This document is downloaded from the
Digital Open Access Repository of VTT

Title	Ontology-driven software engineering: Beyond model checking and transformations
Author(s)	Artem Katasonov
Citation	International Journal of Semantic Computing Vol. 6 (2012) No: 2, Pages 205 - 242
Date	2012
URL	http://dx.doi.org/10.1142/S1793351X12500031
Rights	Electronic version of an article published as International Journal of Semantic Computing, Volume 6, Issue 2, 2012, Pages 205 - 242, Doi:10.1142/S1793351X12500031, ©World Scientific Publishing Company, http://www.worldscientific.com/worldscinet/ijsc

VTT
http://www.vtt.fi
P.O. box 1000
FI-02044 VTT
Finland

By using VTT Digital Open Access Repository you are bound by the following Terms & Conditions.

I have read and I understand the following statement:

This document is protected by copyright and other intellectual property rights, and duplication or sale of all or part of any of this document is not permitted, except duplication for research use or educational purposes in electronic or print form. You must obtain permission for any other use. Electronic or print copies may not be offered for sale.

Ontology-Driven Software Engineering: Beyond model checking and transformations

Artem Katasonov

VTT Technical Research Center of Finland, 33101, Tampere, Finland

Abstract

This paper introduces a novel framework for Ontology-Driven Software Engineering. This framework is grounded on the prior related work that studied the interplay between the model-driven engineering and the ontological modelling. Our framework makes a contribution by incorporating a more flexible and means for ontological modelling that also has a higher performance in processing, and by incorporating a wider range of ontology types into ODSE. In result, it extends the power and speed of the classification and the model consistency checking ontological services enabled by the prior work, and brings new ontological services: semantic search in model repositories, three kinds of semi-automated model composition services: task-based, result-based, and opportunistic, and the policy enforcement service. The primary intended use for this framework is to be implemented as part of model-driven engineering tools to support software engineers. We describe our reference implementation of such a tool called Smart Modeller, and report on a performance evaluation of our framework carried out with the help of it.

Keywords: Ontology-driven software engineering, Model-driven engineering, Component metadata, Software composition, Model checking

1. Introduction

Ontology-Driven Software Engineering (ODSE) has been gaining a growing attention in recent years. ODSE, in a wide sense, refers to various ways in which ontologies and other semantic technologies can contribute to improving the software engineering, both its processes and its artefacts. As ontologies are formalized conceptual models of some domains, ODSE is often seen as a kind, or an extension, of a more established trend that is Model-Driven Engineering.

Model-Driven Engineering (MDE) [1] plays today a key role in describing and building software systems [2]. The MDE approach is based on increasing the level of abstraction to cope with complexity. In MDE, software models are used not only for design and maintenance purposes, but also as a basis for generating the final executable code. MDE attempts to insulate business applications from technology evolution through increased platform independence, portability and cross-platform interoperability, encouraging developers to focus their efforts on domain specificity [3]. The MDE approach finds support also from the emergence and popularization of the open-source development and the software-as-a-service paradigm. Both these trends lead to a situation where software applications are increasingly created with utilization of existing components, while defining the control and data flows between such components in a graphical modelling tool often appears to be an intuitive way to proceed.

MDE suggests to develop some models describing a system in an abstract way and then to transform them, in several steps, into real, executable systems (e.g. source code) [2]. The key concepts of MDE are thus models, modelling languages, and transformations. Modelling languages can be generic, such as UML, or be *Domain-Specific Languages (DSL)*; different languages can be used for different levels of models. Transformations may be performed along the model specification dimension [2] where an original coarse model, e.g. from a business viewpoint, is progressively refined into more detailed models, e.g. from an expert viewpoint, a developer viewpoint. One standardized view on MDE, which is Model Driven Architecture (MDA) [4] by the Object Management Group (OMG), suggest also transformations along the platform specification dimension. MDA process, e.g. as in [5], involves the following steps. First, a *Computation-Independent Model (CIM)* is created, which could be e.g. a UML's use case diagram. Then, based on CIM, the *Platform-Independent Model (PIM)* is developed, which can include UML's class diagram, state-transition diagrams, and so on. In the next step, PIM combined with a *Platform Profile* is transformed (at least some automation is assumed) into the *Platform-Specific Model (PSM)*, which is tailored to and provides more information for a particular operating system, particular programming language, and so on. Finally, the programming code is generated from PSM.

There are several recognized issues with MDE. One issue is that transformations between models, especially along the platform specification dimension, is a hard problem which is difficult to automate or even provide support for. Another issue is that in MDE the semantics of modelling languages, i.e. denotational, operational or axiomatic interpretation of their constructs, is usually not defined explicitly. This semantics is either hidden within the modelling tools [2] or even left to the people

Email address: artem.katasonov@vtt.fi (Artem Katasonov)

that design and use the models. The consistency of models is often analysed using procedural checks of the modelling tools, providing mostly superficial verification. Absence of semantics defined outside the tools further hinders automated transformation of the models, unless both the source and the target models are managed within the same tool.

In response to these issues, the Ontology-Driven Software Engineering approach was recently introduced, which includes the use of ontologies and other semantic technologies as part of the MDE process. Most existing ODSE approaches focus on providing *ontological services* to support traditional MDE tasks such as model transformations and model consistency checking. We believe, however, that the great potential of ODSE goes well beyond the improvement to traditional MDE tasks. ODSE can provide support to software engineers with respect to new tasks touched little by MDE, most notably semi-automated construction of software models.

In this paper, we present a novel ODSE framework. It is grounded in the viewpoint of e.g. [2] that studied the interplay between the model-driven engineering and the ontological modelling and formalised some ontological services such as the classification service and the model consistency checking service. The contribution of our framework is then two-fold. First, it incorporates a more flexible and means for ontological modelling that also has a higher performance in processing, which is a hybrid of Description Logic and SPARQL patterns. In result, the expressiveness as well as the speed of execution of e.g. model checking is improved. Second, it incorporates a wider range of ontology types into ODSE and enables new ontological services: semantic search in model repositories, three services for semi-automated construction of models: task-based, result-based, and opportunistic, as well as the policy enforcement service. An important fact is that this framework provides a common uniform basis for both the traditional ontological services and the new ones.

A brief presentation of an earlier version of our ODSE framework was published in [6]. However, the framework has evolved since that publication and, therefore, this paper replaces rather than just extends it. Our framework was exploited in an ODSE tool called Smart Modeller, which is one of the products of the research project SOFIA [7]. We will refer to the Smart Modeller several times in this paper. However, as this paper focuses on the framework itself, technical details of this tool and description of its modelling language are outside the scope of this paper. More information on Smart Modeller can be found in [8, 9].

The rest of the paper is structured as follows. Section 2 discusses ODSE, the advantages of ontologies it currently fails to exploit, and argues for new ontological services to enable. Sections 3, 4, and 5 discuss the three topical areas that form the background for our framework. Section 3 discusses the bridging of modelling languages with ontologies, as this is the cornerstone of our framework the same way as it is the cornerstone of most other ODSE approaches. Section 4 discusses SPARQL patterns as an alternative approach to representing ontological knowledge, and the advantages it brings. Section 5 discusses different types of knowledge, for which ontological modelling is of benefit in the context of ODSE. After that, Section 6 describes our ODSE framework in detail. Section 7 briefly describes our reference implementation of the framework in the Smart Modeller tool. Section 8 reports on a performance evaluation of the framework. Finally, Section 9 concludes the paper.

2. Ontology Driven Software Engineering

2.1. State of the art

One early work related to ODSE is [10], where the authors discussed the similarities between model engineering and ontology engineering and introduced the concept of *bridging* the two (see more on this in Section 3). Following this concept, [11] provided a bridging between the Ecore metamodeling language, which is a part of the Eclipse Modelling Framework (EMF) [12], and the Web Ontology Language (OWL), which is an ontology-definition language based on Description Logic (DL). Then, the authors proposed semi-automatic creation of OWL ontologies representing and extending Ecore-based metamodels (i.e. modelling languages). Finally, the authors discussed that, after defining an explicit mapping between two ontologies created for two different modelling languages, the production can be facilitated of the code for automatic transformation of a model in one language into the corresponding model in the other language.

[13] also targeted generation of code for performing transformations between models in different modelling languages. In contrast to [11], i.e. to creating different ontologies for different languages and then mapping them, [13] relied on binding, through semantic annotations, of each metamodel to a common reference ontology.

Instead of model transformations, [2] focused on ontological services that a designer of a single model can receive if an ontology representing the modelling language metamodel is created. These services include model consistency checking, i.e. checking if the model contains any contradictory facts with regard to the ontology, and model refining, where a generic element with some particular properties or connections is substituted with a more specific matching element. Both services are provided through a reasoning process over the ontology and the model in question. This work also introduced an approach where the ontology is not maintained as a complete and separate document, but rather OWL annotations are embedded into the textual representation of the modelling language metamodel. The full OWL ontology is then automatically produced when needed. A related work [14] provided a case study for this approach in the network configuration software domain.

[15] discussed ontology-driven consistency checking in cases where there are more than two modelling levels, i.e. model has to conform to a metamodel while the metamodel has to conform to a metametamodel. Such a checking process was demonstrated based on the authors' extension to OWL.

Although the ODSE term commonly refers to the use of Description Logic based frameworks such as OWL, comparable solutions exist also based on different formalisms, most notably the Object Constraint Language (OCL). OCL-based solutions

target similar goals as DL-based ones, with recent work focusing on verification of models [16], model transformations [17], and checking of conformance between models at different levels of abstraction [18]. A study [19] also attempted to integrate OCL with OWL in a way that OCL constraints, pre and post conditions can rely on OWL class definitions, to increase the expressiveness and in so to improve model checking processes.

In addition to approaches that are ontology-driven in the sense that they use an ontology representing the modelling language, some ODSE approaches attempt to exploit as a part MDE the ontologies describing the software application domains. Several works, e.g. [20], propose the transformation of such a domain ontology directly into the software application's hierarchy of classes as represented, e.g., by a UML class diagram. In terms of MDA (see Section 1), this means that the domain ontology becomes a part of the Computation Independent Model (CIM) level and is utilized for generating some parts of the Platform Independent Model (PIM) [21], resulting in some level of automation also in this transformation step.

As the above overview of ODSE research indicates, and to our best knowledge, the work has been focused and restricted to supporting traditional MDE tasks, i.e. model checking, model transformation, and model refining, which is in turn a kind of local transformation.

In fact, an earlier article [22] by the same authors as above mentioned [13] discussed on a conceptual level the bridging MDA and ontologies and introduced, in addition to model transformations, also some other potential applications of ontologies in MDA. One is search and composition of components: semantic descriptions of, e.g., commercial-off-the-shelf components or services can be stored in a knowledge base where ontological reasoning and matchmaking mechanisms can be applied. The other one is automated composition of models: semantic descriptions of components can be exploited for composing models, for example, when creating a platform specific model based on a platform independent one. To our knowledge, the authors did not pursue these directions further.

The ontology applications like search and matchmaking are discussed more often outside the MDE community. Generic reviews of the relationship between ontologies and software development [23, 24] discuss, similarly to [22] above, discovery of appropriate software components based on semantic descriptions of them. There are a number of practical studies using ontologies in this way. [25] discussed online repositories of software components, from which components are discovered based on metadata. [26] studied the use of semantic metadata in application servers for a variety of reasons, including search of components and services and conflict resolution between them. Several studies [27, 28] also suggest that semantic metadata can provide a way of deciding on *compatibility* of software components and thus give an input for the process of composing applications from components.

These studies, however, do not follow a formal MDE approach and, therefore, their solutions do not enjoy modelling-related ontology benefits such as consistency checking. In contrast, our work follows the MDE tradition but extends it. As stated in Section 1, our ODSE framework is designed to provide a common uniform basis for both the traditional ontological services, such as model checking, and new ontological services such as search and composition of models.

2.2. New ontological services

Following the discussion started in the later part of Section 2.1, we argue here for new ontological services to become an integral part of a formal ODSE framework.

Some of the issues, which these new ontological services aim at addressing, are explained by the following simple example, which we will use throughout the paper where appropriate. Software development for the present Nokia smart phones, both Symbian and Maemo/MeeGo, is commonly done using Qt C++. The Qt environment includes the Qt Mobility library that provides the applications access to phone's common resources, such as call service, messaging, contacts, media, GPS, sensors, and so on. Qt Mobility is divided into a set of separate Application Programming Interfaces (APIs), e.g. Messaging API and Contacts API, which are designed not to have cross-dependencies, i.e. they do not use the classes of each other. The Contacts API has, among other functions, a function that searches for contacts matching a given filter and returns results in the form of QContact objects. The Messaging API has, in turn, a function that opens the phone's standard dialog for composing and sending a short message to a given contact. The latter function does not expect a QContact object as the input, but just a string representation of a phone number. Now consider an application "find a contact, then compose and send a message to him/her". The above two functions can obviously be used in a chain and jointly enable such an application. Let us call them semantically compatible. However, on the syntactic level, there is nothing indicating such compatibility: both the data types and the names of the parameters are different for the output of the former and the input of the latter. Therefore, it is the application developer who has to possess the knowledge of such semantic compatibility and also of the *adapter* that has to be used. In this case, the adapter is a line of code that extracts the string representation of the phone number from a QContact object ("contact" object in the snippet):

```
((QContactPhoneNumber) contact.detail(
        QContactPhoneNumber::DefinitionName)).number()
```

Note that the above example is about using together two parts of substantially the same library. The problem becomes much harder when attempting to use together different APIs from different vendors with different original purposes. Nowadays, programmers and developers who want to (re)use existing components and services have to either search for those components manually or perform lookups which are based on syntactical descriptions [22]. In practice, a lot of effort goes into finding appropriate components or API functions, reading documentation about them, searching for examples, understanding how to use the components together, e.g. to adapt the output of one to fit into the input of another.

Ontologies and ontology-based semantic data are one way of representing knowledge. They allow some knowledge to be

explicitly captured and, in so, to remove the need for a human to possess this knowledge and to allow a computer-based system to automatically utilize this knowledge. With respect to the problem above, an ontological approach can thus enable automatic discovery of compatible components / API functions as well as automatic discovery and incorporation of adapters. Roughly speaking, it is like software documentation that is machine-processable and that can automatically be put into use.

When combined with MDE, this means in short the following. Components and API functions are modelled using a DSL, these models are extended with semantic metadata, and both the models and the metadata are published into some repositories or to Web. Then, when designing a particular application model, based on the designer input and the current model, those components models are searched for, matched, and incorporated into the application model. In result, we have not just machine-processable documentation of software components, but documentation that is automatically analysable *in the context* of the current model. Enabling this is an important goal of our ODSE framework.

Based on these considerations, we identified the following new ontological services to become a part of ODSE:

Semantic search in repositories. Models corresponding to software components or their compositions are stored along with semantic annotations in local or online repositories, and are automatically discovered based on those annotations. The main reason for this service is that it is needed as an enabler for model composition services below. A designer can also use it directly as a re-use mechanism.

Task-based model composition. Models of components are discovered based on high-level tasks they realize and then incorporated into the model. This is the simplest application of the semantic search service. As one option, it can be triggered by a direct input from a designer. For example: a model designer browses defined tasks, selects "Search contact", and receives the list of implementations including search in Outlook contacts, search in mobile phone directory, and so on, also for different target platforms and implementation languages. Alternatively, such a high-level task can be a part of a computation-independent or a platform-independent model. Then, the task-based composition service would be an enabler for the transformation of this model into a platform-specific model. The main reason for such a service is to reduce the need for possessing platform-specific knowledge.

Result-based model composition. Model designer specifies what kind of resource he/she would like to be produced as a part of the designed software. Then, models of components annotated as producing this kind of resource are searched for. After that, the current model is searched for possible suppliers of the inputs required by those components. Adapters are also discovered, if needed. It is a kind of a backward-chaining reasoning. For example: locate all functions that produce a QContact and analyse their required inputs. The main reason for such a service is to reduce the need for possessing API- or components-specific knowledge. This can be useful also simply because, in our experience, the process of understanding how to obtain a certain object is often painful in modern APIs as they often utilise factories and other indirect means of creating objects rather than constructors.

Opportunistic-based model composition. The current model is analysed with respect to what resources are being available, and after that models of components that can utilize/consume those resources are searched for. It is a kind of a forward-chaining reasoning. For example: the current model includes a "search contact" function and, therefore, has a Contact resource available, so the "compose SMS" function is proposed. The reasons are both to reduce the need for possessing API- or components-specific knowledge and to support opportunistic decision making. We use the term "opportunistic" in a sense similar to [29, 30].

Policy enforcement. Policies defined in relation to the designed software are checked, automatically or on-demand. This service is in many aspects similar to the model checking service but operates on a higher level: on policies about allowed model compositions, not the modelling language as such. One option is automatic checking after every change to the model. If the change leads to a situation prohibited by a policy, one possible response of the service is to cancel this last change while informing the designer about the policy. For example: a policy may state that applications are not allowed to persistently store a user's private data.

3. Bridging models and ontologies

This section discusses in more detail the bridging of modelling languages with ontologies and benefits that steam directly from this, as this is the cornerstone of our ODSE framework the same way as it is the cornerstone of most other ODSE approaches.

Modelling can be conceptually described using the four-layer architecture [31]. A model itself and a part of the world it represents are referred to as *M1* and *M0* layers, respectively. An M1 model is constructed using some kind of a modelling language that consists of an abstract syntax, one or more concrete syntax (graphic or textual), and semantics. The explicit definition of the abstract syntax of a language is referred to as its metamodel and placed at the *M2* layer. Each metamodel at the M2 layer determines how expressive its models can be. Analogously, a metamodel is defined by using concepts given in a metametamodel at the *M3* layer. Although one could continue the analogy to even higher layers, in practice it is common to use a standard M3-layer metametamodel, most commonly a derivative of the Meta-Object Facility (MOF) defined by Object Management Group (OMG). One popular MOF derivative is the Ecore metamodeling language that is a part of the Eclipse Modelling Framework (EMF) [12]. Henceforth, we will consider Ecore for the M3 Layer.

Ecore defines a small set of constructs including EClass, EAttribute, EReference, and EDataType. Using these constructs, a designer of a modelling language can define a M2-layer metamodel, which, for example, could describe such classes as ActionNode and ControlFlowEdge along with their attributes and references. Using these M2-layer constructs, a

user of the modelling language can then create a M1-layer process model consisting of a set of specific actions and control flow connections between them. This process model would then act as a representation of some M0-layer existing or planned process.

If two different modelling languages are based on the same metametamodel (e.g. Ecore), it is possible to formally define the transformation operation between these languages. Then, one is able to perform automated transformation of a model based on one of the two languages into a model based on the other one.

It can be noticed that the world of the semantic technology follows the same four-layer architecture. Assume there is a part of the real world (M0) and a semantic description of it (M1). The M1-layer semantic description uses concepts from one or more ontologies, which are M2-layer metamodels. Such ontologies are often explicitly defined using RDF-Schema (RDF-S) and Web Ontology Language (OWL), which are therefore M3-layer metametamodels.

One way to refer to M1 and M2 layers here, as e.g. in [2], is ABox (assertions) and TBox (terms) of a knowledge base. In this paper, we assume that M1 semantic description is based, as it is common in practice, on the Resource Description Framework (RDF). Therefore, instead of speaking of ABox and TBox, we prefer to speak of RDF and ontologies. Note also that when speaking of "RDF" we mean narrowly a description of a set of individuals and relationships between them, i.e. an M1-layer model. Unless explicitly specified, we do not mean higher-layers uses of RDF, such as RDF serialization of OWL ontologies.

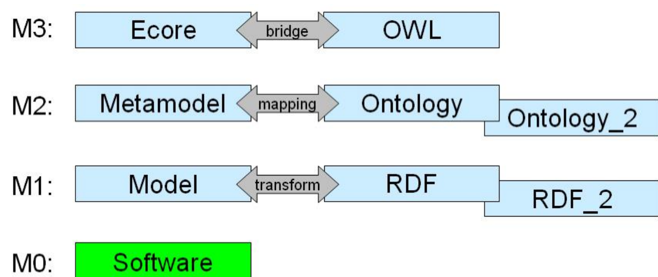

Figure 1: Bridging modelling languages and ontologies.

Several studies [11, 2] note that although the metametamodels behind OWL and Ecore are not the same, there are sufficient similarities, and, therefore, one can define a bridge between the two. Such a bridge treats Ecore concepts as special cases of OWL concepts: EClass as a subclass of owl:Class, EAttribute as a subclass of owl:DatatypeProperty, and EReference as a subclass of owl:ObjectProperty. Note that both owl:DatatypeProperty and owl:ObjectProperty are in turn subclasses of rdf:Property. With such a bridge defined, one can then define a transformation operation between an Ecore-based modelling language and an OWL-based ontology. As a result, one can perform automated transformation of a model based on that modelling language into an RDF-based model following the ontology. Figure 1 depicts this approach, which e.g. in [2] is referred to as "OWLizing".

The basic motivation for such a transformation is that ontology languages like OWL offer a greater expressiveness than metamodeling tools like Ecore. An OWL ontology can define not only the abstract syntax, but also at least a part of the axiomatic semantics of a modelling language, using the framework of Description Logic (DL).

One approach is to maintain the needed OWL ontology as a complete and separate document. Another approach is to embed OWL annotations into the textual representation of the metamodel of the modelling language, and then to automatically and on-demand transform this extended metamodel into the full OWL ontology. For example, [14] used the latter approach.

The practical motivation then is to provide a model designer with a set of *ontological services*. [2] discussed a few such services:

- *Consistency checking* - to check automatically if the model contains any contradictory facts with regard to the ontology. Such contradiction would indicate that the model does not fulfil a restriction set by the ontology and is therefore either erroneous or incomplete. A check can also return an explanation of the inconsistency facilitating debugging of the model.

- *Classification* - to determine all the types, or the most specific type, a model element belongs to according to the ontology. This service facilitates refinement of models, including automated, by suggesting suitable concepts to be used in a refined model.

- *Querying* - to provide the designer with an ability to query the model more flexibly than normally provided by modelling tools. For example, one can search for all the elements of a certain type that are connected to elements of some other type in a certain way. This service does not necessarily use the ontology definition but rather enjoys the direct benefits of having the model represented in RDF.

[2] also discussed a couple of guidance services that apply to the ontology itself: *satisfiability checking* to check the consistency of the metamodel, and *subsumption checking* to compute/refine the hierarchy of defined classes and as a result to possibly refine the metamodel.

In Figure 1, we included Ontology_2 and the corresponding RDF_2 model to stress the fact that more than one ontological representation for a given modelling language is possible, and also to point out that at least two such representations are straightforward for Ecore-based languages. When using Ecore to design a graph-based modelling language (that has nodes and edges), it is common to define an edge as an EClass in its own right with two EReference, usually called "source" and "target". The most direct OWL representation will map the edge EClass into a corresponding owl:Class with the source and the target as two owl:ObjectProperty. Another OWL representation, however, is possible that maps an edge directly into an

Table 1: Examples of OWL-expressible semantics

Restriction type	OWL	Meaning
Required attributes	Action: *SubClassOf*: name *some* string	An Action node should have a name defined.
Allowed values	Action: *SubClassOf*: visibility *only* {'public','private','package','protected'}	The visibility attribute of an Action node must be one of defined four values.
Required connections	FinalNode: *SubClassOf*: incoming *some* ControlFlowEdge	Final node must have some incoming control edges.
Disallowed connections	FinalNode: *SubClassOf*: *not* outgoing *some* Edge	Final node must have no outgoing edges.
Cardinality of edges	ForkNode: *SubClassOf*: outgoing *min* 2	A fork node should have at least two outgoing edges.
Interaction of edges	ForkNode: *SubClassOf*: incoming *some* ControlFlowEdge *and* outgoing *only* ControlFlowEdge *or* incoming *some* ObjectFlowEdge *and* outgoing *only* ObjectFlowEdge	The edges of a fork node must be either all control flows or all object flows.
Qualification of edges' sources and targets	FollowupAction: *SubClassOf*: Action *and* incoming *some* (source *some* Action)	FollowupAction is an Action node that directly follows another action without any control nodes in between. This is not a basic UML node type – included as an example of OWL modelling capabilities.

owl:ObjectProperty, or a set of those in a case where edges carry an attribute like "type" to specify more precisely the kind of an edge. The OWL definitions will be shorter and reasoning somewhat faster in the latter approach. The drawback, however, is that any additional attributes of the edges, if any, cannot then be represented. Note that in the former approach, it is mandatory for the ontology to include the inverse properties of "source" and "target", for instance called "outgoing" and "incoming".

Table 1 provides some examples of OWL definitions aimed for exposing the semantics of a modelling language. It uses the UML Activity diagram as the subject language, presents OWL declarations in the compact Manchester syntax, and assumes that edges are modelled as OWL classes. The first six definitions are directed mostly to the consistency checking service that would treat them as constraints to the model. The last definition is directed mostly to the classification service. This service may use such a definition to refine generic elements into more specific ones, or to locate elements for which some additional constraints must be checked.

We must comment, however, that using OWL for implementing the model checking service is not as straightforward as it may sound, and this issue is not discussed in [2]. OWL is meant to infer new knowledge rather than to act as a schema language. In particular, OWL makes an Open World Assumption (OWA), which means that absence of a fact does not imply its falsehood, and standard OWL reasoning engines follow OWA as well. This means that while the second, the fourth, and the sixth restriction in the Table 1 will trigger inconsistency errors if the model does not satisfy them, the first, the third, and the fifth will not. For checking those, an indirect approach has to be used: one attaches a restriction to a separate class, say, ValidAction, and then checks if the sets of Action and ValidAction are the same. Such an approach would be in turn sufficient for all the restrictions in Table 1 except of the fourth (the one with *not some*), for which, again because of OWA, it would not work and always give an empty set. In result, one has to always use a combination of both verification means.

In the column "meaning" of Table 1, we used words "must" and "should" to distinguish between cases of definite error and probable error. As OWL does not have means for specifying this part of the constraint semantics, it still can only be embedded and thus hidden inside the modelling tools.

4. Using SPARQL patterns

In present, OWL is a dominant framework for encoding ontological knowledge. For MDE, as we discussed in Section 3, OWL provides some useful expressive power and, therefore, enables realization of some ontological services. In 2009, OWL 2 specification was finalized that added some new useful features to the language. Yet, being a closed framework, OWL has its limitations. [32] puts it as that OWL is hard-coded against specific design patterns, but anything that goes beyond those patterns cannot be expressed.

In Section 3, we were defining an OWL ontology with the purpose of reasoning over data encoded as RDF, in particular. Also, the reasoning tasks were limited to pattern-based matching: the consistency checking service basically checks if a certain data pattern occurs where it should, while the classification service searches for all occurrences of a certain data pattern. It can be noticed that at least as long as these two conditions (RDF and reasoning as matching) hold, there is an alternative way of encoding ontological knowledge – SPARQL-based patterns. SPARQL is the W3C standard for querying RDF data. As SPARQL 1.1 is now in works (currently a working draft yet already supported by some tools), the whole set of OWL 2 features becomes expressible in SPARQL. Most importantly, SPARQL 1.1 introduces the aggregate function COUNT that enables expressing OWL cardinality constraints. For example, [32] discussed this relationship between SPARQL and OWL

Table 2: Examples of SPARQL 1.1 patterns

Restriction type	SPARQL
Required attributes	*SELECT* ?this *WHERE* {?this :name ?name}
Allowed values	*SELECT* ?this *WHERE* {?this :visibility ?v. *FILTER* (?v = 'public' || ?v = 'private' || ?v = 'package' || ?v = 'protected') }
Required connections	*SELECT* ?this *WHERE* {?this :incoming [*a* :ControlFlowEdge] }
Disallowed connections	*SELECT* ?this *WHERE* {?this *a* :Node. *FILTER NOT EXISTS* { ?this :outgoing ?edge} }
Cardinality of edges	*SELECT* ?this *WHERE* { ?this :outgoing ?edge } *GROUP BY* ?this *HAVING (*COUNT*(?edge) >= 2)*
Interaction of edges	*SELECT* ?this *WHERE* { {?this :incoming [*a* :ControlFlowEdge]. FILTER NOT EXISTS {?this :outgoing [*a* :DataFlowEdge]} } *UNION* { ?this :incoming [*a* :DataFlowEdge]. FILTER NOT EXISTS {?this :outgoing [*a* :ControlFlowEdge] } } }
Qualification of edges' sources and targets	*SELECT* ?this *WHERE* {?this *a* :Action. ?this :incoming [:source [*a* :Action] }

and also commented on the strengths of SPARQL compared to the limitations of OWL 2. Table 2 shows how the example restrictions from Table 1 are represented as corresponding SPARQL patterns. A resource belongs to the class defined with a SPARQL pattern if it is (or would be) returned as a possible value for *?this* variable.

Using SPARQL in place of OWL has some advantages. Some of these advantages are purely technical, but some are related to a higher expressive power. Let us first comment on the technical ones:

- Although SPARQL patterns may look longer than corresponding OWL definitions, SPARQL is optimized for execution. As our evaluation in Section 8 indicates, the difference in performance is significant and also grows fast as the complexity of the model or of the ontology increases.
- One does not need a separate reasoner, but just an RDF data storage (persistent or in-memory) supporting SPARQL querying (most do).
- [14] reported that one of the problems related to using OWL for model consistency checking is that available reasoners stop upon discovery of the first inconsistency rather than find and report all of them. This is not a problem when using SPARQL patterns, as one execution always returns all possible answers to the query.
- As already mentioned in Section 3, using OWL for implementing the model checking service is somewhat tricky, due to OWL's Open Words Assumption.

Most importantly, however, SPARQL provides expressive features that are very useful in the context of ODSE yet unavailable in OWL. First, consider the following SPARQL pattern:

```
SELECT ?this WHERE {
    ?this :visibility ?v. ?this :outgoing [:target ?next].
    ?next :visibility ?v
}
```

The meaning of this is: an action followed by another action with the same visibility value. OWL 2 introduced the concept "self" to allow defining restrictions such that the subject and the object of a property must be the same, e.g. "self-regulating process". However, the pattern above cannot be expressed in OWL – it is impossible to define a requirement that two different properties, or property chains, have as their targets the same object or literal.

Second, the OWL is all about putting restrictions on the classes of RDF atoms, e.g. single model elements in case of ODSE. In contrast, SPARQL can be used for describing and putting restrictions on non-atomic RDF structures, e.g. compositions of model elements. Consider the following SPARQL pattern:

```
SELECT ?this_1 ?this_2 WHERE {
    ?this_1 a :Action. ?this_1 :outgoing [:target ?this_2].
    ?this_2 a :Action. ?this_2 :outgoing [:target [a :FinalNode]].
}
```

The meaning of this is: a *sequence* of two actions leading to the final node, i.e. a kind of a finalizing sequence. In OWL, one could refer to a member of such a sequence, but not the whole sequence as a composition of two elements.

Third, when describing some resource through its relationships to some other resources, OWL always hides those other resources behind "some" or "only" definitions. While this mechanism is also available in SPARQL as the blank-node [] syntax, one can also refer to those resources explicitly through variables. This then has a double utility. The first use is that one can add these variables into SELECT clause. Then, a consistency check or a classification run will return not only the resources in question but also the resources identified by those additional variables (e.g. attributes, edges, connected elements). This additional information can be used for providing better explanation of the results or for a follow-up analysis. The second use is that one can define a mechanism for extending the ontology itself, so that it would provide not only the restriction-based definitions of classes, but also some interpretation rules applicable to the instances of those classes. As in most rule-representation formalisms, variables provide the value links between the head and the tail of a rule. Section 6 will later describe such a mechanism as a part of our ODSE framework.

Fourth advantage we already mentioned in the end of Section 3. Not being constrained by the OWL framework, which

defines classes through restrictions, one can use a more flexible way of *linking* patterns to classes. Then, one is able to distinguish between patterns describing the actual constraints to the language (should raise errors) and patterns describing intended use of the language (should raise warnings). Then, one can even add patterns describing recommended modelling practices and so on.

One obvious inconvenience with using SPARQL in place of OWL is the following. As OWL reasoning is an iterative process, one can conveniently define a hierarchy of classes where every class is defined through some additional restrictions on its direct super-class (the rest of the restrictions is therefore inherited from the chain of super-classes). In contrast, each SPARQL pattern must be complete and defined in terms of the ground data on which the query will be run.

To mitigate this issue, our framework (see Section 6) is designed to be a hybrid of the straightforward SPARQL-patterns approach and the Description Logic (DL). The framework allows the ontology designer to define classes in terms of common DL operations such as intersection, union and complement, and to use some additional operations such as composition (see above on non-atomic structures). The framework then assumes the existence of a patterns pre-processing engine. Such an engine is a part of an ODSE modelling tool and is responsible for constructing the full pattern for a class before it is used in a query. In this sense, we do not replace OWL as a whole, but only its "ground" level, i.e. its mechanism for defining classes as restrictions (owl:Restriction class).

5. Extending the range of ontologies

Using an alternative means for ontological modelling, which utilizes SPARQL patterns as a part of it, is one novel feature of our ODSE framework. Another novel feature is that our framework relies on a wider range of knowledge being represented with ontologies, not only the ontology that is a translation of a modelling language itself (as in Section 3).

Based on the review of ontology classifications given in [23] and putting it in the context of ODSE, we can identify the following four vocabulary types, for which ontological modelling is of benefit in software engineering and which we attempt to explicitly utilize in our framework. Where appropriate, we use the concepts related to the example application introduced in Section 1.

- *Domain* – concepts, usually nouns, that describe the application domain of a software system, e.g. "contact", "message", etc.

- *Tasks* – concepts, usually verbs, that refer to platform-independent problem-solving tasks that exist in the domain, e.g. "search", "compose", "send", etc.

- *Software* – concepts used to refer to the software artefacts themselves, including the structural and the functional perspectives, e.g. "component", "function", "parameter", "transition", etc.

- *Interaction* – concepts used to describe the interaction between software and its environment/domain, e.g. "resource", "produces", "precondition", etc.

An ontological representation of a modelling language would often be restricted as an *ontology of software* (third vocabulary in the list above). In addition to UML and various DSLs with their metamodels, several generic metamodels have been developed for the purpose of describing the structure and behaviour (but not purpose) of software artefacts. The most elaborated among them is the Knowledge Discovery Meta-Model (KDM) [33] by Object Management Group (OMG). An ontological representation of KDM would be the best available example of a generic ontology of software.

However, it depends on the modelling language in question, especially if it is a DSL, whether and how much the corresponding ontology will reach into the three other areas. A DSL can possibly include some elements or connectors that belong to an interaction vocabulary. A DSL for an information-processing system could also possibly include some domain concepts like "request", "report" as modelling blocks when there is a wish to embed into them some semantics different from that of a generic "object". Abstract tasks as a vocabulary is probably covered the least by DSLs.

The *domain ontologies* are the most common kind of ontologies considered when developing an information-processing system. Domain ontologies are ones sometimes explicitly developed for specific applications that are subject to interoperability or extensibility requirements. As was mentioned in Section 1, it is also often proposed to exploit domain ontologies in the ODSE process through transforming a domain ontology into the software application's hierarchy of classes, e.g. [20]. Another possible application of domain ontologies in software engineering is related to the idea of providing semantic annotations for software components, often referred to as *component metadata* (which, however, is a kind of word misuse since components are not data). Such annotations can be used for component search or for checking compatibility between components. In present, in most practical cases where component metadata is utilized, this metadata is restricted to the concepts of an ontology of software only, as surveyed e.g. in [34]. However, some researchers [26, 28] suggest annotating the input and output parameters of components with the concepts from a domain ontology. For example, if one component has a string-type output parameter while another component has a string-type input parameter, this does not say much. On the other hand, if these both parameters are annotated with the "contact" concept from the same ontology, one can reasonably assume that they can be connected by a data flow.

The importance of defining *task ontologies* is argued in [24]. Also [26] advocated for *service taxonomies*, which follow a similar idea. The main goal is to be able to search for software components (for a certain platform) implementing a certain abstract task. It can also be noticed that parameterization of tasks with domain concepts, e.g. "search contact", "send message", is useful for further facilitation of the component search.

The concepts belonging to the last vocabulary type, interac-

tion, are more often embedded into larger ontologies than yield an ontology of their own. The WSMO ontology of Web Services includes such concepts as "precondition" and "effect" for describing capabilities of services [35]. [27] suggested using the same concepts for providing an abstract specification of the behaviour of any software component in order to improve the accuracy of component matching. One example of a separate interaction ontology is one of [36], which is an ontology for describing consumption, usage and production of *resources* by software, as well as properties of those resources. Among the goals is resource-based reasoning. Something produced in one software process, if it is shareable, can be used in another process. Also, if two software processes use the same resource, and that resource is known to be non-shareable, these two processes cannot be executed simultaneously. As to ODSE, component metadata using this ontology can facilitate the design of an application's control flow and can help in avoiding resource-based conflicts.

6. ODSE Framework

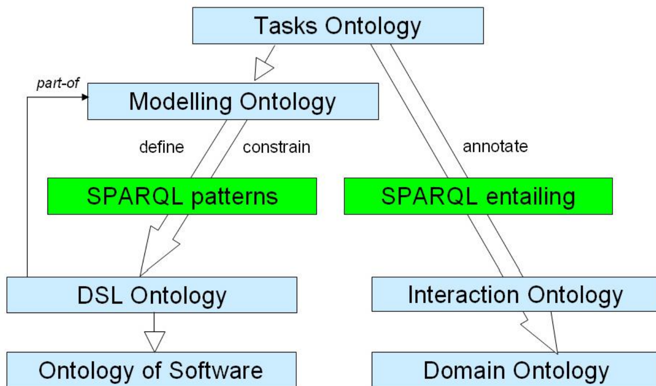

Figure 2: Ontological framework: Constituents.

Building on the background set in Sections 3, 4, and 5, this section presents our framework for Ontology-Driven Software Engineering.

6.1. Overview

Figures 2 and 3 provide a generic overview of our ODSE framework. Figure 2 depicts how various constituent ontologies are used and integrated in it. The arrow from DSL ontology to Modelling ontology represents the part-of relationship. The arrows from DSL ontology to Ontology of software and from Tasks ontology to Modelling ontology represent the special-case-of relationship. The wide arrows are used to provide indications how our framework combines ontologies. The arrow from Modelling ontology to DSL ontology represents the use of SPARQL patterns-based approach for defining and imposing constraints on modelling concepts. The arrow from Tasks ontology through Interaction ontology to Domain ontology represents the use of SPARQL entailing to annotate the tasks with interaction and domain concepts.

The starting point is an ontology of the modelling language, referred in the Figure 2 as the *DSL ontology*. As discussed in Section 3, such an ontology is basically a translation of the (e.g. Ecore) metamodel of the modelling language. As pointed out in Section 5, such a DSL ontology is a kind of an *Ontology of software*, i.e. it includes the concepts used to refer to the software artefacts themselves. In Smart Modeller that contains a reference implementation of the framework, the metamodel of the modelling language, and thus the DSL ontology, includes such concepts as Action, Parameter, Condition, Variable, Connector, and others. A DSL ontology may also include some concepts borrowed from a domain ontology, an interaction ontology, etc. In such a case, the ontologies will not be as clearly separated as in Figure 2. Such a separation, however, is neither required nor assumed in our framework.

The DSL ontology is then just a part of a bigger *Modelling ontology*. In addition to the concepts from the modelling language's metamodel, the Modelling ontology contains also more abstract concepts not explicitly represented in the metamodel. A straightforward usage is defining some specific subclasses of the generic DSL concepts. In Smart Modeller for example, the Modelling ontology contains such concepts as Input Parameter and Output Parameter (subclasses of Parameter), Constant (subclass of Variable), and others. The concepts from the Modelling ontology are defined in terms of DSL concepts using SPARQL-based patterns. Section 4 presented the background of this approach while Section 6.2 below will describe it in more detail. SPARQL-based patterns are used also for describing the validity constraints imposed on the Modelling concepts, both the DSL ones and more abstract ones (see below in Section 6.3). In contrast to prior work, e.g. [2], our framework separates between the definition of a class and the validity constraints imposed on this class.

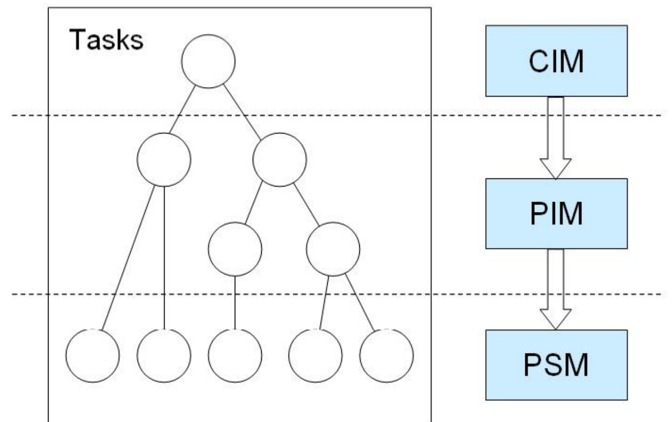

Figure 3: Ontological framework: A Tasks ontology and model levels.

A *Tasks ontology* is then treated in our framework as a special case of a Modelling ontology. Instead of drawing a clear line between the tasks (domain entities) and their implementations (software entities), we use an approach depicted in Figure 3. We assume that a hierarchy of tasks is defined that

stretches over the abstraction levels, such as MDA's CIM-PIM-PSM levels. In terms of our example from Section 1, there will be a computation-independent task like "search contact", then a more specific platform-independent task "search contact in mobile phone's contacts book", and a platform-specific "search contact with Qt Mobility Contacts API". Task ontologies act therefore in our framework as important facilitators for model transformations along the platform specification dimension. More specifically, our framework supports such transformation through the provision of the task-based model composition service (see Section 6.4).

The tasks are linked to the DSL concepts via SPARQL-based patterns the same way as any other modelling concepts. They can also have validity constraints defined, if needed. If different model levels use different modelling languages, the tasks at different levels of abstraction will be defined in terms of different DSL ontologies. Yet, the links between the tasks in the Tasks ontology will support transformations. In cases where the same modelling language is used for models at different levels, e.g. platform-independent and platform-specific, the framework allows for more abstract tasks not to have pattern-based definitions of their own. Then, they are treated as unions of all their known specific subclasses. Section 6.4 below will provide mode details on this.

Tasks can also be annotated with various properties from *Interaction ontologies* and through that can be linked to concepts from *Domain ontologies*. A mechanism for entailing used in such annotations is similar to that of the CONSTRUCT query-type of SPARQL. Sections 6.5 and 6.6 below will describe the parts of the framework related to such annotations. Finally, the framework also allows tasks to be referred to in the policy definitions. Section 6.7 will discuss this topic.

As can be noticed, our ODSE framework puts Tasks ontologies in the focus as a place where all kinds of ontological knowledge are integrated. This is principally different from any other ODSE approach we are aware of. Our framework can also be compared to existing frameworks for semantic component metadata (see Section 5). All such frameworks, we are aware of, follow a bottom-up approach. This means that physical software components are in the focus. These components, through some unique identifiers of them, are annotated using various properties, both low-level, e.g., parameters and more abstract, e.g., provided services, preconditions. In contrast, our approach can be seen as *top-down*. An abstract entity, namely a *task*, is in the focus. Specific physical software entities are then attached to tasks using a pattern-based approach. This implies that the unique identifiers of those entities do not have to be known or even exist. In other words, our approach follows closer the semantic technology view where it is assumed that specific resources can be pointed at through semantic properties of them, not only through their unique identifiers. We believe that this approach is both more flexible and powerful.

Following this overview, the following subsections will focus on notational aspects of the framework and describe ontological services enabled. As the primary intended use for this framework is to be implemented as part of model-driven engineering tools, the summary after each subsection is presented in the form of functional requirements to such MDE tools.

6.2. Defining modelling concepts

As mentioned several times above, our ODSE framework relies on SPARQL-based patterns as ontological definitions of the modelling concepts. The RDF representation for a Modelling ontology (and thus a Tasks ontology as well) is defined, however, to the extent possible to follow the Description Logic (DL) and to resemble the RDF serialization of OWL. The ontology designer is able to define modelling concepts in terms of common DL operations such as intersection, union and complement. A pre-processing engine, which has to exist as a part of an ODSE modelling tool, is then responsible for constructing the full pattern for a concept before it is used in a query.

An example definition is provided below. Here and henceforth, the notation for RDF is N3/Turtle [37]. The empty namespace is used for the Modelling ontology, *model:* for the DSL ontology, and *meta:* for the properties defined by the ODSE framework itself. Namespaces *owl:* and rdfs: are used for the standard properties of OWL and RDF-S, correspondingly. This definition describes an input parameter as a Parameter DSL element that has an outgoing Connector (a DSL element as well) towards an Action DSL element.

```
:InputParameter rdfs:subClassOf model:Parameter;
    owl:equivalentClass [
        owl:intersectionOf (
            model:Parameter
            [ meta:pattern
                "?c a model:Connector; model:source ?this;
                model:target [a model:Action]"
            ]
        )
    ] .
```

Table 3 describes all the defined properties for use in modelling concepts' definitions. The most straightforward way of defining a concept is by using the *meta:pattern* property with a literal value giving the concept's full SPARQL pattern. For the concepts from the DSL ontology, a tautological *meta:pattern* definition is assumed:

```
model:Parameter owl:equivalentClass
    [ meta:pattern "?this a model:Parameter" ]
```

The *owl:intersectionOf* property takes as the object an ordered list of classes (this is defined in the RDF serialization of OWL). The SPARQL pattern corresponding to the intersection of classes is constructed as the concatenation of those classes' patterns. However, it must be first assured that the constituent patterns do not use the same names for variables, with the exception of *?this*. Any detected variable conflict must be resolved by renaming variables. Since the predictability of variables naming is required for the entailing, this resolution process must follow a well-defined algorithm known to the ontology designers. An algorithm we use in Smart Modeller is given below. For example, if a variable *x* appears in the patterns of two classes, the second one will be renamed as *x2*; if *x2* was already taken as well, *x3* will be tried, and so on.

Table 3: Properties for defining modelling concepts

Property	Meaning
owl:equivalentClass	States that two classes are equivalent. Used for connecting a class to its definition.
meta:pattern	Defines a class with an explicit SPARQL pattern. Must use ?this variable.
owl:intersectionOf	Defines a class to be the intersection of several classes.
owl:unionOf	Defines a class to be the union of several classes.
owl:complementOf	Defines a class to be the complement of another class.
meta:base + meta:restriction	Defines a class to be another class (base) with one or more restrictions imposed.
meta:compositionOf	Defines a class to be a composition of several classes. Describes a non-atomic structure of more than one DSL element.

```
Let allVars be an empty list
For each pattern to intersect:
    Let vars be the list of all variable names in pattern
    For each var in vars:
        If var does not start with "?this":
            If allVars does not contain var: add var to allVars
            Else:
                Let i be 2
                While allVars contains var+i: i=i+1
                Substitute var with var+i in pattern
                Add var+i to allVars
```

The *owl:unionOf* property takes as the object an ordered list of classes as well. The SPARQL pattern corresponding to the union is constructed simply as {*pattern1*} UNION {*pattern2*} UNION {*pattern3*}, etc. No variable names resolution is needed. The SPARQL pattern resulting from *owl:complementOf* is also constructed without a need to do any variable renaming, simply as FILTER NOT EXISTS {*pattern*}.

Use of the *meta:base* property along with the *meta:restriction* property can be seen as "syntactic sugar" to allow more concise definitions. The base pattern is simply concatenated with all the restrictions without any variable naming checks. Restrictions will not normally be full class patterns. They do not have to refer to *?this*, they may just refer to some other variable existing in the base pattern.

The *meta:compositionOf* property is a special new operation in our framework, which logically corresponds to the Cartesian product operation on sets. The composition of two concepts is a concept all instances of which are pairs with one member being an instance of the first class and the other being an instance of the second class (e.g. a couple is a composition of a man and a woman). For example, the SPARQL pattern for a "finalizing sequence" example from Section 4 can be defined as (note that it is for UML Activity diagram, not for Smart Modeller DSL):

```
owl:intersectionOf (
    [ meta:compositionOf (
        :Action
        [ owl:intersectionOf (
            :Action
            [ meta:pattern
                "?this :outgoing [:target [a :FinalNode]" ]
        ) ]
    ) ]
    [ meta:pattern "?this_1 :outgoing [:target ?this_2]" ]
)
```

The second class in the composition list could, of course, have been defined separately as, e.g., FinalAction and then referred to by name. The SPARQL pattern corresponding to the composition of classes is constructed as a concatenation of those classes' patterns with renaming all the variables, including *?this*. In Smart Modeller, the algorithm followed is given below. It is very simple is a sense that it just appends to the names of all the variables in the first class pattern _*1*, the second _*2*, and so on.

```
For each pattern to compose:
    Let i be the index of pattern in compositionOf
    Let vars be the list of all variable names in pattern
    For each var in vars:
        Substitute var with var+"_"+i in pattern
```

The part of our ODSE framework described in this subsection poses the following requirements to ODSE tools implementing the framework:

- *Requirement 1.* Ability to transform the software model into an RDF representation that follows a defined DSL ontology.
- *Requirement 2.* Ability to process Modelling ontologies, to apply pattern operations defined, and to perform variable name resolution in order to construct full SPARQL patterns for the modelling concepts.
- *Requirement 3.* Ability to execute a resulting SPARQL pattern as a query against the RDF representation of the model to locate matching elements or composites of elements. This results in the provision of the *classification service* (see Section 3).

6.3. Adding validity constraints

As discussed in Section 3, one commonly considered ontological service is consistency checking (or validation) service. As also mentioned above, in contrast to prior work, e.g. [2], our framework distinguishes between the definition of a modelling concept and the validity constraints imposed on this concept.

The validity constraints are attached to the modelling concepts using *meta:constraint* property. For example, below we define that a Parameter DSL element should be used in a model as either an input parameter to an action, or an output parameter of an action. As noted in Section 6.2, in this example DSL,

the classification as an InputParameter requires presence of an outgoing connector to an Action DSL element. Similarly, the classification as an OutputParameter requires presence of an incoming connector from an Action element.

```
model:Parameter meta:constraint [
    a meta:WellformednessConstraint;
    owl:equivalentClass [
        owl:unionOf (
            :InputParameter
            :OutputParameter
        )
    ];
    rdfs:comment "A Parameter should act
        as either an input or an output of an Action"
] .
```

The definitions of constraints can use all the pattern operations available, see Section 6.2. The constraints of two types are distinguished: *meta:ModelConstraint* and *meta:WellformednessConstraint*. Model constraints are hard constraints that should yield validation errors. Alternatively, the ODSE modelling tool may even attempt to prevent such errors from being committed. This can be done is the form of ontology-driven disabling of some elements/attributes/connectors or informing the model designer of the error as soon as it is introduced into a model. Wellformedness constraints, on the other hand, should only yield warnings in validation and should not be attempted to be enforced. As discussed in Section 4, a wish to be capable of separating between different kinds of constraints is one reason why OWL as such was not considered sufficient for our framework. More constraint types can be defined if needed.

Realization of the model checking service according to our framework requires satisfaction of the requirements 1-3 from Section 6.2 and poses the following new requirement to ODSE tools:

- *Requirement 4.* Ability to perform the following process: (1) execute the concept definition pattern against the RDF representation of the model to obtain the set of elements to which the constraint applies; (2) in the same way, execute the constraint pattern to obtain the set of elements fulfilling the constraint; (3) report errors or warnings for those elements that are in the first set but not in the second set.

6.4. Defining tasks hierarchy

As mentioned above, instead of drawing a clear line between the tasks (domain entities) and their implementations (software entities), our framework rather suggest definition of a hierarchy of tasks that stretches from abstract computation-independent tasks till very platform-specific tasks that have one-to-one correspondence with particular implementations.

For a platform-specific task, a SPARQL-based pattern that corresponds to the task's realization (a model element or a set of elements that realize the task) is treated as the definition of the task. The classification service can therefore be used for checking if (the realization of) a certain task is a part of the current model and to locate all the occurrences of the task in the model. For platform-independent and computation-independent abstract tasks, the framework allows not to have pattern-based definitions of their own. Then, they are treated as unions of all their known subclasses. Then, the classification service can be used as well.

A simple example of a task hierarchy follows. The namespace *abstract:* is used here and henceforth for important modelling concepts which are, however, abstract and do not belong to a DSL ontology.

```
abstract:Task owl:equivalentClass model:Action.
abstract:TaskWithInput rdfs:subClassOf abstract:Task.
abstract:TaskWithOutput rdfs:subClassOf abstract:Task.
:SearchContact rdfs:subClassOf abstract:TaskWithInput,
                               abstract:TaskWithOutput.
:SearchContactOutlook rdfs:subClassOf :SearchContact.
:SearchContactPhone rdfs:subClassOf :SearchContact.
:SearchContactPhoneQt rdfs:subClassOf :SearchContactPhone.
```

The first definition is as in the Smart Modeller: a *Task* there is equivalent to the Action DSL element. *TaskWithInput* is a subclass of *Task* and is defined (not shown here) as an Action with at least one input Parameter connected to it, identified with *?input* variable. *TaskWithOutput* is defined analogously. Therefore, while being abstract, *Task*, *TaskWithInput*, and *TaskWithOutput* have proper pattern-based definitions. Assume then that *:SearchContact*, *:SearchContactOutlook* and *:SearchContactPhone* do not have such definitions of their own, and the ontology only states their place in the tasks hierarchy. Finally, assume that *:SearchContactPhoneQt* is specific enough to have a proper pattern again, for example as follows.

```
:SearchContactPhoneQt owl:equivalentClass [
    owl:intersectionOf (
        :TaskWithInput
        :TaskWithOutput
        [ meta:pattern "?this model:implementation
            \"ContactsActions::findContact\" "]
    )
] .
```

Note that platform-specific tasks like *:SearchContactPhoneQt* can nevertheless be further specialized, e.g., by requiring a specific value of an input parameter. It is also possible to define tasks that are sequences of two or more specific tasks using the *meta:compositionOf* operation, see Section 6.2.

The ability to locate tasks in the current model gains a practical value when combined with other features described below in Sections 6.5, 6.6, and 6.7. Our framework suggest, however, also the existence of a *Repository mechanism*. A repository means here an RDF data storage (document or database, local or online) where partial models are stored in an RDF representation that follows a defined DSL ontology. For example, Smart Modeller has such a mechanism and allows model designers to export partial models (any combination of elements and connectors) to repositories and then import then back into a model. Being in the first place a re-use mechanism, repositories also enable *semantic search service* through simple application of the classification service on a repository. If a repository contains partial models corresponding to implementations of some

specific tasks, executing a SPARQL pattern corresponding to a task as a query against the repository will locate all the matching implementations defined there. This enables *task-based composition* of models. For example, a model designer can browse a task ontology and select *:SearchContact* task. Then, he can receive a list of all matching implementations including one for *:SearchContactPhoneQt* and others if defined, select one and import it into the model.

The part of our ODSE framework described in this subsection poses the following additional requirements on ODSE tools implementing the framework:

- *Requirement 5.* Ability to construct the SPARQL pattern for a task, for which no explicit definition is given, as the union of all its known subclasses.
- *Requirement 6.* Existence of a repository mechanism that enables storing partial models in a defined RDF representation, as well as importing them into a model. This also requires the ability to transform RDF representation into the modelling language used (opposite direction as in Requirement 1).
- *Requirement 7.* Ability to apply the classification service (Requirement 3) not only to the model currently edited in the tool, but also to a repository.

6.5. Annotating task parameters

As mentioned in Section 5, some studies [26, 28] suggest annotating the input and output parameters of software components with the concepts from a domain ontology. Such annotations can be used for component search or for checking compatibility between components.

Our ODSE framework exploits such annotations, which are, however, attached to tasks, not directly to software entities (see Section 6.1). For example:

```
:SearchContactPhoneQt inter:produces
    [ inter:type nco:Contact;
    inter:format qtm:Contact; inter:maps ”?output” ].
:ComposeSMSQt inter:requires
    [ inter:type nco:MessagingNumber;
    inter:format qtm:PhoneNumberString; inter:maps ”?input” ].
```

The namespace *nco:* is used here for the concepts from NEPOMUK Contact Ontology [38], and *qtm:* for Qt Mobility concepts.

Please note that a task definition combined with an annotation of that task forms an entailing rule. Such a rule can be represented explicitly using e.g. CONSTRUCT form of SPARQL. The rule corresponding to the annotation of *:SearchContactPhoneQt* above would look like:

```
CONSTRUCT { ?this inter:produces [ inter:type nco:Contact;
        inter:format qtm:Contact; inter:maps ?output ] }
WHERE { pattern of :SearchContactPhoneQt
        introducing ?this, ?output }
```

We do not use this particular syntax for connecting rule's head and tail. One reason is because, with this syntax, rules would be just strings, not RDF fragments. Another reason is to separate tasks' definitions from annotations and to avoid repetition of definition patterns. Except for this, there is no difference between our representation and SPARQL, and that is why we generalized it in Figure 2 as ”SPARQL entailing”.

Table 4: Interaction-vocabulary properties

Property	Meaning
inter:produces	States that a task produces a shareable resource, e.g. a data item.
inter:requires	States that a task requires a resource, e.g. a data item, as one of its inputs.
inter:type	Defines the most specific semantic class of a resource.
inter:format	Defines the syntactic representation format of a data item.
inter:maps	Links a produced or required resource to an output or an input parameter of a task.
inter:partOf	States that a resource class is a part of another composite resource class.
inter:adaptFrom	States the source format for an adapter.
inter:adaptTo	States the target format for an adapter.
inter:precondition	Defines a precondition of a task execution.
inter:effect	Defines an effect of a task execution.
inter:prohibited	Defines a policy prohibiting a task for a certain class of actors.
inter:permitted	Defines a policy permitting a task for a certain class of actors.

Table 3 lists the interaction properties defined in our framework. Some of them are used already in the example above. The *:SearchContactPhoneQt* task, the pattern for which was provided in Section 6.4, corresponds to the Qt Mobility Contacts API function mentioned in Section 1. It searches for contacts matching a given filter and returns results in the form of QContact objects. The annotation above states that *:SearchContactPhoneQt* produces a data item that semantically corresponds to the *nco:Contact* concept. Then, it also stated that this data item is produced in a syntactic format identified as *qtm:Contact*, which means a QContact object. Finally, the produced data item is linked to the task's only output parameter through *?output* variable.

:ComposeSMSQt task corresponds to the Qt Mobility Messaging API function mentioned in Section 1. It opens the phone's standard dialog for composing and sending a short message to a given contact. It does not expect a QContact object as the input, but just a string representation of a phone number. Its annotation above states that the required input data item is semantically *nco:MessagingNumber* and syntactically *qtm:PhoneNumberString*.

A task producing a resource and a task requiring a resource are semantically compatible if these two resources belong to the same semantic class, or if the class of the produced resource is a subclass of the required resource, or if the required resource, or

a subclass or it, is a part of the produced resource. Syntactically, these two resources have to either use the same representation format, or there has to exist an adapter from the produced format to the required format.

Assume that a model contains an instance of *:SearchContactPhoneQt*. Assume also that we have the following definitions available as part of a tasks ontology:

```
nco:CellPhoneNumber rdfs:subClassOf nco:MessagingNumber.
nco:CellPhoneNumber inter:partOf nco:Contact.
ex:ExtractPhoneNumber rdfs:subClassOf abstract:Adapter;
    owl:equivalentClass [
        owl:intersectionOf (
            abstract:Adapter
            [ meta:pattern "?this model:mappingAdapter
                \" ContactsActions::getPhoneNumber\" " ]
        )
    ];
    inter:adaptFrom qtm:Contact;
    inter:adaptTo qtm:PhoneNumberString
```

The first two lines above give us a reason to conclude the semantic compatibility of *:SearchContactPhoneQt* task with *:ComposeSMSQt* task. The adapter definition then indicates that there exists a way to transform a QObject into a simple string representation of the contact's phone number. The above example assumes that the adapter was wrapped as a single method. Smart Modeller, however, also provides an option of giving a free-form line of code as the value of model:mappingAdapter. In this case, it would be *((QContactPhoneNumber) ?.detail(QContactPhoneNumber::DefinitionName)).number()*, i.e. as was given in Section 1 only with a question mark to indicate the place of the input resource.

Due to availability of an adapter, we can conclude that, in addition to the semantic compatibility, syntactic compatibility is achievable as well. In result, we have all the inputs needed for automated *opportunistic composition* of models.

While opportunistic composition is forward-chaining, backward-chaining composition is possible as well. We refer to it as *result-based composition* of models. In it, a model designer specifies what kind of resource he/she would like to be produced. Then, tasks annotated as producing this kind of resource are searched for, and after that, the current model is searched for tasks that can supply required inputs.

This part of our ODSE framework poses the following new requirement to ODSE tools:

- *Requirement 8.* Ability to perform the following process: (1) find in a Tasks ontology a task that is annotated as producing a resource, (2) locate the task in the model, (3) find in a Tasks ontology semantically-compatible consuming tasks, (4) if the producing and the consuming tasks use different syntactic formats, find proper adapters, (5) locate consuming tasks' implementations in repositories, (6) list to the designer as propositions semantically-compatible tasks that do not need or have proper adapters and have implementations, (7) upon a selection, import the implementation of the new task into the model and automatically connect with proper control and data flows. This provides the opportunistic composition.

- *Requirement 9.* Ability to perform the following process: (1) find in a Tasks ontology tasks producing some resources and present the list of those resource types to the designer, (2) upon selection of a resource type, list tasks that can produce it, (3) upon selection of a task, search the current model for included tasks that can provide resources that the new task requires, and find adapters if needed, (4) list options for each input to the designer, (5) upon selections, import the implementation of the new task into the model, and add data flow connections based on selections. This provides the result-based composition.

6.6. *Using other interaction properties*

In many cases, an action enables another action not directly through a resource produced but indirectly through changing the state of the software system or its environment. For this reason, our ODSE framework suggest use at least two other interaction properties for tasks' annotation: *inter:precondition* and *inter:effect*. As objects, both take SPARQL-based patterns referring to concepts from some domain ontology. Variables in these patterns can refer to variables defined in the task definition pattern, as well as introduce new ones. For example:

```
:VoiceInform inter:precondition
    "FILTER NOT EXISTS { ?x ex:volumeLevel ?vol.
        FILTER (?vol > 30) }".
:Mute inter:effect "?device ex:volumeLevel 0".
```

Above, the precondition for *:VoiceInform* task is that any device in the environment has to have the volume level below or equal to 30. The effect of *:Mute* task is that the device identified by *?device* variable, which is most likely an input parameter to the task, has the volume level at zero.

Our framework does not mandate how exactly such preconditions and effects annotations are to be processed in ODSE tools. The purpose is, however, clear: to match preconditions against effects for improved opportunistic design and/or compatibility checking.

6.7. *Defining policies*

Semantic Web based approaches to policies, especially for access control, have been developed in recent years [39]. Usually, such approaches, e.g. [40, 39], define policies in terms of *prohibitions* or *permissions* for certain actors to perform certain operations.

In our ODSE framework, the policies are included as prohibitions and permissions for a certain task to appear as a part of a software model. The policy definitions are as simple as:

```
owl:Thing inter:prohibited :StorePrivateData.
ex:PersonalDevice inter:permitted :StorePrivateData.
```

These definitions assume that a policy defined for a more specific class, *ex:PersonalDevice*, overrides a policy defined for a more general class, *owl:Thing* (that means anything or anyone). Therefore, the implied policy is that only devices belonging to the class of personal devices are allowed to persistently store the user's private data.

Enforcement of such policies in our framework is straightforward. Because a task has a detectable SPARQL-based pattern, running this pattern as a query against the RDF representation of the model will result in detecting all instances of the task. This can be done, for example, after every change made to the model. If a task is prohibited by a policy and is detected after a change to the model, one possible response from the ODSE tool is to cancel this last change while informing the designer about the policy. An issue is the need to know the behaviour of what actor the software model in question describes. This question is dependent on the modelling language used and is therefore, out of the scope of the ODSE framework.

In many practical cases, the policies would apply not to execution of a single software component but rather to several components connected with some particular control and data flows. Recall that our framework allows defining composite tasks with *meta:compositionOf* property (see Section 6.2) and, therefore, easily enables defining such policies.

This part of our ODSE framework poses the following new requirement to ODSE tools:

- *Requirement 10.* Ability to check and enforce task-based policies defined in the form of prohibitions and permissions.

7. Smart Modeller

The ODSE framework presented in Section 6 was implemented and exploited in a ODSE tool called Smart Modeller, which is one of the products of the research project SOFIA [7]. A description of the modelling language used in Smart Modeller and some technical details of this tool can be found in [8, 9] but are outside the scope of this paper.

Smart Modeller is a central tool in the ontology-driven Application Development Kit of SOFIA. Smart Modeller defines a domain-specific modelling language tailored to SOFIA needs and provides a graphical editor for that language realized with the Eclipse Graphical Modelling Framework (GMF) [41]. This editor enables the developer to create a model of an application (presented as a directed graph consisting of elements and connectors) and then to automatically generate executable programming code for it.

Additionally, an extension interface is provided so that various extensions (plug-ins) can be developed for Smart Modeller. Such extensions aim at further automation of the process, contributing to the ease and speed of application development. One set of extensions that was developed consists of extensions that jointly implement our ODSE framework, i.e. fulfil the Requirements 1 through 10 set in Section 6. These extensions provide the classification and model consistency checking services, the repository mechanism, the task-based, result-based, and opportunistic model composition services, as well as the policy enforcement service. Processing of RDF, SPARQL querying, and RDF-S reasoning are handled in Smart Modeller by exploiting OpenRDF Sesame [42].

Below, we demonstrate a few features brought into Smart Modeller through the implementation of the ODSE framework. The depicted models belong to a simple application, of which our example from Section 1 is a part. This application monitors incoming short messages ("Subscribe To and Get Last Message"), searches for a contact mentioned by name in the message text ("Find Contact Reference"), and then opens the dialog for composing a message to that contact ("Compose Message").

Figure 4 depicts an example of the Model Checking extension run. The identified model issues are presented as a dialog window with a list of errors and warnings. When one of the rows is selected, the corresponding model element, as well as its containers, is emphasized in the model with a red bounding rectangle. The dialog window is non-modal and always-on-top, which enables fixing the problems in the model without a need to close the dialog first. This is our recommendation for implementing such a model checking guidance service in ODSE tools.

Figure 5 depicts an example of the performance of the Opportunistic Composition extension. The starting model (left subfigure) includes a "Find Contact Reference" block which corresponds to *:SearchContactPhoneQt* task that we used in Section 6. After the designer engages the opportunistic composition function, it analyses the model and proposes, among other things, to add "Compose Message" task (*:ComposeSMSQt* from Section 6). After "Select" is clicked, the model is automatically extended to one shown in the right subfigure – graph is exported from the repository (a composite with Action and Parameter elements, shown collapsed), proper connectors are automatically added, and even the adapter extracting the phone number from a QContact object (see Section 6.5) is incorporated as an attribute of the connector from *out:contact* port to *in:number* port (not represented graphically).

8. Evaluation

In this section, we report on a performance evaluation of our ODSE framework. This evaluation was carried out based on the framework implementation in the Smart Modeller, which was briefly described in Section 7. The experiments were performed on a Windows XP laptop with Intel Core 2 Duo 2.4 Hz processor. All reported numbers are obtained as averages of 10 runs.

In the first series of experiments presented in Tables 5 and 6 as well as Figures 6 and 7, we evaluated the performance of the framework with respect to the model consistency checking service. We also compared it to the performance of OWL reasoning as a more traditional means for implementing such a service. To do such a comparison, the Smart Modeller was extended with OWL-based model checking that utilized as the reasoning engine Pellet [43, 44] version 2.2.2 with OWL-API access interface. To deal with OWL's Open World Assumption

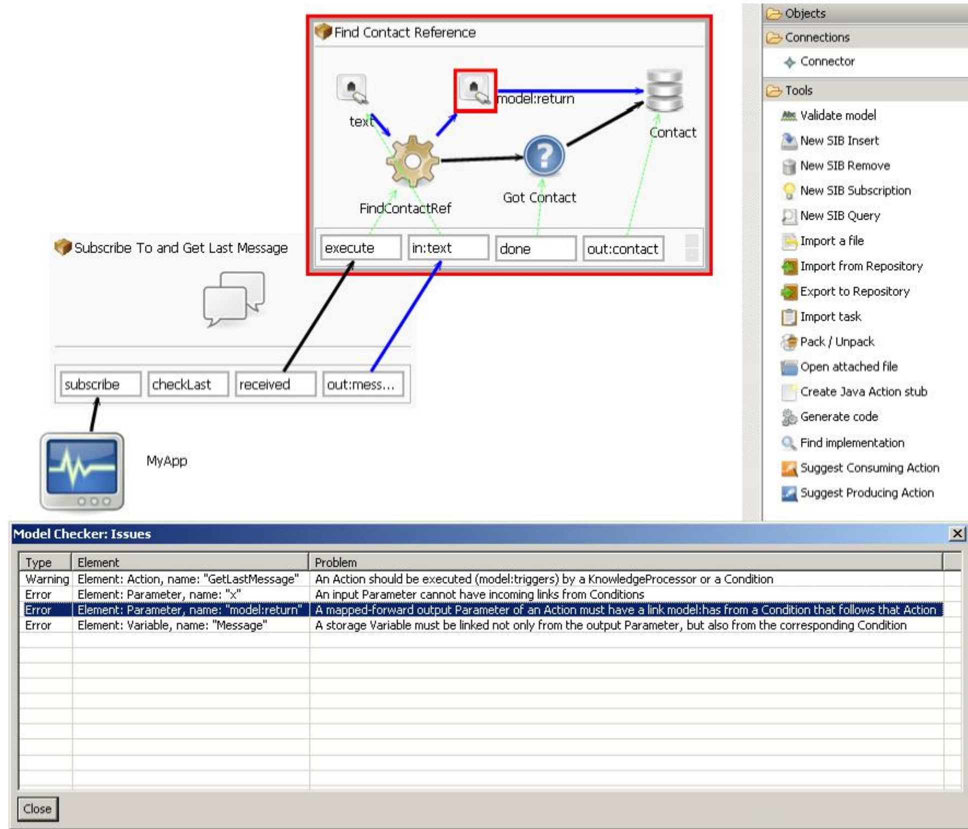

Figure 4: Model checking service in Smart Modeller.

to be able to represent a wider range of constraints, we used the approach mentioned in the end of Section 3, where a validity restriction is attached to a separate classe, say, ValidAction, and then a check is performed if the sets of Action and ValidAction are the same.

Table 5 and the corresponding Figure 6 show the dependency between the processing time measured in milliseconds and the complexity of the ontology measured as the number of classes with validity constraints in it. The complexity of the model, measured as the number of contained individuals: elements and connectors, was held at 50 individuals. As can be seen, our framework achieves a noticeably better performance even with 2 classes in the ontology, and the difference in performance grows fast as the complexity of the ontology increases. Our framework is performing very fast, always under 0.05 second in these experiments, and the processing time grows relatively slow as the complexity increases. OWL reasoning also seemed to process certain constraints slower than others, resulting in non-linearity in Figure 6.

Table 5: Performance in model checking vs. ontology complexity

N of classes	2	4	6	8	10
OWL/Pellet	162.4	335.9	370.3	799.9	853.2
Our	35.9	37.2	43.6	46.8	47.0

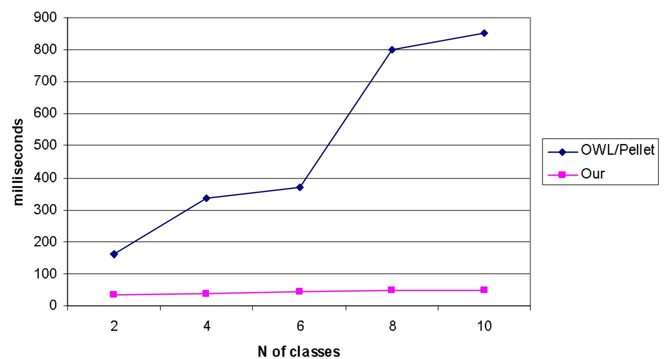

Figure 6: Performance in model checking vs. ontology complexity with 50 individuals

Table 6 and the corresponding Figure 7 show the dependency between the processing time the complexity of the checked model. The complexity of the ontology was held at 10 classes. The difference in performance is even more significant in these experiments. OWL reasoning demonstrates a performance that degrades fast in a fashion of a square function, and with 100 individuals in the model it almost reaches 6 seconds. Note that a model of 100 individuals and an ontology of 10 classes is a case of a rather moderate complexity. Given the obtained pro-

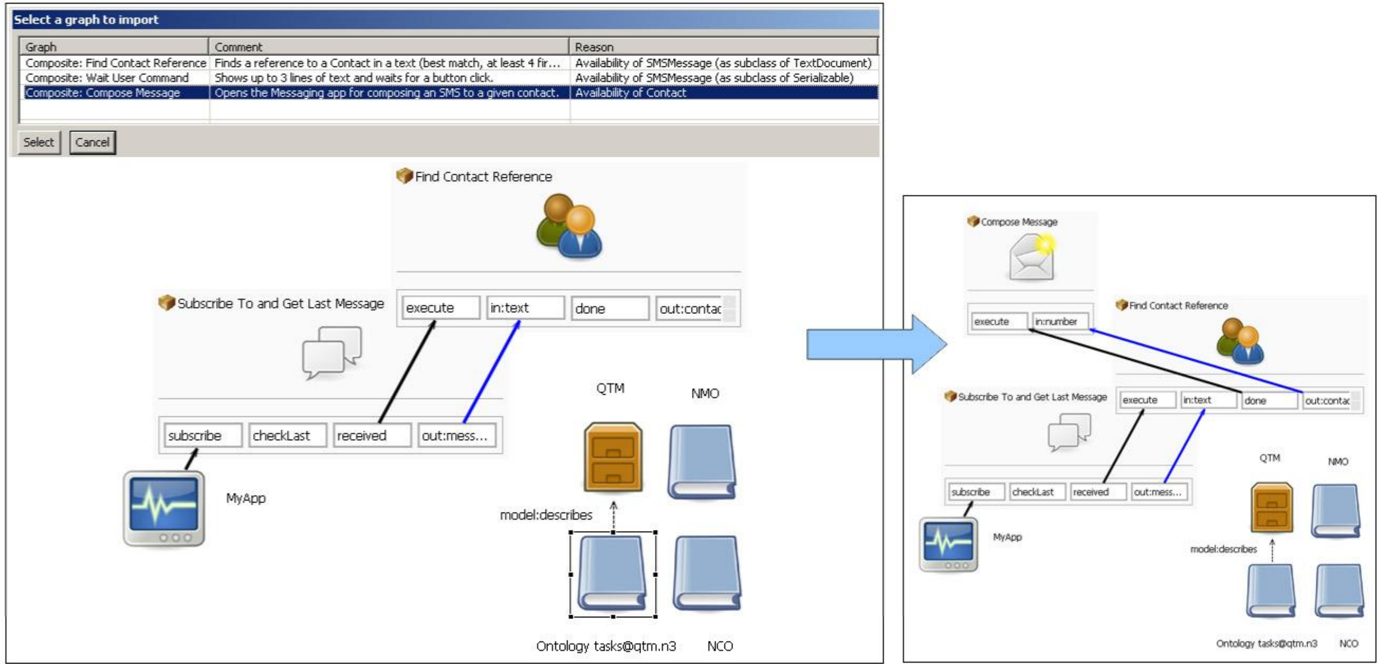

Figure 5: Opportunistic model composition in Smart Modeller.

cessing times and the shape of the dependency, one may even question the practical usability of OWL for checking more complex models. In contract, the performance of our framework degraded linearly in our experiments and stayed under 0.1 second in all cases. Therefore, one can expect it to perform very fast also on more complex models.

Table 6: Performance in model checking vs. model complexity

N of individuals	25	50	75	100
OWL/Pellet	265.8	853.2	2661.2	5571.8
Our	40.7	47.0	57.8	73.6

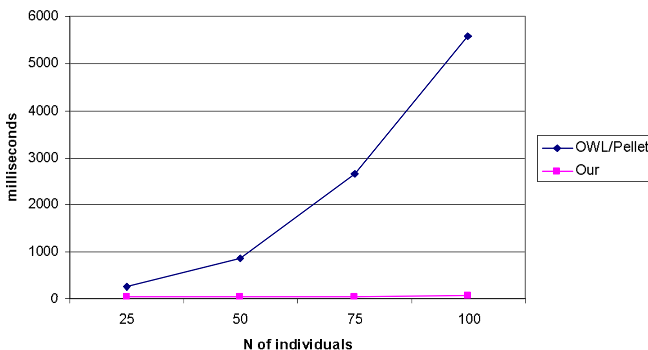

Figure 7: Performance in model checking vs. model complexity, 10 classes

In the second series of experiments, we evaluated the performance of the framework with respect to the opportunistic model composition service. This service was selected because it involves, compared to other ontological services, the most extensive multi-step reasoning. Figure 8 shows the processing times for the cases of different complexity. The complexity is measured here as the number of tasks in a Tasks ontology that have implementations and therefore found in the current model.

We report separately the time of the actual reasoning and the time that it takes to load all the relevant data into a Sesame RDF repository. In our experiments, the data loading times were comparable to the reasoning times and therefore not negligible, at around 0.5 seconds. It is because five different documents have to be loaded: the current model, the model repository, the Tasks ontology, as well as two domain ontologies: NEPOMUK Contact and Message ontologies. We report the loading time separately as most of the loaded data is static and it is easy to avoid re-loading it every time the service is engaged.

As can be seen, the reasoning time grows with the complexity but still remains quite fast, under 1 second. Note that the case of 20 tasks involved around 300 individuals in the model, i.e. is somewhat more complex than those in the first series of experiments.

9. Conclusions

In this paper, we proposed a framework for Ontology-Driven Software Engineering. This framework is grounded on the prior related work [2, 14] that studied the interplay between the model-driven engineering and the ontological modelling. The contribution of our framework is then two-fold. First, it incorporates a more flexible and means for ontological modelling that also has a higher performance in processing, which

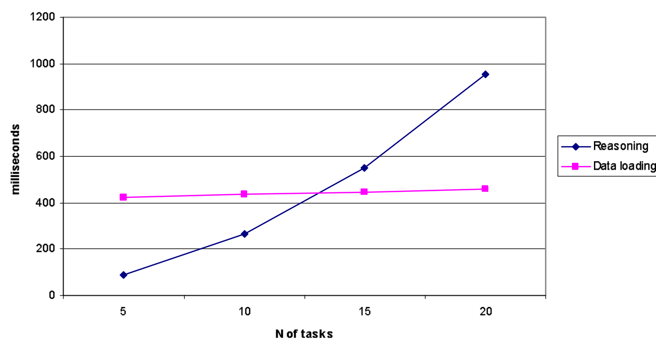

Figure 8: Performance in opportunistic model composition

is a hybrid of Description Logic with SPARQL patterns. In result, the expressiveness as well as the speed of execution of e.g. model checking service is improved. Second, it incorporates a wider range of ontology types into ODSE and enables new ontological services: semantic search in model repositories, three kinds of semi-automated model composition services: task-based, result-based, and opportunistic, and policy enforcement service.

The primary intended use for this framework is to be implemented as part of model-driven engineering tools to support software engineers. We described our reference implementation of such a tool called Smart Modeller, and reported on a performance evaluation of our framework carried out with the help of it.

Our performance evaluation indicates that the framework enables not only the delivery of above mentioned services, but their delivery in a very timely manner. In particular, the model checking takes less than 0.1 sec. This basically means that there is no visible delay between clicking on the "Check Model" and appearance of the window with the results. Therefore, the model designer can engage this service as often as he/she wants, even as a way to clarify for him/herself the specifics of the modelling language. As can be seen from our evaluation in Section 8, our framework reaches up to 75 times improvement over an OWL-based implementation of model checking even on moderately complex models.

The performance of more complex services, such as opportunistic model composition, is also quite fast, at around 1 second for even more or less complex models. Therefore, the model designer does not have to think twice before engaging this service. Even if such semi-automatic composition of models is not used as the primary means of work, the lightweight nature and speed of this function makes it a reasonable addition to MDE tools. Nowadays, programmers and developers who want to (re)use existing components and services have to either search for those components manually or perform lookups which are based on syntactical descriptions. In practice, a lot of effort goes into finding appropriate components or API functions, reading documentation about them, searching for examples, understanding how to use the components together, e.g. to adapt the output of one to fit into the input of another. Semi-automated composition services provided by our framework can act as software documentation that can automatically be analysed in the context of the current model. A designer can engage these services just to obtain some information and support own decision making, or to actually speed up the process of a model composition. For example, the composition from scratch of a 3-step application as in Figure 5 takes around 14 mouse clicks and 20 seconds.

Although it is not difficult or expensive to implement our ODSE framework into MDE tools, the actual cost of enabling the composition services is obviously related to preparing the machine-processable documentation, i.e. modelling software components and API functions as well as extending these models with semantic annotations. A study of the trade-off between this cost and the value added is an important direction of future work.

Acknowledgements

This work is performed in the project SOFIA, which is a part of EU's ARTEMIS Joint Undertaking. SOFIA is coordinated by Nokia and the partners include Philips, NXP, Fiat, Elsag Datamat, Eurotech, Indra, as well as a number of research institutions from Finland, Netherlands, Italy, and Spain.

We are grateful to the anonymous reviewers for their valuable comments and suggestions.

References

[1] D. C. Schmidt, Model-driven engineering, IEEE Computer 39 (2) (2006) 25–31.

[2] S. Staab, T. Walter, G. Gröner, F. S. Parreiras, Model driven engineering with ontology technologies, in: Proc. Summer School on Reasoning Web, LNCS vol. 6325, Springer, 2010, pp. 62–98.

[3] W3C, Ontology Driven Architectures and Potential Uses of the Semantic Web in Systems and Software Engineering, online: http://www.w3.org/2001/sw/BestPractices/SE/ODA/ (2006).

[4] Object Management Group, OMG Model Driven Architecture, online: http://www.omg.org/mda/.

[5] Y. Singh, M. Sood, Model driven architecture: A perspective, in: Proc. IEEE International Advance Computing Conference, 2009, pp. 1644–1652.

[6] A. Katasonov, Enabling non-programmers to develop smart environment applications, in: Proc. IEEE symposium on Computers and Communications, 2010, pp. 1055–1060.

[7] P. Liuha, A. Lappeteläinen, J.-P. Soininen, Smart objects for intelligent applications – First results made open, ARTEMIS Magazine 5 (2009) 27–29.

[8] A. Katasonov, M. Palviainen, Towards ontology-driven development of applications for smart environments, in: Proc. Workshops of IEEE Intl. Conf. on Pervasive Computing and Communications, 2010, pp. 696–701.

[9] M. Palviainen, A. Katasonov, Model and ontology-based development of smart space applications, in: Pervasive Computing and Communications Design and Deployment: Technologies, Trends, and Applications, IGI Global, 2011, pp. 126–148.

[10] J. Bezivin, V. Devedzic, D. Djuric, J.-M. Favreau, D. Gasevic, F. Jouault, An M3-neutral infrastructure for bridging model engineering and ontology engineering, in: Proc. Intl. Conf. on Interoperability of Enterprise Software and Applications, Springer, 2005, pp. 159–171.

[11] G. Kappel, E. Kapsammer, H. Kargl, G. Kramler, T. Reiter, W. Retschitzegger, W. Schwinger, M. Wimmer, Lifting metamodels to ontologies: A step to the semantic integration of modeling languages, in: Proc. MoDELS, LNCS vol.4199, Springer, 2006, pp. 528–542.

[12] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks, EMF: Eclipse Modeling Framework, 2nd ed., Addison-Wesley Professional, 2009.
[13] S. Roser, B. Bauer, Automatic generation and evolution of model transformations using ontology engineering space, in: Journal on Data Semantics XI, LNCS vol.5383, Springer, 2008, pp. 32–64.
[14] K. Miksa, P. Sabina, M. Kasztelnik, Combining ontologies with domain specific languages: A case study from network configuration software, in: Proc. Summer School on Reasoning Web, LNCS vol. 6325, Springer, 2010, pp. 99–118.
[15] N. Jekjantuk, J. Pan, G. Gröner, Verifying and validating multi-layered models with OWL FA toolkit, in: Proc. 7th Extended Semantic Web Conference, LNCS vol. 6089, Springer, 2010, pp. 391–395.
[16] M. Gogolla, M. Kuhlmann, L. Hamann, Consistency, independence and consequences in UML and OCL models, in: Proc. Tests and Proofs, LNCS vol.5668, Springer, 2009, pp. 90–104.
[17] D.-H. Dang, M. Gogolla, Precise model-driven transformations based on graphs and metamodels, in: Proc. Intl. Conf. on Software Engineering and Formal Methods, IEEE, 2009, pp. 307–316.
[18] D.-H. Dang, A.-H. Truong, M. Gogolla, Checking the conformance between models based on scenario synchronization, Journal of Universal Computer Science 16 (17) (2010) 2293–2312.
[19] F. S. Parreiras, S. Staab, Using ontologies with UML class-based modeling: The TwoUse approach, Data and Knowledge Engineering 69 (11) (2010) 1194–1207.
[20] M. Vanden Bossche, P. Ross, I. MacLarty, B. Van Nuffelen, N. Pelov, Ontology driven software engineering for real life applications, in: Proc. 3rd Intl. Workshop on Semantic Web Enabled Software Engineering, 2007.
[21] A. Soylu, P. de Causmaecker, Merging model driven and ontology driven system development approaches: Pervasive computing perspective, in: Proc. 24th Intl. Symposium on Computer and Information Sciences, 2009, pp. 730–735.
[22] B. Bauer, S. Roser, Semantic-enabled software engineering and development, in: Proc. Informatik, LNI vol.94, Gesellschaft für Informatik, 2006, pp. 293–296.
[23] F. Ruiz, J. R. Hilera, Using ontologies in software engineering and technology, in: Calero, C., Ruiz, F., Piattini, M. (eds) Ontologies for Software Engineering and Software Technology, Springer Verlag, 2006, pp. 49–102.
[24] K. M. de Oliveira, K. Villela, A. R. Rocha, G. H. Travassos, Use of ontologies in software development environments, in: Calero, C., Ruiz, F., Piattini, M. (eds) Ontologies for Software Engineering and Software Technology, Springer Verlag, 2006, pp. 276–309.
[25] D. Song, W. Liu, Y. He, K. He, Ontology application in software component registry to achieve semantic interoperability, in: Proc. Intl. Conference on Information Technology: Coding and Computing, Volume II, IEEE CS, 2005, pp. 181–186.
[26] D. Oberle, A. Eberhart, S. Staab, R. Volz, Developing and managing software components in an ontology-based application server, in: Proc. 5th ACM/IFIP/USENIX international conference on Middleware, LNCS Vol.3231, Springer-Verlag, 2004, pp. 459–477.
[27] C. Pahl, An ontology for software component matching, Int. J. Software Tools for Technology Transfer 9 (2) (2007) 169–178.
[28] A. H. H. Ngu, M. P. Carlson, Q. Sheng, H.-Y. Paik, Semantic-based mashup of composite applications, IEEE Transactions on Services Computing 3 (1) (2010) 2–15.
[29] B. Hartmann, S. Doorley, S. Klemmer, Hacking, mashing, gluing: Understanding opportunistic design, IEEE Pervasive Computing 7 (3) (2008) 46–54.
[30] M. T. Gamble, R. Gamble, Monoliths to mashups: Increasing opportunistic assets, IEEE Software 25 (6) (2008) 71–79.
[31] C. Atkinson, T. Kuhne, Model-driven development: A metamodeling foundation, IEEE Software 20 (5) (2003) 36–41.
[32] H. Knublauch, Where OWL fails, online: http://composing-the-semantic-web.blogspot.com/2010/04/where-owl-fails.html (2010).
[33] Object Management Group, Architecture-Driven Modernization (ADM): Knowledge Discovery Meta-Model (KDM), online: http://www.omg.org/spec/KDM/1.1/PDF/ (2009).
[34] K.-K. Lau, V. Ukis, Component metadata in component-based software development: A survey, Preprint CSPP-34, The University of Manchester (2005).
[35] T. Vitvar, J. Kopecky, M. Zaremba, D. Fensel, WSMO-Lite: Lightweight descriptions of services on the web, in: Proc. 5th IEEE European Conference on Web Services, IEEE CS, 2007, pp. 77–86.
[36] T. Moyaux, B. Lithgow-Smith, S. Paurobally, V. Tamma, M. Wooldridge, Towards service-oriented ontology-based coordination, in: Proc. IEEE Intl. Conference on Web Services, 2006, pp. 265–274.
[37] T. Berners-Lee, Notation 3: A readable language for data on the Web, online: http://www.w3.org/DesignIssues/Notation3.html.
[38] NEPOMUK, NEPOMUK Contact Ontology, online: http://www.semanticdesktop.org/ontologies/nco/.
[39] T. Finin, A. Joshi, L. Kagal, J. Niu, R. Sandhu, W. Winsborough, B. Thuraisingham, ROWLBAC: Role based access control in OWL, in: Proc. ACM Symposium on Access Control Models and Technologies, 2008, pp. 73–82.
[40] A. Naumenko, Semantics-based access control – Ontologies and feasibility study of policy enforcement function, in: Proc. ACM 3rd International Conference on Web Information Systems and Technologies, Volume Internet Technologies, 2007, pp. 150–155.
[41] Eclipse Foundation, Graphical Modeling Project, online: http://www.eclipse.org/modeling/gmp/.
[42] OpenRDF.org, Sesame, online: http://www.openrdf.org/.
[43] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, Y. Katz, Pellet: A practical OWL-DL reasoner, Journal of Web Semantics 5 (2) (2007) 51–53.
[44] Clark&Parsia, Pellet: OWL 2 Reasoner for Java, online: http://clarkparsia.com/pellet/.