linked Data and the OASIS OSLC standard. We then present the case study that has driven and validated this work in Section III. Section IV then elaborates on the challenges experienced during our case study, before detailing the modelling approach taken to solve these challenges in Section V. Details on the modelling views, as well as their realisation in an open-source tool, are presented. Reflections on applying the modelling approach on the case study are then discussed in Section VI, followed by a discussion of related work. The article is then concluded in Section VIII.

II. Linked Data and the OASIS OSLC Standard

Linked Data is an approach for publishing structured data on the web, such that data from different sources can be connected, resulting in more meaningful and useful information. Linked Data builds upon standard web technologies such as HTTP, URI and the RDF family of standards. The reader is referred to [7] for Tim Berners-Lee's four principles of Linked Data.

OASIS OSLC is a standard that targets the integration of heterogeneous software tools, with a focus on the linking of data from independent sources. It builds upon the Linked Data principles, and its accompanying standards, by defining common mechanisms and patterns to access, manipulate and query resources managed by the different tools in the toolchain. In particular, OASIS OSLC is based on the W3C Linked Data Platform (LDP) [8], and it follows the Representational State Transfer (REST) architectural pattern.

This Linked Data approach to tool interoperability promotes a distributed architecture, in which each tool autonomously manages its own product data, while providing – at its interface - RESTful services through which other tools can interconnect. Figure 1 illustrates a typical architecture of an OSLC tool interface, and its relation to the tool it is interfacing. With data exposed as RESTful services, such an interface is necessarily an “OSLC Server”, with the connecting tool defined as an “OSLC Client”. Following the REST architectural pattern, an OSLC server allows for the manipulation of artefacts – once accessed through the services - using the standard HTTP methods C.R.U.D. to Create, Read, Update and Delete. In OSLC, tool artefacts are represented as RDF resources, which can be represented using RDF/XML, JSON, or Turtle. A tool interface can be provided natively by the tool vendor, or through a third-party as an additional adaptor. In either case, a mapping between the internal data and the exposed RDF resources needs to be done. Such mapping needs to deal with the differences in the technologies used. In addition, a mapping between the internal and external vocabulary is needed, since the vocabulary of the resources being exposed is not necessarily the same as the internal schema used to manage the data.

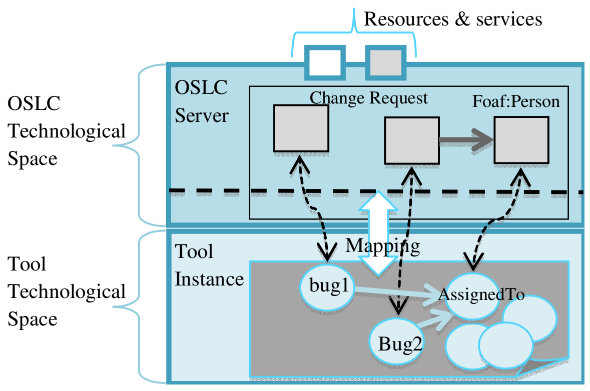


Figure 1. Typical tool architecture, with an OSLC Server

III. Case Study Description

Typical of many industrial organizations, the development environment at the truck manufacturer Scania consists of standard engineering tools, such as issue-tracking and computer-aided design (CAD) tools; as well as a range of proprietary tools that cater for specific needs in the organization. Moreover, much product information is managed as generic content in office productivity tools, such as Microsoft Word and Excel.

To comply with the ISO 26262 standard, the current development environment needs to improve in its management of vehicle architectures and requirement specifications, in order to provide the expected traceability between them. This in turn necessitates a better integration of the tools handling the architecture and requirements artefacts to allow for such traceability. To this end, five proprietary tools and data sources were to be integrated using OSLC:

1. *Code Repository* – A tool that defines the vehicle software architecture through parsers that analyze all software code to reconstruct the architecture, defining entities such as software components and their communication channels [9]. The analyzed software code resides in a typical version-control system. When defining the architecture, the tool references artefacts – defined in other external tools - dealing with communication and the hardware architecture.
2. *Communication Specifier* – A tool that centrally defines the communication network and the messages sent between components of all vehicle architectures.
3. *ModArc* – A CAD tool that defines the electrical components, including all hardware entities and their interfaces such as communication ports.
4. *Diagnostics Tool* - A tool that specifies the diagnostics functionality of all vehicle architectures, including communication messages of relevance to the diagnostics functionality.
5. *Requirements Specifier* - A proprietary tool that allows for the semi-formal specification of system requirements [10]. Requirements are specified at different levels of abstraction. By anchoring the specifications on different parts of the system architecture, the tool helps the developer define correct requirements that can only reference appropriate product artefacts within the system architecture.

As a first step, it was necessary to analyze the data that needed to be communicated between the tools. This was captured using a Class Diagram (Figure 2), as is the current state-of-practice at Scania for specifying a data model. For the purpose of this paper, it is not necessary to have full understanding of the data artefacts. It is worth highlighting that color-codes were initially used to define which tool managed which data artefact. Yet, this appeared to be a non-trivial task since an artefact might be used in multiple tools, with no clear agreement on the originating source tool. For example, a *Signal* can be found in both the *Code Repository* as well as the *Communication Specifier* tool. Given that there exists no data integration between the two tools to keep the artefact synchronized, different developers may have a different perspective over which of the two tools holds the source and correct *Signal* information, from which the other tool needs to be – manually – updated.

In addition, it is important to note that the model focuses on the data to be communicated between the tools, and not necessarily all data available internally within each tool.

IV. Identified Needs and Shortcomings

In this paper, we focus on the initial development stages of specifying and architecting the desired OSLC-based F-IDE, as well as its design and implementation. The latter verification and validation phases are not yet covered in the case study, yet there is naturally recognition of the need to support them in the near future. Based on the case study, we here elaborate on the needs and shortcomings experienced by the toolchain architects and developers during these stages:

Information specification – There is a need to specify an information model that defines the types of artefacts or resources to be communicated between the tools across the toolchain. For pragmatic reasons, a UML class diagram was initially adopted by the Scania toolchain architects to define the entities being communicated and their relationships. Clearly, the created model does not comply with the semantics of the class diagram, since the entities being modelled are not objects in the object-oriented paradigm, but resources according to the Resource Description Framework (RDF) graph data model. Since the information model is to be maintained over time, and is intended for communication among developers, using a class diagram - while implying another set of semantics – may lead to misunderstandings. A specification that is semantically compatible with the intended implementation technology (of Linked Data, and specifically the OSLC standard) is necessary. However, the initial experience from using the class diagram helped identify the necessary requirements on any appropriate solution. First, graphical models are essential to facilitate the communication of the models among the different stakeholders. It is also beneficial to – wherever possible - borrow or reuse graphical representations from common modelling frameworks (such as UML) in order to reduce the threshold of learning a new specification language. For example, adopting a hollow triangle shape to represent class inheritance (as defined in UML) would be recommended in RDF modelling as well.

Domain ownership – It is necessary to structure the information model specification into domains (such as requirements engineering, software, testing, etc.). Domains can be generic in nature. Alternatively, such domain grouping can reflect the organization units that are responsible to manage specific parts of the information model. For example, the testing department may be responsible to define and maintain the testing-related resources, while the requirements department manages the definition of the requirements resources. This is particularly relevant in an organization where different departments are responsible for their own tools and processes, and where it no longer becomes feasible to expect the information model to be centrally defined. Dependencies between the responsible departments can then be easily identified through the dependencies in the information models.

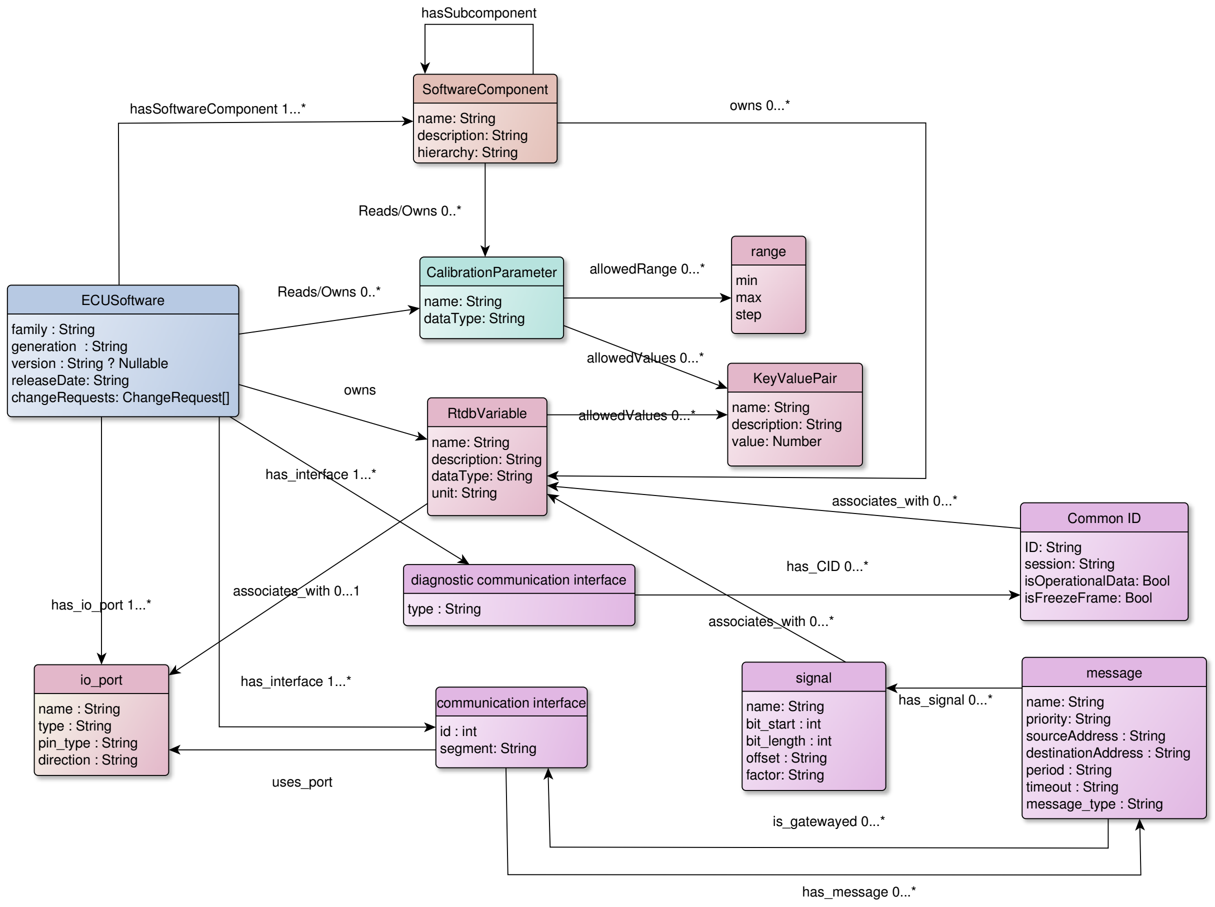


Figure 2. A UML class diagram of the resources shared in the desired F-IDE.

Tool ownership – Orthogonal to domain ownership, it is also necessary to clearly identify the data source (or authoring tool) that is expected to manage each defined resource being shared in the F-IDE. That is, while representations of a resource may be freely shared between the tools, changes or creations of such a resource can only occur via its owning tool. Assuming a Linked Data approach also implies that a resource is owned by a single source, to which other resources link. In practice, it is not uncommon for data to be duplicated in multiple sources, and hence mechanisms to synchronize data between tools are needed. For example, resources of type *Communication Interface* may be used in both *Communication Specifier* and *ModArc,* with no explicit decision on which of the tools defines it. To simplify the case study, we chose to ignore the *ModArc* source, but in reality, one needs to synchronize between the two sources, as long as it is not possible to make one of them redundant.

That is, in architecting an F-IDE, there is a need to support the data specification using Linked Data semantics, while covering the two ownership aspects of tools (ownership from the tool deployment perspective) and domains (ownership from the organizational perspective).

Avoid mega-meta-modelling – Information specifications originate from various development phases and/or development units in the organization. The resulting information models may well overlap, and would hence need to be harmonized. Hence, there is a need to harmonize the information models – while avoiding a central information model. Earlier attempts at information modeling normally resulted in large models that can easily become harder to maintain over time. The research project CESAR presents in [11] a typical interoperability approach in which such a large common meta-model is proposed. It is anticipated that the Linked Data approach would reduce the need to have such a single centralized mega information model. The correct handling of information through Domain and Tool Ownership (see above) ought to also help in that direction.

Development support – Similar to the challenge faced in general software development, there is a need to maintain the information specification and desired architecture harmonious with the eventual design and implementation of the F-IDE and its components. The current use of a class diagram works well as an initial specification, and for documentation purposes. However, there is no mechanism in place to ensure the model is updated relative to changes later performed during the development. Especially when adopting an agile development approach, the specification and architecture are expected to change over time, and hence the implementations of individual adaptors need to capture such eventual changes. Likewise, feedback from the design and implementation phases may lead to necessary changes in the specification and architecture.

Appropriate tool support is needed to make the specification models an integral part of development. This can take the form of a Software Development Kit (SDK), code generators, graphical models, analysis tools, etc. Such tool support should also help lower the threshold of learning as well as adopting OSLC, since implementing OSLC-compliant tools entails competence in a number of additional technologies such as RESTful web services and the family of RDF standards.

V. MODELLING SUPPORT

We take an MDE approach to F-IDE development, in which we define a graphical modelling language that supports the toolchain architects and developers with the needs identified in the previous section.

The language is designed to act as a digital representation of the OASIS OSLC standard. This ensures that any defined toolchain complies with the standard. Such a graphical representation also helps lower the threshold of learning as well as implementing OSLC-compliant toolchains.

The language is structured into a set of views, in which each view focuses on a specific need, stakeholder and or aspect of development. The analysis of the needs from Section IV leads to the following three views:

• **Domain Specification View** – for the specification of the information to be shared across the F-IDE, with support for the organizational needs.

• **Resource Allocation View** – for the specification of information distribution and ownership across the F-IDE architecture.

• **Adaptor Design View** – for the detailed design and implementation of the tool interfaces of the F-IDE.

The next subsection presents further details of the OSLC standard, which then leads to its reflection by the proposed meta-model. Based on this OSLC meta-model, three views are then derived in Section V.B. The proposed graphical notation of each view is presented through examples from the use case of Section III. Finally, Section V.C details the open-source modelling tool developed to realize the proposed approach.

A. The Meta-model

The OASIS OSLC standard consists of a Core Specification and a set of Domain Specifications. The OSLC Core Specification [12] defines the set of resource services that can be offered by a tool. Figure 3 illustrates the structure of an OSLC interface and its services. A Service Provider is the central organizing entity of a tool, under which artefacts are managed. Typical examples of a Service Provider are project, module, product, etc. It is within the context of such an organizing concept that artefacts are managed (created, navigated, changed, etc.). For a given Service Provider, OSLC allows for the definition of two Services (Creation Factory & Query Capability) that provide other tools with the possibility to create and query artefacts respectively. In addition, OSLC defines Delegated UI (Selection and Creation) services that allow other tools to delegate the user interaction with an external artefact to the Service Provider under which the artefact is managed. The structure of Figure 3 allows for the discoverability of the services provided by each Service Provider, starting with a Service Provider Catalog, which acts as a catalog listing all available Service Providers exposed by a tool.

OASIS OSLC also defines Domain Specifications, which include domain vocabularies (or information models) for specific lifecycle domains. For example, the Quality Management Specification [13] defines resources and properties related to the verification phase of development such as test plans, test cases, and test results. The standardized Domain Specifications are minimalistic, focusing on the most common concepts within a particular domain, while allowing different implementations to extend this common basis.

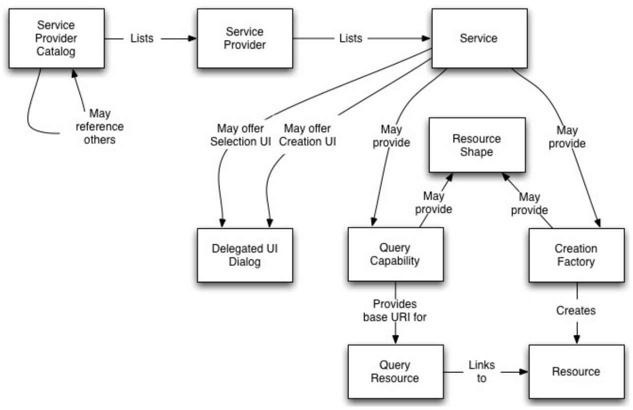


Figure 3. OSLC Core Specification concepts and relationships [12]

Using EMF [18], we define the meta-model that reflects the structure and concepts of the OASIS OSLC standard, as illustrated in Figure 4. A *Toolchain* consists of (1) a set of *AdaptorInterfaces* and (2) a set of *DomainSpecifications* (for legacy reasons grouped under a Specification element):

• An *AdaptorInterface* represents a tool's OSLC interface, and reflects the Core standard structure as illustrated in Figure 3.

• A *DomainSpecification* reflects how an OSLC Domain Specification defines vocabularies. It models the resources types, their properties and relationships, based on the Linked Data constraint language of Resource Shapes [14]. Resource Shapes is a mechanism to define the constraints on RDF resources, whereby a Resource Shape defines the properties that are allowed and/or required of a type of resource; as well as each property's cardinality, range, etc.

B. The Modelling Views

Based on the OSLC meta-model, we define the following three views:

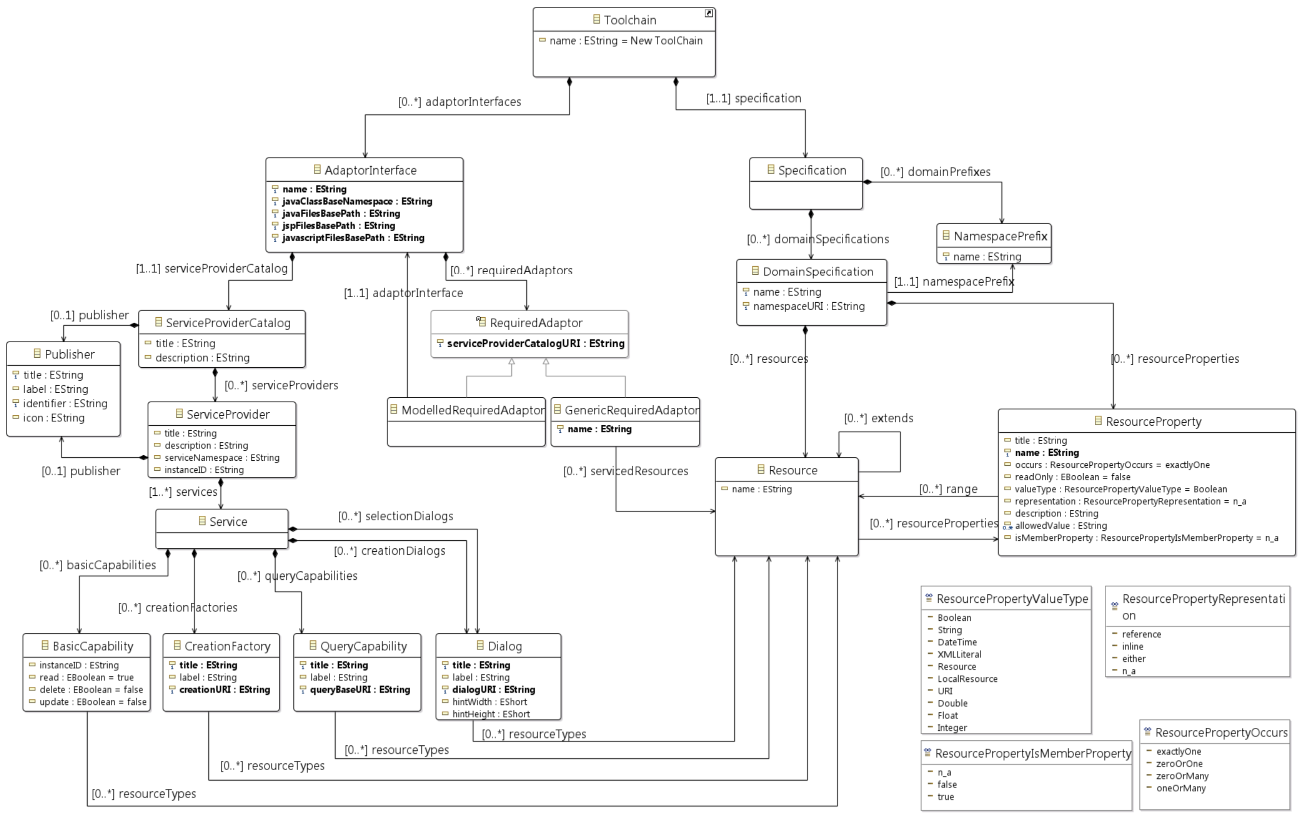


Figure 4. The underlying meta-model of the OASIS OSLC standard, reflecting the Core and Domain Specifications.

Domain Specification View From this perspective, the toolchain architect defines an information model that details the types of resources, their properties and relationships, using mechanisms compliant with the OSLC Core Specification [12] and the Resource Shape constraint language [14]. Figure 5 exemplifies the proposed graphical notation of the Domain Specification view for the resources needed in our case study.

The top-level container, *DomainSpecification*, groups related *Resources* and *Resource Properties*. Such grouping can be associated with a common topic (such as requirements or test management), or reflects the structure of the organization managing the F-IDE. This view ought to support standard specifications, such as Friend of a Friend (FOAF) [15] and RDF Schema (RDFS) [16], as well as proprietary ones. In Figure 5, three Domain Specifications are defined: *Software*, *Communication* and *Variability*, together with a subset of the standard domains of Dublin Core and RDF.

As required by the OSLC Core, a specification of a *Resource* type must provide a *name* and a *Type URI*. The *Resource* type can then also be associated with its allowed and/or required properties. These properties could belong to the same or any other *DomainSpecification*. A *Resource Property* is in turn defined by specifying its cardinality, optionality, value-type, allowed-values, etc. Figure 6 illustrates an example property specification highlighting the available constraints that can be defined. A *Literal Property* is one whose value-type is set to one of the predefined literal types (such as string or integer); while a *Reference Property* is one whose value-type is set to either “resource” or “local resource”. In the latter case, the *range* property can then be used to suggest the set of resource types the *Property* can refer to.

In RDF, *Resource Properties* are defined independently, and may well be associated with multiple *Resource* types (Unlike, for example UML Classes, where a class attribute is defined within the context of a single class). For this reason, *Resource Properties* are graphically represented as first-class elements in the diagram. So, borrowing from the typical notation used to represent RDF graphs, *Resource* types are represented as ellipses, while *Properties* are represented as rectangles (A *Reference Property* is represented with an ellipse within the rectangle.).

The association between a *Resource* type and its corresponding *Properties* is represented by arrows for *Reference Properties*, while *Literal Properties* are listed graphically within the *Resource* ellipse. Such a representation renders the diagram almost similar – visually - to the UML class diagram of Figure 2. This makes the diagram intuitive and familiar for the modeler, yet with the more appropriate Linked Data semantics behind the view.

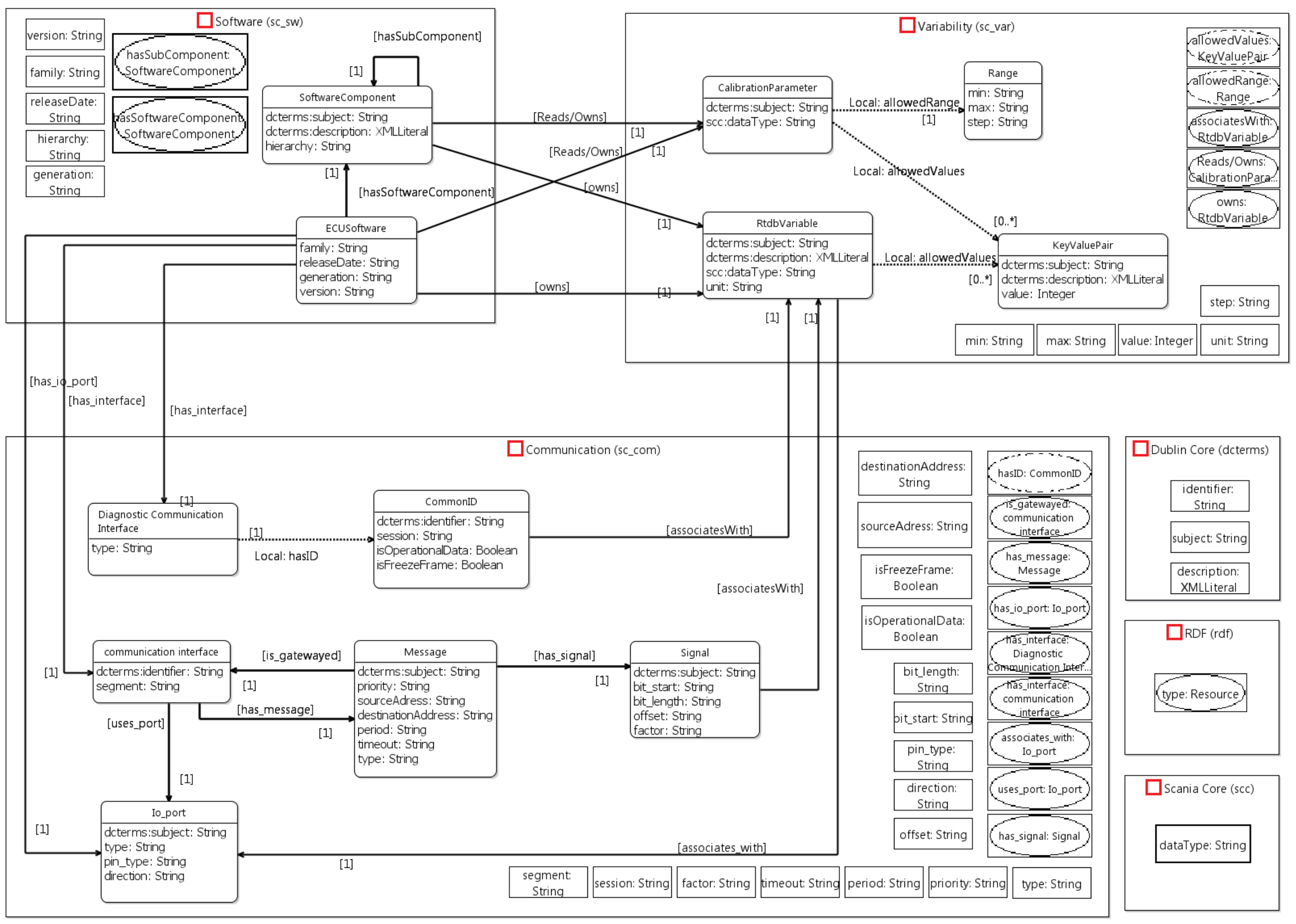
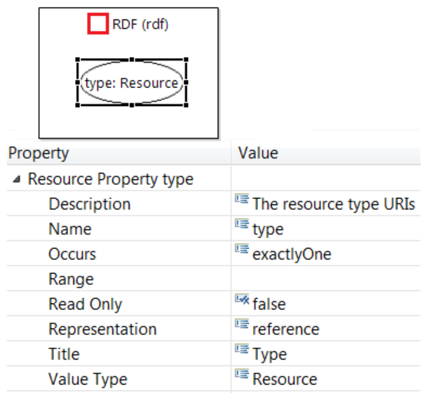


Figure 5. Domain Specification View

While the possibility to represent *Properties* as first-class elements was appreciated, it was experienced that they (The squares in Figure 5) cluttered the overall model, and did not make efficient usage of the available modeling space. An alternative representation is available, in which *Property* definitions are collected within a single sub-container, confined within its containing *DomainSpecification*. Figure 7 presents this alternative diagram for a subset of the domain specification of Figure 5, focusing on the *Communication* Domain Specification. Such a notation still ensures that *Properties* are defined independently of *Resources*, while making the graphical entities more manageable for the modeler.

Moreover, typical RDF graphs notations represent all associations between resources and properties by arrows, irrespective of whether they are Literal or Reference Properties. If desired, such a representation can be chosen as well. Figure 8 presents this alternative for a subset of the domain specification of Figure 5, focusing on the *Software* Domain Specification. Such a representation is intuitive for a small specification. However, it is experienced that for large specifications, the many associations between *Resources* and their associated *Literal Properties* cluttered the diagram. Furthermore, common *Literal Properties*, such as *dcterms:subject*, can be associated to many resources across many domains, leading to many cross-domain arrows that further clutter the diagram.



Property	Value
Resource Property type	
Description	The resource type URIs
Name	type
Occurs	exactlyOne
Range	
Read Only	false
Representation	reference
Title	Type
Value Type	Resource

Figure 6. The specification of the rdf:type predicate, in the Domain Specification View [1]

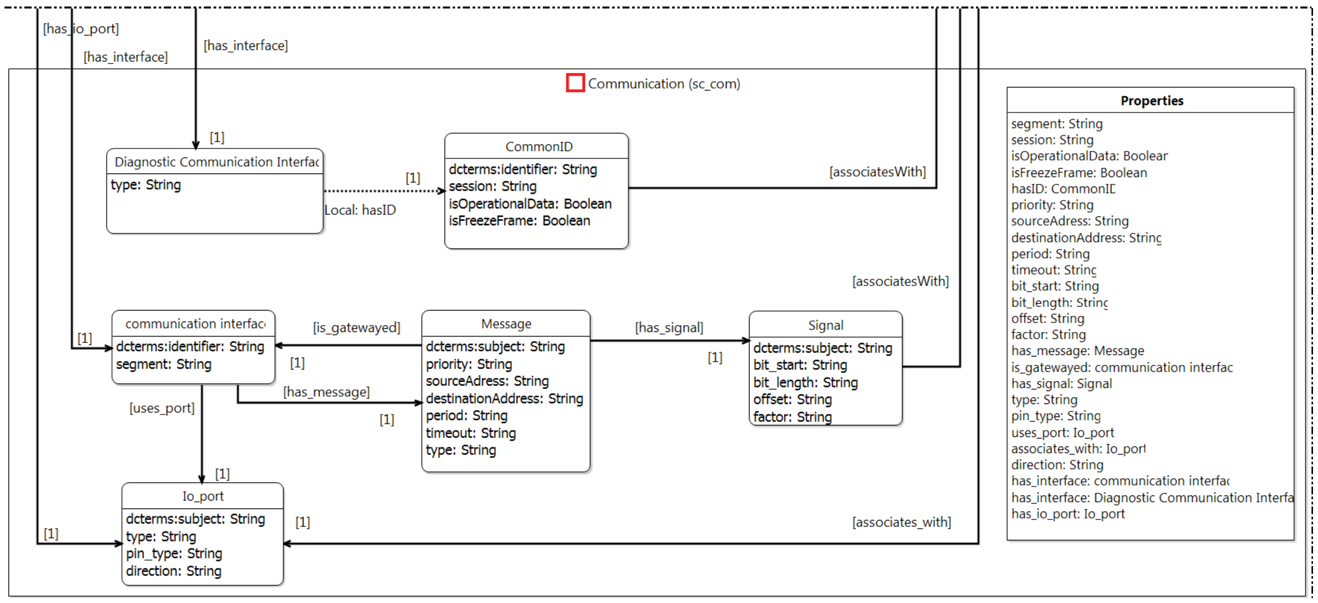


Figure 7. An alternative Domain Specification View, with Property definitions collected within a single sub-container.

The three alternative representations of *Resource Properties* and their associations with *Resources* are made available through filtering mechanisms to suite the preferences of the modeler. It is important to recall that these alternatives are semantically similar, and are based on the same meta-model of Figure 4.

Resource Allocation View provides the overall architecture of the F-IDE, where the toolchain architect allocates resources to data sources. It gives the architect an overview of where the resources are available in the F-IDE, and where they are consumed. For each data source, the architect defines the set of *Resources* it exposes; as well as those it consumes. These *Resources* are graphically represented as *"provided"* and *"required"* ports on the edge of the *AdaptorInterface* element, as illustrated in Figure 9 for our case study. For example, the *Communication Specifier* interface exposes the *Message* resource, which is then consumed by the *Requirements Specifier*.

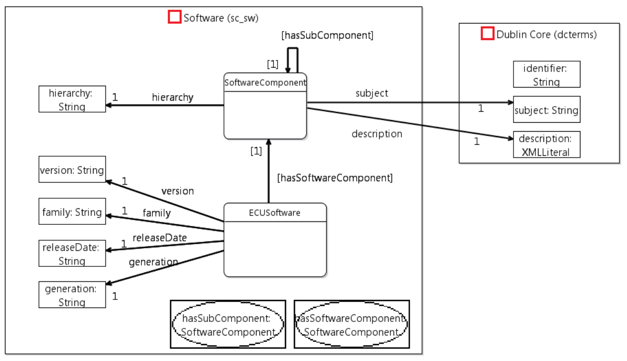


Figure 8. An alternative Domain Specification View, representing all associations between resources and properties by arrows.

In the *Resource Allocation* view, the interaction between a provider and consumer of a given resource is presented as a solid edge between the corresponding ports. In addition, any dependencies between resources that are managed by two different data sources are also represented in this model – as a dotted edge. For example, the resource *ECUSoftware*, managed by the *Code Repository*, has a property *has_io_port* that is a reference to resource *IO_port* (which is in turn managed through the data source *Modarc*). Hence, for a consumer of *ECUSoftware*, it is beneficial to identify the indirect dependency on the *Modarc* tool, since any consumption of an *ECUSoftware* resource, is likely to lead to the need to communicate with *Modarc* in order to obtain further information about the property *has_io_port*.

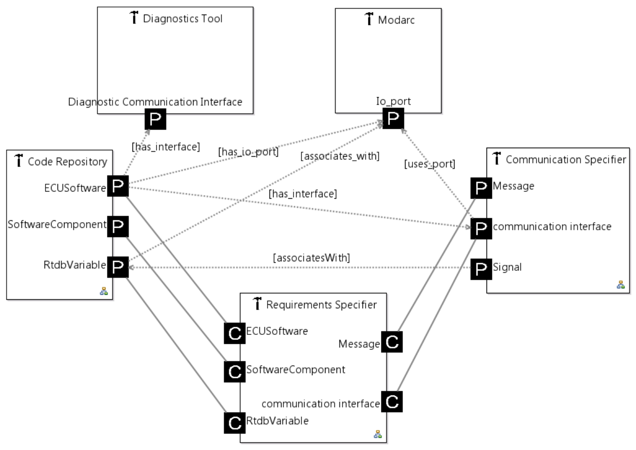


Figure 9. Resource Allocation View

Adaptor Design View is where the toolchain architect (or the tool interface developer) designs the internal details of the tool interface – according to the OSLC standard. This can be performed for any of the *Tool* entities in the *Resource Allocation* view. The *Adaptor Design* view is a realization of the OSLC interface structure of Figure 3. Sufficient information is captured in this view, so that an almost complete interface code, which is compliant with the OSLC4J software development kit (SDK) can be generated, based on the Lyo code generator [17] (See next subsection for further details.).

An example of the proposed notation from our case study is presented in Figure 10, in which the *Core Repository* provides query capabilities and creation factories on all three resources. The *Adaptor Design* view also models its consumed resources (In Figure 10, no consumed resources are defined.). Note that the provided and required resources - as defined in this view - remain synchronized with those at the interface of the *Tool* entity in the *Resource Allocation* view.

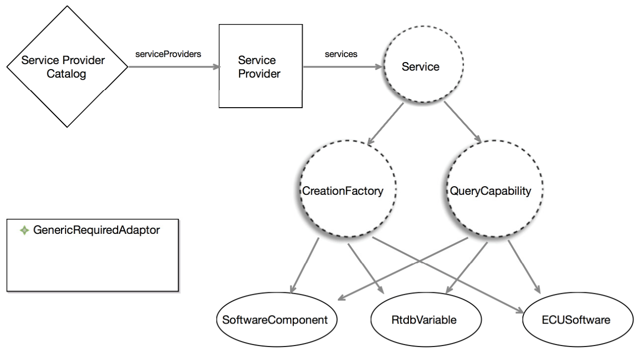


Figure 10. Adaptor Design View [1]

There is no particular ordering of the above views, and in practice, the three views can be developed in parallel. Consistency between the views is maintained since they all refer to the same model. For example, if the toolchain architect removes a resource from the *Adaptor Design* view, the same resource is also removed from the *Resource Allocation* view.

C. *Architecture of Modelling Tool*

An open-source Eclipse-based modelling tool was developed to realize the proposed approach, whose main components are presented in Figure 11. Central in the architecture is the *Toolchain Meta-model* component that realizes the meta-model of Figure 4, based on the Eclipse Modeling Framework (EMF) [18]. The *Graphical Modelling Editor* then allows the end-user to graphically design a toolchain based on the three views presented in Section V.B. (The figures presented in that section are snapshots of the graphical editor.) A toolchain design model is ultimately an instance of the toolchain meta-model. This model can then be inputted into the *Lyo Code Generator* [17] to generate almost-complete code for each of the tool interfaces in the toolchain.

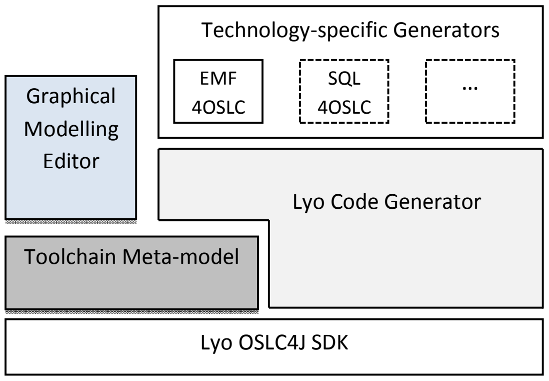


Figure 11. The layered architecture of the modelling tool, building upon the Lyo OSLC4J SDK, to provide a model-based development approach.

The *Lyo code generator* runs as a separate Eclipse project, and assumes a minimal set of plug-in dependencies. It is based on Acceleo [19], which implements the OMG MOF Model-to-Text Language (MTL) standard [20]. The code generator is designed to be independent of the *Graphical Modelling Editor*, and hence its input tool meta-model instance can be potentially created by any other EMF mechanism. This facilitates the extension of the generator with additional components as later described in this section.

The code generator builds upon the OSLC4J Software Development Kit (SDK) from the Lyo [21] project. While the OSLC4J SDK targets the implementation phase of adaptor implementation, our tool complements it with a model-based development approach, which allows one to work at a higher level of abstraction, with models used to specify the adaptor design, without needing to deal with all the technical details of the OSLC standard (such as Linked Data, RDF, etc.).

The generator produces OSLC4J-compliant Java code. Once generated, the java code has no dependencies to either the code generator, or the input toolchain model. The generated code can be further developed – as any OSLC4J adaptor – with no further connections to the generator.

Moreover, it is possible to modify the toolchain model and re-generate its code, without the loss of any code manually introduced between generations. For example, a JAX-RS class method may need to perform some additional business logic before - or after - the default handling of the designated web service. The generator ensures that such manual code remains intact after subsequent changes in the model and code generations. This promotes the incremental development of the toolchain, where the specification model and implementation can be gradually developed.

Upon generation, an adaptor is – almost – complete and ready-to-run, and needs not be modified nor complemented with additional manual code. Only a set of methods that communicate with the source tool to access its internal data need to be implemented (the dotted arrows in Figure 1). This communication is reduced to a simple set of methods to (a) get (b) create (c) search and (d) query each serviced resource. This manual code is packaged into a single class, with skeletons of the required methods generated. It remains the task of the developer to manual code these methods.

However, for specific tool technologies, a full adaptor implementation can be generated. For example, targeting EMF-based modelling tools in general, an additional *EMF4OSLC* component was developed [22] to complement the code generator with the automatic generation of the necessary code to access and manipulate the data in the backend tool. This leads to the complete generation of the adaptor. *EMF4OSLC* moreover generates the actual interface specification model based on a predefined mapping between EMF and OSLC. Similarly, an additional component – *EMF4SQL* – is being developed to handle SQL-based tools. It is anticipated that much code reuse can be gained between *EMF4OSLC* and *EMF4SQL*, while ensuring that both components build on top of the Lyo code generator.

The modelling tool and supporting documentation are available as open-source under the Eclipse Lyo [21] project.

VI. Discussion

Compared to the original approach practiced by Scania engineers of using a UML class diagram (see Figure 2) to represent the F-IDE resources, the proposed model may seem to add a level of complexity by distributing the model information into three views. However, upon further investigation, it becomes clear that the class diagram was actually used to superimpose information for both the *Domain Specification* and *Resource Allocation* views into the same diagram. For example, classes were initially color-coded to classify them according to their owning tool. However, the semantics and intentions behind this classification soon become ambiguous, since the distinction between tool and domain ownership is not identified explicitly. In the original approach, different viewers of the same model could hence draw different conclusions when analyzing the model, depending on their implicit understanding of the color codes.

Through a multi-view modelling approach, and by describing the information model from the two orthogonal views of managing domains and managing tools, the information model is no longer expected to be developed in a top-down and centralized manner. Instead, a more distributed process is envisaged, in which resources are defined within a specific domain and/or tool. Only when necessary, such sub-models can then be integrated, avoiding the need to manage a single centralized information model. Moreover, these two orthogonal views of the F-IDE allow the toolchain architect to identify dependencies within the F-IDE, from both the organizational as well as the deployment perspective:

- In the *Resource Allocation View* of the model, the toolchain architect can obtain an overview of the coupling/cohesion of the tools of the F-IDE. One could directly identify the direct producer/consumer relations, as well as the indirect dependencies, as detailed in Section V.B.
- In the *Domain Specification view*, the toolchain architect views the dependencies between the different domains (irrespective of how the resources are deployed across tools). Such dependencies reveal the relationship between the organizational entities involved in maintaining the overall information model. This explicit modelling of domain ownership helps lift important organizational decisions, which otherwise remain implicit.

Semantically, the usage of a class diagram is not compatible with the open-world view of Linked Data. Instead, a dedicated domain-specific language (DSL) that follows the expected semantics can be better used uniformly across the whole organization. We here illustrate two examples where our DSL helped communicate the correct semantics, which were previously misinterpreted or not used:

- A *Resource Property* is a first-class element that ought to be defined independently of *Resource* definitions within a *Domain Specification*. A *Property* can then be associated with multiple *Resource* types, which in turn can belong to the same or any other *Domain Specification*. For example, the *allowedValues* property (with range *KeyValuePair*) is defined within the *Variability* domain, and its definition ought not to be dependent on any particular usage within any *Resource*. This same *Property* is then being associated to the *CalibrationParameter* & *RtdbVariable* resources. Previously, two separate *Properties* were unnecessarily defined within the context of each *Resource*, which is not appropriate when adopting Linked Data and its RDF data model.
- Certain *Resources* can only exist within the context of another parent Resource, and hence ought not to have their own URI. For example, *Range* is defined as a simple structure of three properties (*min*, *max* and *step*). The *CalibrationParameter* resource contains the *allowedRange* property whose *value-types* is set to *Local Resource*, indicating that property value is a resource that is only available inside the *CalibrationParameter* instance. Our DSL helped communicate the capability of defining *Local Resources*. A class diagram does not provide a corresponding concept that can be correctly used to convey the same semantics.

Adopting a class diagram may have been satisfactory at the early stages of development, where focus was on the information specification. However, it became apparent that the diagram is not sufficient in supporting the later phases of development, when the tool interfaces need to be designed and implemented. Furthermore, no complements to the UML class diagram can provide all necessary information according to the OSLC standard. Instead, the third *Adaptor Design View* serves this need satisfactorily. In addition, by sharing a common meta-model with the other two views, it is ensured that the detailed designs remain consistent with the specification and architecture of the F-IDE. Furthermore, given that the complete model (with its three views) can lead to the generation of working code, the model's completeness and correctness is confirmed.

While the need for a dedicated DSL is convincing, the proposed notations are not necessarily final, and there remains room for improvements. The alternative representations of the *Domain Specification View* (discussed in Section V.B) highlight some of the refinements that need to be dealt with.

VII. Related Work

There exists a large body of research that in various ways touches upon information modeling and tool integration. (See for example [2] and [23]). Our work - and the related work of this section - is delimited to the Linked Data paradigm, and its graphical modelling.

The most relevant work found in this area is the Ontology Definition Metamodel (ODM) [24]. ODM is an OMG specification that defines a family of Meta-Object Facility (MOF) metamodels for the modelling of ontologies. ODM also specifies a UML Profile for RDFS [16] and the Web Ontology Language (OWL) [25], which can be realized by UML-based tools, such as Enterprise Architect's ODM diagrams [26]. However, as argued in [14], OWL and RDFS are not suitable candidates to specify and validate constraints, given that they are designed for another purpose - namely for reasoning engines that can infer new knowledge. The work in this paper builds upon the Resource Shape constraint language suggested in [14], by providing a graphical model to specify such constraints on RDF resources. The constraint language is part of the OSLC standard, making for its easy adoption within our work. On the other hand, SHACL [27] is an evolving W3C working draft for a very similar constraint language, which can also be supported in the future.

Earlier work by the authors has also resulted in a modelling approach to toolchain development [28]. In this earlier work, even though the information was modelled targeting an OSLC implementation, the models were directly embedded in the specific tool adaptors, and no overall information model is readily available. The models did not support the tool and domain ownership perspectives identified in this paper.

In general, there are inspiring works done for defining a visual language for the representation of ontologies. Even if ontologies are not directly suitable for the specification of constraints, such languages can be used as inspiration for the graphical notation presented in this paper. One example is the Visual Notation for OWL Ontologies (VOWL) [29] that is based on only a handful of graphical primitives forming the alphabet of the visual language. In this visual notation, classes are represented as circles that are connected by lines and arrowheads representing the property relations. Property labels and datatypes are shown as rectangles. Information on individuals and data values is displayed either in the visualization itself or in another part of the user interface. VOWL also uses a color scheme complementing the graphical primitives. It defines colors for the visual elements to allow for an easy distinction of different types of classes and properties. In addition to the shapes and colors, VOWL also introduces dashed lines, dashed borders and double borders for visualizing class, relation or data properties.

OWLGrEd [30] is another graphical OWL editor that extends the UML class diagram to allow for OWL visualization. The work argues that the most important feature for achieving readable graphical OWL notation is the maximum compactness. The UML class diagram is used to present the core features of OWL ontologies. To overcome the difference between UML's closed-world assumption and OWL's open-world assumption, the authors changed the semantics of the UML notation and added new symbols. OWLGrEd introduces a colored background frame for the relatively autonomous sub-parts of the ontology. Furthermore, the editor contains a number of additional services to ease ontology development and exploration, such as different layout algorithms for automatic ontology visualization, search facilities, zooming, and graphical refactoring. Finally, GrOWL [31] is a visual language that attempts to accurately visualize the underlying description logic semantics of OWL ontologies, without exposing the complex OWL syntax.

Lanzenberger et al. [32] summarize the results of their literature study on tools for visualizing ontologies as: "*A huge amount of tools exist for visualizing ontologies, however, there are just a few for assisting with viewing multiple ontologies as needed for ontology alignment. … Finally, in order to support an overview and detail approach appropriately, multiple views or distortion techniques are needed.*" We identified the need to support multiple views in this study, in order to support the different stakeholders of the same language.

VIII. Conclusion

In this paper, an MDE approach to F-IDE development based on Linked Data and the OSLC standard is presented. The proposed set of modelling views supports the toolchain architect with the early phases of toolchain development, with a particular focus on the specification of the information model and its distribution across the tools of the toolchain. Additionally, such views are tightly integrated with a design view supporting the detailed design of the tool interfaces. The modelling views are designed to be a digital representation of the OASIS OSLC standard. This ensures that any defined toolchain complies with the standard. It also helps lower the threshold of learning as well as implementing OSLC-compliant toolchains.

An open-source modelling tool was developed to realize the proposed modelling views. The tool includes an integrated code generator that can synthesis the specification and design models into a running implementation. This allows one to work at a higher level of abstraction, without needing to deal with all the technical details of the OSLC standard (such as Linked Data, RDF, etc.). The Eclipse-based modelling tool and supporting documentation are available as open-source under the Eclipse Lyo project.

It is envisaged that the modelling support will be extended to cover the complete development lifecycle, specifically supporting the requirements analysis phase, as well as automated testing. The current focus on data integration needs to be also extended to cover other aspects of integration, in particular control integration [33].

References

[1] J. El-Khoury, D. Gürdür, F. Loiret, M. Törngren, D. Zhang, and M. Nyberg, "Modelling Support for a Linked Data Approach to Tool Interoperability," The Second International

Conference on Big Data, Small Data, Linked Data and Open Data, ALLDATA 2016, pp. 42-47.

[2] M. Törngren, A. Qamar, M. Biehl, F. Loiret, and J. El-khoury, "Integrating viewpoints in the development of mechatronic products," Mechatronics (Oxford), vol. 24, nr. 7, 2014, pp. 745-762.

[3] Road vehicles - functional safety, ISO standard 26262:2011, 2011.

[4] (2016, Nov.) PTC Integrity. [Online]. Available: http://www.ptc.com/application-lifecycle-management/integrity/

[5] B. Weichel and M. Herrmann, "A backbone in automotive software development based on XML and ASAM/MSR," SAE Technical Papers, 2004, doi:10.4271/2004-01-0295.

[6] (2016, Nov.) OASIS OSLC. [Online]. Available: http://www.oasis-oslc.org/

[7] T. Berners-Lee. (2016, Nov.) Linked data design issues. [Online]. Available: http://www.w3.org/DesignIssues/LinkedData.html

[8] Linked Data Platform 1.0, W3C Recommendation, 2015.

[9] X. Zhang, M. Persson, M. Nyberg, B. Mokhtari, A. Einarson, H. Linder, J. Westman, D. Chen, and M. Törngren, "Experience on Applying Software Architecture Recovery to Automotive Embedded Systems," IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE 2014, pp. 379-382.

[10] J. Westman, M. Nyberg, and O. Thyden, "CPS Specifier - A Specification Tool for Safety-Critical Cyber-Physical Systems," Sixth Workshop on Design, Modeling and Evaluation of Cyber Physical Systems, CyPhy'16, 2016

[11] A. Rossignol, "The reference technology platform" in CESAR - Cost-efficient methods and processes for safety-relevant embedded systems, A. Rajan and T. Wahl, Eds. Dordrecht: Springer, pp. 213-236, 2012.

[12] OSLC Core Specification, OSLC standard v2.0, 2013.

[13] OSLC Quality Management Specification, OSLC standard v2.0, 2011.

[14] A. G. Ryman, A. Le Hors, and S. Speicher, "OSLC resource shape: A language for defining constraints on linked data," CEUR Workshop Proceedings, Vol.996, 2013.

[15] (2016, Nov.) FOAF Vocabulary Specification. [Online]. Available: http://xmlns.com/foaf/spec/

[16] RDF Schema 1.1, W3C Recommendation, 2014.

[17] J. El-Khoury, "Lyo Code Generator: A Model-based Code Generator for the Development of OSLC-compliant Tool Interfaces," SoftwareX, 2016.

[18] (2016, Nov.) Eclipse EMF. [Online]. Available: https://eclipse.org/modeling/emf/

[19] (2016, Nov.) Eclipse Acceleo. [Online]. Available: https://www.eclipse.org/acceleo/

[20] MOF Model to Text Transformation Language (MOFM2T), 1.0, OMG standard, document number: formal/2008-01-16, 2008.

[21] (2016, Nov.) Eclipse Lyo. [Online]. Available: https://www.eclipse.org/lyo/

[22] J. El-Khoury, C. Ekelin, and C. Ekholm, “Supporting the Linked Data Approach to Maintain Coherence Across Rich EMF Models,” Modelling Foundations and Applications: 12th European Conference, ECMFA 2016, pp. 36-47.

[23] R. Basole, A. Qamar, H. Park, C. Paredis, and L. Mcginnis, "Visual analytics for early-phase complex engineered system design support," IEEE Computer Graphics and Applications, vol. 35, nr. 2, 2015, pp. 41-51.

[24] Ontology Definition Metamodel, OMG standard, document number: formal/2014-09-02, 2014.

[25] OWL 2 Web Ontology Language, W3C Recommendation, 2012.

[26] (2016, Nov.) Enterprise Architect ODM MDG Technology. [Online]. Available: http://www.sparxsystems.com/enterprise_architect_user_guide/9.3/domain_based_models/mdg_technology_for_odm.html

[27] Shapes Constraint Language (SHACL), W3C Working Draft, 2016.

[28] M. Biehl, J. El-khoury, F. Loiret, and M. Törngren, "On the modeling and generation of service-oriented tool chains," Software & Systems Modeling, vol. 13, nr 2, 2014, pp. 461-480.

[29] S. Lohmann, S. Negru, F. Haag, and T. Ertl, “Visualizing Ontologies with VOWL,” Semantic Web, vol 7, nr 4, 2016, pp. 399-419.

[30] J. Bārzdiņš, G. Bārzdiņš, K. Čerāns, R. Liepiņš, and A. Sproģis, "UML style graphical notation and editor for OWL 2," International Conference on Business Informatics Research, Springer Berlin, 2010.

[31] S. Krivov, F. Villa, R. Williams, and X. Wu, "On visualization of OWL ontologies," Semantic Web, Springer US, 2007, pp 205-221.

[32] M. Lanzenberger, J. Sampson and M. Rester, "Visualization in Ontology Tools," International Conference on Complex, Intelligent and Software Intensive Systems, 2009, pp. 705-711.

[33] A. I. Wasserman, "Tool integration in software engineering environments," the international workshop on environments on Software engineering environments, 1990, pp. 137-149.