Journal of Automated Software Engineering manuscript No.
(will be inserted by the editor)

From Requirements to Source Code: A Model-Driven Engineering Approach for RESTful Web Services

Christoforos Zolotas · Themistoklis Diamantopoulos · Kyriakos Chatzidimitriou · Andreas L. Symeonidis

Received: date / Accepted: date

Abstract During the last few years, the REST architectural style has drastically changed the way web services are developed. Due to its transparent resource-oriented model, the RESTful paradigm has been incorporated into several development frameworks that allow rapid development and aspire to automate parts of the development process. However, most of the frameworks lack automation of essential web service functionality, such as authentication or database searching, while the end product is usually not fully compliant to REST. Furthermore, most frameworks rely heavily on domain specific modeling and require developers to be familiar with the employed modeling tech-

Christoforos Zolotas
Electrical and Computer Engineering Dept. Aristotle University of Thessaloniki GR54124, Thessaloniki, Greece
Tel.:+30 2310 996365
Fax: +30 2310 996398
E-mail: christopherzolotas@issel.ee.auth.gr

Themistoklis Diamantopoulos
Electrical and Computer Engineering Dept. Aristotle University of Thessaloniki GR54124, Thessaloniki, Greece
Tel.:+30 2310 996365
Fax: +30 2310 996398
E-mail: thdiaman@issel.ee.auth.gr

Kyriakos Chatzidimitriou
Electrical and Computer Engineering Dept. Aristotle University of Thessaloniki GR54124, Thessaloniki, Greece
Tel.:+30 2310 996349
Fax: +30 2310 996398
E-mail: kyrcha@issel.ee.auth.gr

Andreas L. Symeonidis
Electrical and Computer Engineering Dept. Aristotle University of Thessaloniki GR54124, Thessaloniki, Greece
Tel.:+30 2310 99 4344
Fax: +30 2310 99 6398
E-mail: asymeon@eng.auth.gr

nologies. In this paper, we present a Model-Driven Engineering (MDE) engine that supports fast design and implementation of web services with advanced functionality. Our engine provides a front-end interface that allows developers to design their envisioned system through software requirements in multimodal formats. Input in the form of textual requirements and graphical storyboards is analyzed using natural language processing techniques and semantics, to semi-automatically construct the input model for the MDE engine. The engine subsequently applies model-to-model transformations to produce a RESTful, ready-to-deploy web service. The procedure is traceable, ensuring that changes in software requirements propagate to the underlying software artefacts and models. Upon assessing our methodology through a case study and measuring the effort reduction of using our tools, we conclude that our system can be effective for the fast design and implementation of web services, while it allows easy wrapping of services that have been engineered with traditional methods to the MDE realm.

Keywords RESTful Web Services · Model-Driven Engineering · Software Requirements · Automated Software Engineering

1 Introduction

Lately, offering software in the form of web services has gained popularity due to the evolution of cloud architectures and the introduction of Internet of Things concepts. As expected, next generation web services have been developed that have revolutionized the way software is developed. Ever since its introduction by Fielding (2000), the *REpresentational State Transfer (REST)* architectural style has been increasingly preferred by developers for its simplicity and scalability, and has practically grown to be the state-of-the-practice for creating web services. REST comprises a set of rules and practices offering simple comprehensible APIs, clear representations, and scalable services (Richardson and Ruby, 2007).

Meanwhile, from an automated software engineering perspective, *Model Driven Engineering (MDE)* is gaining popularity. MDE and its most prominent instance, *Model Driven Architecture* (MDA)[1] introduced by the *Open Management Group* (OMG)[2], *do* bring some benefits to its practitioners. According to Liebel et al (2014) and Hutchinson et al (2011), MDE practitioners report that it leads to a higher degree of automation and improved productivity, an important aspect of which is code generation. Additionally, they report that MDE results to improved quality and reduced defects, which are detected earlier in the lifecycle of the project, hence they can be fixed at a lower cost. Finally, others have argued that MDE accelerates the implementation of new requirements and increases understandability.

[1] `http://www.omg.org/mda/`

[2] `http://www.omg.org`

Due to its transparent resource-oriented model, the RESTful paradigm has been incorporated into several development frameworks, some of which embed MDE methodologies that allow rapid development and aspire to automate parts of the development process. Some of those systems (Ed-Douibi et al, 2015; Parastatidis et al, 2010) aim at creating a skeleton and in some cases a database schema for resources, while others construct fully functional web services. However, most of them require developer input in some form of a model/language, therefore exposing the developer to the most prominent MDE pitfalls. According to Liebel et al (2014) and Hutchinson et al (2011), these pitfalls are the significant training overhead needed in order to successfully use the technique, and the complexity of the modeling process which is considered challenging and requires too much effort.

In this paper we attempt to mitigate the aforementioned drawbacks of RESTful service frameworks, which also lack wide-spread requirements engineering techniques. Hence, we present the synergy of upper and lower CASE[3] AI technologies and tools, such as *Natural Language Processing (NLP)*, Ontologies and First Order Predicate Logic with several upper/lower CASE Software Engineering staple facilities, namely Multi-modal Requirements Management, Domain Specific Meta-modeling and MDE transformation chains. The outcome is a coherent mechanism that allows rapid prototyping and development of functional RESTful services, whilst the developer is able to model his/her envisioned system using comprehensive requirements representations, i.e. textual requirements and visual storyboards, without requiring further training on Domain Specific Languages or being exposed to complex modeling activities that pure MDE techniques embed.

Specifically, we present a two-fold mechanism that aids developers in building RESTful services. Its lower CASE back-end employs an MDE engine that automates the generation of RESTful services that abide by the 3rd level of *Richardson's Maturity Model (RMM)*[4]. It offers automatic Basic Authentication[5] functionality, it automates the process of popular database keyword-searching and ensures the interoperability of the envisioned RESTful service with existing 3rd party services.

Additionally, as already mentioned, in contrast to most transformation engines that receive input in the form of a model or language, our MDE engine is coupled with an upper CASE front-end interface that allows developers to design their envisioned system through software requirements in multimodal formats. In specific, the upper CASE effectively models the static and dynamic views of the envisioned system, and employs NLP techniques and semantics to transform functional requirements and dynamic system scenarios to system specifications. The derived representations are instantiated in software ontologies using tools specifically designed and developed for constructing REST-compliant models from functional requirements and storyboards (i.e. graphi-

[3] `https://en.wikipedia.org/wiki/Computer-aided_software_engineering`

[4] `http://martinfowler.com/articles/richardsonMaturityModel.html`

[5] `https://en.wikipedia.org/wiki/Basic_access_authentication`

cal scenarios). Functional requirements are parsed and semantically annotated to compose RESTful resources and properties, while storyboards are used to determine the hypermedia connections among resources.

The combination of the *requirements-to-specifications (Reqs2Specs)* module with the MDE engine results in a comprehensive and effective system with multiple advantages. It introduces increased automation in RESTful service development by using NLP and semantics to construct the input model of the MDE engine. Less effort is required for translating requirements to specifications, while the construction of web service prototypes is easy, even with minimal knowledge of the MDE and the RESTful paradigms. Furthermore, our methodology eases the migration of legacy software systems engineered with traditional methods to the MDE realm, since it is capable of taking as input their textual requirements. Finally, note that changes in software requirements are instantly propagated to the underlying software artefacts and models, thus providing high model consistency and traceability.

The rest of this paper is structured as follows. Section 2 reviews the current state-of-the-art in the area of specification extraction from software requirements and MDE transformation engines. Section 3 introduces the framework behind our approach and describes the modules of our system, including the design of the ontologies for describing the components of the envisioned service and the methods used for automatically processing functional requirements and storyboards to instantiate them. Section 4 introduces the MDE engine we designed and developed, focusing on the essential meta-models and the transformations between them. Finally, Section 5 illustrates how our system can be used to create a RESTful web service and provides assessment of the effort reduction by using our mechanism, and Section 6 concludes this paper, summarizing our contributions and providing useful insight for future research.

2 Related Work

Several research efforts aspire to help developers minimize the cost and development time and maximize their effectiveness. In the context of *Computer Aided Software Engineering (CASE)*, several tools, known as *lower CASE tools*, require as input a model that undergoes subsequent transformations in order to provide the source code of the envisioned system. On the other hand, extraction of specifications, i.e. the initial model, from software requirements, is traditionally handled by *upper CASE tools*. Subsections 2.1 and 2.2 discuss upper and lower CASE efforts and illustrate their relation to our approach.

2.1 Upper CASE Approaches

Translating software requirements to models involves designing meaningful representations for system specifications and instantiating them using input from software requirements. Concerning the type of the input, most approaches

propose specialized languages that can be easily translated to models. The tools of the *ReDSeeDS* project (Kaindl et al, 2007; Smialek, 2012) use a constrained language named *Requirements Specification Language (RSL)* (Kaindl et al, 2007) in order to extract specifications from use cases and domain models. *Cucumber* (Wynne and Hellesoy, 2012) and *JBehave* (North, 2003) are two other popular frameworks based on the *Behavior-Driven Development (BDD)* paradigm. These tools allow defining scenarios using a GIVEN-WHEN-THEN approach and aspire to construct testable behavioral models. Other notable examples using specialized languages include *Tropos*, a requirements-driven methodology designed by Mylopoulos et al (2000), which is based on the notions of actors and goals of the i* modeling framework (Yu, 1995).

Although the aforementioned approaches can be quite useful for constructing precise models, their compliance to UML and/or other standards is sometimes limited. Using them may require training in a new language, which is sometimes frustrating for the developers. As a result, current literature also includes employing semantics and NLP techniques on natural (or semi-structured) language functional requirements and UML diagrams to extract specifications. One of the first rule-based methods for extracting data types, variables and operators from requirements were introduced by Abbott (1983), according to which nouns are identified as objects and verbs as operations between them. Subsequently, Abbott's approach was extended to object-oriented development by Booch (1986).

Saeki et al (1989) were among the first to construct a system for extracting object-oriented models from informal requirements. Their system uses NLP methods to extract nouns and verbs from functional requirements and determines whether they are relevant to the model by means of human intervention. The work by Mich (1996) also involved a semantics module using a knowledge base in order to further assess the retrieved syntactic terms. In a more recent approach, Harmain and Gaizauskas (2003) developed *CM-Builder*, a natural language upper CASE tool in order to identify object classes, attributes and relationships in the analysis stage using functional requirements.

As noted in the previous paragraphs, there are several upper CASE tools that can be used to extract specifications from models. However, most of these tools focus on object-oriented development and use specialized languages. Hence, most of them are not oriented towards MDE (with the exception of ReDSeeDS (Smialek, 2012)), while none of them, and no other to the best of our knowledge, complies with the main characteristics of the RESTful paradigm. In contrast to the object-oriented paradigm, the building blocks of a RESTful web service are the *resources*. Each resource provides an object of the system that can be addressed using a *Create, Read, Update, or Delete (CRUD)* operation and connects to other objects via *hypermedia* links. In this work, we employ NLP techniques in a domain-agnostic context to extract resources as structural system elements from functional requirements. Additionally, we develop a dynamic representation that can describe the dynamic view of RESTful services effectively and provide action flows and hypermedia links for the resources.

Concerning model representations, most of the aforementioned systems use either known literature standards including UML models or ontologies, which have been used extensively in *Requirements Engineering (RE)* for storing and validating information extracted from requirements (Castañeda et al, 2010; Siegemund et al, 2011). Ontologies are also useful for storing domain specific knowledge and are known to integrate well with software modeling languages (Happel and Seedorf, 2006)[6]. They provide a structured means of organizing information, are particularly useful for storing linked data, allow retrieving stored data via queries, and allow reasoning over implied relationships between data items. Thus, we also employ ontologies to store the concepts extracted from functional requirements and storyboards and construct the specifications of the envisioned service.

2.2 Lower CASE Approaches

Although there is lack of upper CASE tools that model software requirements for RESTful services, there is a plethora of lower CASE tools that aim to help the developer model RESTful services in some way. However, most of them fall short in some way. The majority does not fully support the RESTful design, as they either do not support the hypermedia concept of REST or do it partially since developer intervention is necessary. Others, although they are excellent at modeling the REST design, do not provide essential functionality that is necessary for a web service, such as Basic Authentication, automated keyword-searching or interoperation with existing 3rd party services.

More specifically, EMF-REST (Ed-Douibi et al, 2015) models really well the REST domain and supports hypermedia structurally, however it is too data-centric and thus does not model and automate any non-CRUD functionality. RESTfulie (Parastatidis et al, 2010) also models the REST concepts adequately, including hypermedia, however it does not include any common web service functionality either. In a more research-wise work, Schreier (2011) models both structurally and behaviorally the REST architectural style, although any other functionality such as authentication etc. is left as future work. In the same manner, several other popular tools such as Persevere[7], Restlet[8], Django-REST[9] and Rails[10] also fall short in some way, for example by not supporting hypermedia or by not being able to take into account User Software Requirements in the wide-spread textual format, since their operating mode is based on a API-specification development logic. Most of the other tools and frameworks of this category[11] illustrate such issues as well.

[6] See (Dermeval et al, 2015) for a systematic review on the use of ontologies in RE.

[7] `http://www.persvr.org/`

[8] `http://restlet.com`

[9] `http://www.django-rest-framework.org`

[10] `http://rubyonrails.org`

[11] `https://code.google.com/p/implementing-rest/wiki/ByLanguage`

Other efforts, concern the modeling of RESTful services formally or conceptually. Hernández and García (2010) formally model the REST design using tuples and process calculi, while Zuzak et al (2011) employ Finite State Machines (FSMs). On the other hand, Rauf et al (2010) and Zhao and Doshi (2009) formalize compositions of several RESTful services to form a composite one. All these attempts mostly focus on conceptual modeling of RESTful services, rather than aiming to directly support their development.

Finally, other lower CASE tools or languages model semantically RESTful services in the form of a textual description language. These mostly build on top of the WADL[12] standard and attempt to better describe the behavior of a RESTful service, so that it will be easier consumable by others. Tavares and Vale (2013) apply an MDA methodology to produce textual description of RESTful services in several popular formats, such as WADL, WSDL 2.0, SA-REST and OWL-S, while Porres and Rauf (2011) describe a way to include in the textual description of a RESTful service interface behavioral aspects as well, which are not tackled by the WADL standard.

Further extending previous work, our approach attempts to properly model all the REST design concepts in regard to RMM and in the same time offer modeling capabilities for essential functionality that is used in any web service. Additionally, our approach actively supports development rather than conceptual or language modeling. Finally, the proposed two-fold RESTful development mechanism exploits textual User Requirements in order to ease the modeling effort of the developer, instead of following a pure modeling path.

3 From Requirements to Specifications

3.1 System Overview

The conceptual architecture of our system is shown in Figure 1.

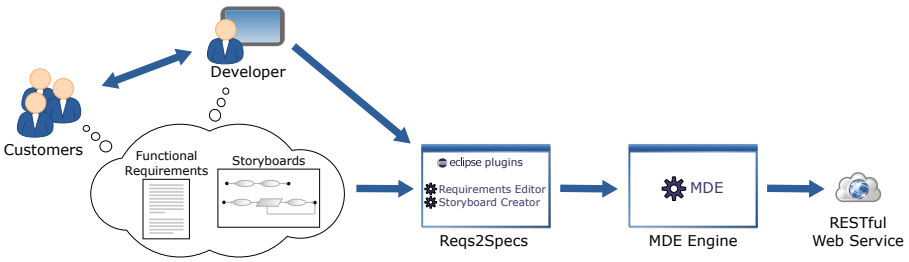

Fig. 1 Overview of the Conceptual Architecture of our System

Our system comprises two modules: the *Reqs2Specs module* and the *MDE engine*. The Reqs2Specs module includes a set of tools for developers to enter the multi-modal representations (functional requirements, storyboards) of the

[12] `http://www.w3.org/Submission/wadl/`

envisaged software application as well as a methodology for transforming these representations to specifications, i.e. models of the envisioned system. The MDE engine performs a series of model-to-model transformations, while also allowing refinements to these transformations by the developer, in order to subsequently produce the final source code of the envisioned service.

In specific, we have designed two ontologies to capture the static and dynamic view of software projects, which are subsequently aggregated to a REST-compliant ontology to produce the specifications of a RESTful web service. Additionally, we have developed front-end tools that allow developers to insert requirements in the form of semi-structured text and storyboards, i.e. diagrams depicting the flow of actions in the system. Using the specifications extracted from these ontologies, the MDE engine constructs a *Computationally Independent Model (CIM)* for the envisioned service. Applying model-to-model transformations, the engine produces a fully-functional web service, including a database for the resources of the system and an API for the produced service.

The combination of these technologies allows the developer to significantly reduce the manual effort required to build RESTful services. Firstly, input is provided in the form of requirements and storyboards instead of complex models, while a semi-automated annotation methodology is used to expedite the process of importing them in our mechanism. Secondly, our mechanism embeds First Order Predicate Logic to help the developer refine the initial produced CIM model. Thirdly, the required expertise is reduced, since as section 4 discusses, our mechanism automatically produces source code for widely used functionality (authentication, search, 3rd party service integration), hence the developer does not have to be familiar with several related libraries and frameworks. Finally, our mechanism provides RESTful wrappers for the non-automatable aspects of the envisioned service, so that the developer only has to fill in manual code fragments in a compilable and executable template.

Most of the steps required to produce a RESTful service using our mechanism are automated. In specific, the user has to intervene only to refine the annotations of the imported requirements and/or storyboards, refine the produced CIM model, from which the MDE engine will automatically generate the corresponding code. A more elaborate presentation of the semi-automated and automated actions of our mechanism are shown in subsection 5.1, upon having presented the components of our system including any required terminology.

3.2 Extracting Artefacts from Software Requirements

This section concerns the design of ontologies for storing information derived from software requirements. In the context of our work we employ the Web Ontology Language (OWL)[13] for representing information, since it is a well-known established standard of current research and industry communities. Although we don't rely on OWL inference capabilities explicitly, they are useful for expressing integrity conditions over the ontology, such as ensuring that certain properties have inverse properties (e.g. `is_actor_of`/`has_actor`).

[13] `http://www.w3.org/TR/2004/REC-owl-guide-20040210/`

We have designed three ontologies, hereafter named the *static* ontology, the *dynamic* ontology, and the *aggregated* ontology. The static ontology represents the static view of software projects. It holds information about the objects and the resources of the system derived from static requirements representations, such as functional requirements or UML use case diagrams. The dynamic ontology stores dynamic information about software projects. It includes information about the activities and action/data flows throughout the system, while it can be instantiated by different types of dynamic representations, such as storyboards or UML activity diagrams. Finally, the aggregated ontology links the information provided by the two ontologies and provides a unified view of a software project, which is modeled to be REST-compliant.

In the following subsections, we initially present the static ontology and describe how it can be instantiated the ontology using functional requirements. In the same manner, we present the dynamic ontology and illustrate how it can be instantiated using storyboards. After that, the aggregated ontology is presented and its instantiation is discussed. Finally, the extraction of a YAML representation from the aggregated ontology is presented that is used as input for the MDE engine module. A YAML representation was considered so that it is more developer and eye friendly than an OWL file.

3.2.1 Static View of Software Projects

An Ontology for the Static View of Software Projects Concerning the static aspects of requirements elicitation, i.e. functional requirements and use case diagrams, the design of the ontology revolves around the concept of an acting unit (e.g. user) performing some action(s) on some object(s). Following the methodology defined by Roth et al (2014, 2015), the ontology was designed to support information extracted from Subject-Verb-Object (SVO) sentences. The class hierarchy of the ontology is shown in Figure 2.

Anything entered in the ontology is a `Concept`. Instances of `Concept` are further classified into `Project`, `Requirement`, `ThingType`, and `OperationType`. The `Project` and `Requirement` classes are used to store the requirements of the system, so that our methodology is traceable, since any instance can be traced back to the originating requirement. `ThingType` and `OperationType` are the main types of objects found in any requirement. `ThingType` refers to acting units and units acted upon, while `OperationType` involves the types of actions performed by the acting units. Each `ThingType` instance can be an `Actor`, an `Object`, or a `Property`. `Actor` refers to the actors of the project, including users, the system itself or any external systems. Instances of type `Object` include any object or resource of the system that receives some action, while `Property` instances include all modifiers of objects or actions. `OperationType` includes all possible operations, including `Ownership` that expresses possession (e.g. “each user has an account”), `Emergence` that implies passive transformation (e.g. “the posts are sorted”), `State` that describes the status of an `Actor` (e.g. “the user is logged in”), and `Action` describes an operation performed by an `Actor` on some `Object` (e.g. “the user creates a profile”).

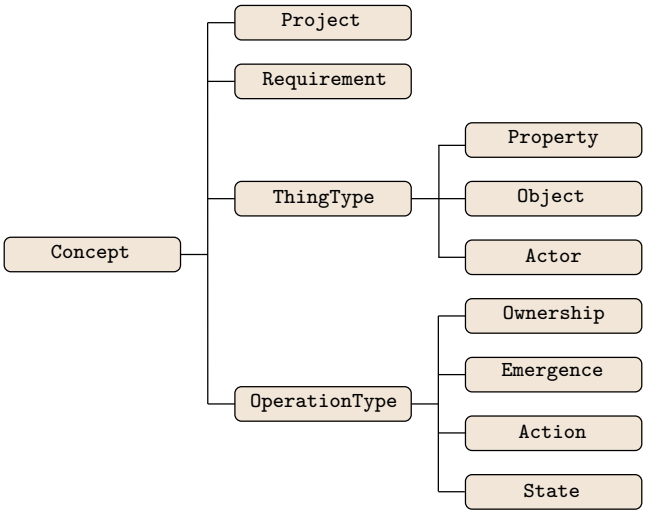

Fig. 2 Static Ontology of Software Projects

The possible interactions between the different concepts of the ontology, i.e. the relations between the ontology (sub)classes, are defined using *properties*. The properties of the static ontology are shown in Figure 3. Note that each property also has its inverse one (e.g. `has_actor` has the inverse `is_actor_of`). Figure 3 includes only one of the inverse properties and excludes properties involving `Project` and `Requirement` for simplicity.

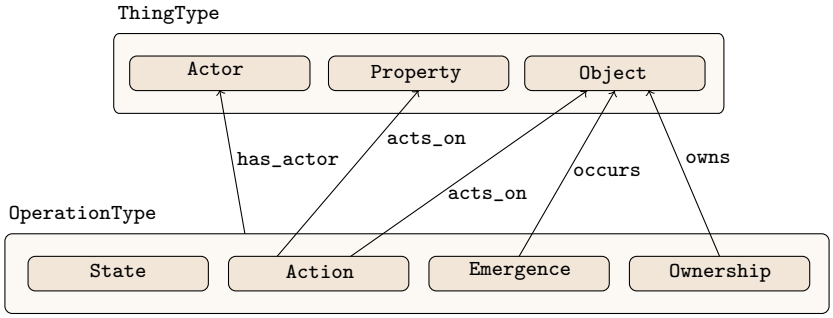

Fig. 3 Properties of the Static Ontology

An instance of `Project` can have many instances of `Requirement`, while each `Requirement` connects to several `ThingType` and `OperationType` instances. The remaining properties define relations between these instances. `OperationType` instances connect with instances of `Actor` via the `has_actor` property. Operations also connect to objects if are transitive. Thus, each `Action` is connected to instances of type `Object` or `Property` via the `acts_on` property, while `Emergence` `occurs` on an `Object` and `Ownership` is connected

with objects via the `owns` and `owned_by` properties. Finally, the non-transitive `State` operation does not connect to any `Object` or `Property`.

Extracting Artefacts from Functional Requirements The ontology described in the previous paragraph is instantiated using functional requirements. The requirements have to be *annotated* and the annotations are used to map to the OWL classes of the ontology. Although annotation schemes similar to the one in (Roth et al, 2015) integrate valuable NLP information, they are too complex for the user. Thus, we use the second level of the hierarchical annotation scheme defined by Roth et al (2014). As a result, our annotation scheme includes only four types of entities and three types of relations among them. In specific, any entity can be an Actor, an Action, an Object, or a Property. The defined relations include: IsActorOf declared from Actor to Action, ActsOn defined from Action to Object or from Action to Property, and HasProperty defined from Actor to Property or from Object to Property or from Property to Property. Thus, using this annotation scheme, we have developed a tool for adding, modifying, and annotating functional requirements. Our tool is called Requirements Editor and is built as an Eclipse plugin using the SWT. A screenshot is shown in Figure 4.

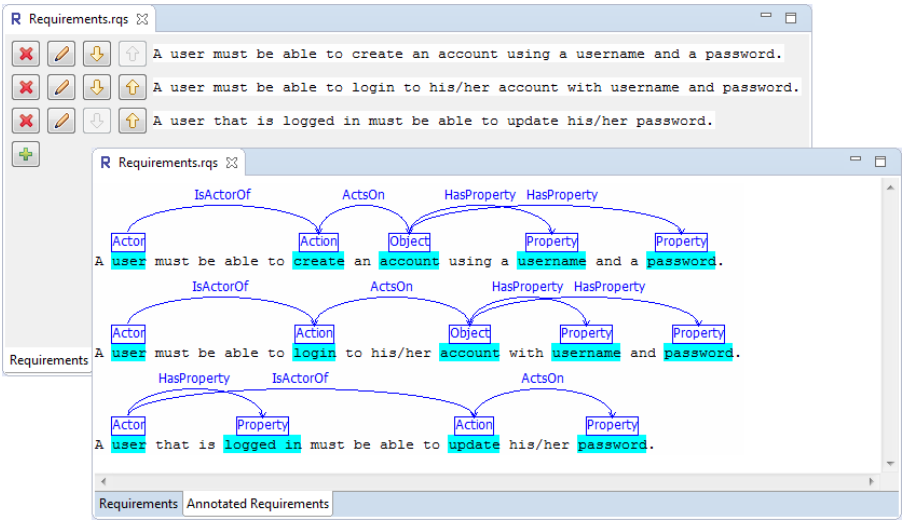

Fig. 4 Screenshot of the Requirements Editor

The tool is a multi-page editor for files with .rqs format. In the first page (shown in the top left of Figure 4), the user can add, delete, or modify functional requirements. The second page of the editor (shown in the bottom right of Figure 4) refers to the annotations of the requirements. The supported functionality includes adding and deleting entity and relationship annotations.

Given that the structure of functional requirements usually follows the SVO motif (as in Figure 4), annotating them is intuitive. However, the procedure

of annotating several requirements can be tiresome for the user. Therefore, we have constructed an NLP parser to automatically map software requirements to concepts and relations in the ontology. The parser, which is thoroughly described in (Roth et al, 2015), operates in two stages. The first stage is the *syntactic analysis* for each requirement. Input sentences are initially split into tokens, the grammatical category of these tokens is identified and their base types are extracted, to finally identify the grammatical relations between the words. The second stage involves the *semantic analysis* of the parsed sentences. This stage extracts semantic features for the terms (e.g. part-of-speech, relation to parent lemma, etc.) and employs a classification algorithm to classify each term to the relevant concept or operation of the ontology.

Upon annotation, the user can select the option to export the annotations to the static ontology. The mapping from the annotation scheme to the concepts of the static ontology is straightforward. Actor, Action, and Object and Property annotations correspond to the relevant OWL classes. Concerning relations, IsActorOf instantiates the `is_actor_of` and `has_actor` properties, ActsOn instantiates `acts_on` and `receives_action`, and HasProperty corresponds to the `has_property` and `is_property_of` properties. The rest of the properties (e.g. `project_has`, `consists_of`) are instantiated using the information of each functional requirement and of the name of the software project.

3.2.2 Dynamic View of Software Projects

An Ontology for the Dynamic View of Software Projects In this paragraph, we present an ontology that captures the dynamic view of a system. The main elements of dynamic representations are flows of actions among system objects. Using OWL, actions can be represented as classes and flows can be described using properties. The class hierarchy of the ontology is shown in Figure 5.

Anything entered in the ontology is a `Concept`. Instances of class `Concept` are further divided in the types of `Project`, `ActivityDiagram`, `AnyActivity`, `Actor`, `Action`, `Object`, `Condition`, `Transition` and `Property`. The class `Project` refers to the project analyzed while `ActivityDiagram` stores each diagram of the system, including not only activity diagrams, but also storyboards and generally any diagrams with dynamic flows of actions. As in the static ontology, `Project` and `ActivityDiagram` can be used to ensure that the concepts extracted from the ontology can be traced in the original diagram representations, allowing even to reconstruct them.

Activities are the main building blocks of dynamic system representations. The activities of a diagram instantiate the OWL class `AnyActivity`. This class is further distinguished in the subclasses `InitialActivity`, `FinalActivity`, and `Activity`. `InitialActivity` refers to the initial state of the diagram and `FinalActivity` refers to the final state of the diagram, while `Activity` holds any other activities of the system. Any action of the system may also require one or more input properties, stored in class `Property`. For instance, performing a “Create account” may require a “username” and a “password”. In this case, “username” and a “password” are instances of class `Property`.

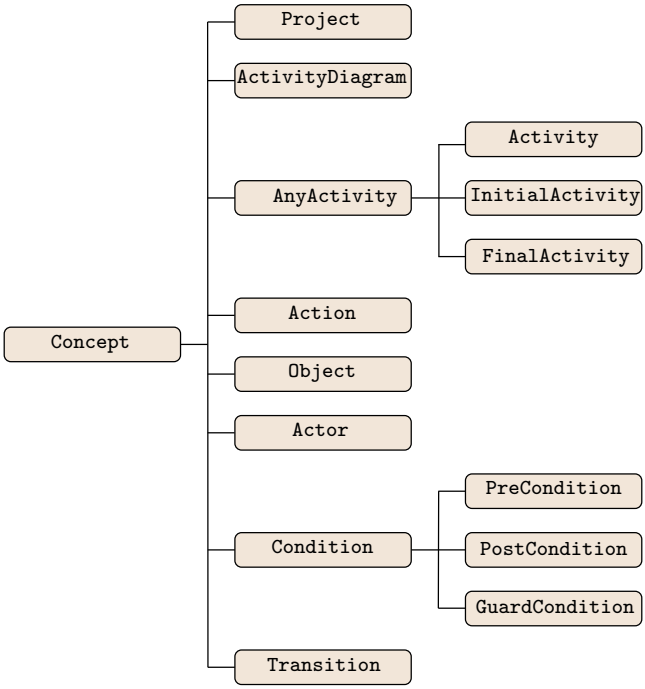

Fig. 5 Dynamic Ontology of Software Projects

The flow of activities in storyboards or activity diagrams is described using transitions. The OWL class `Transition` describes the flow from one instance of `Activity` to the next instance of `Activity` as derived by the corresponding diagram. Each `Transition` may also have a `Condition`. The ontology has three subclasses of `Condition`: `PreCondition`, `PostCondition`, and `GuardCondition`. The first two refer to conditions that have to be met before (`PreCondition`) or after (`PostCondition`) the execution of the activity flow of the diagram, while `GuardCondition` is a condition that "guards" the execution of an activity of the system along with the corresponding answer. For example, "Create account" may be guarded by the condition "is the username unique? Yes", while the opposite `GuardCondition` "is the username unique? No" shall not allow executing the "Create account" activity.

The properties of the ontology define the possible interactions between the different classes, involving interactions at inter-diagram level and relations between elements of a diagram. The properties of the dynamic ontology are illustrated in Figure 6, including only one of the inverse properties and excluding the `Project` and `ActivityDiagram` properties for simplicity. Each project can have one or more diagrams and each diagram has to belong to a project. Additionally, each diagram may have a `PreCondition` and/or a `PostCondition`. An instance of `ActivityDiagram` has elements of the five classes `Actor`, `AnyActivity`, `Transition`, `Property`, and `Condition`.

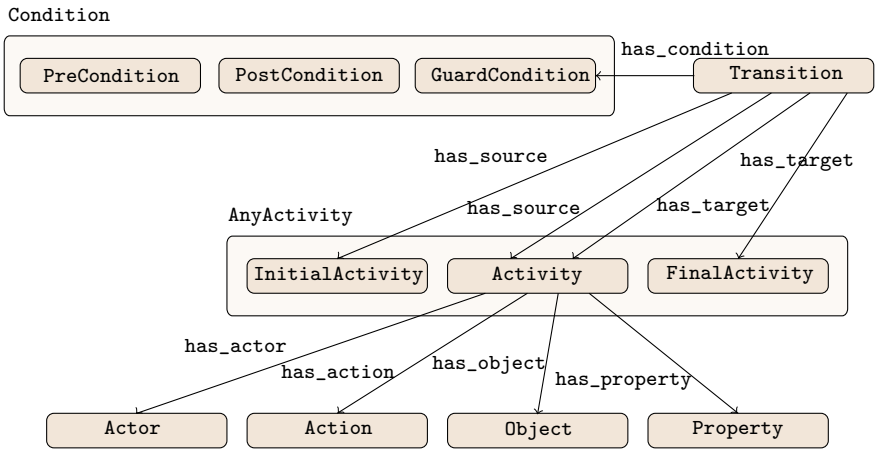

Fig. 6 Properties of the Dynamic Ontology

The different relations of ontology classes are actually forming the main flow as derived from diagram elements. Thus, `Activity` instances are connected with each other via instances of type `Transition`. Any `Transition` has a source and a target `Activity` (properties `has_source` and `has_target`, respectively), and it may also have a `GuardCondition` (property `has_condition`). Finally, any `Activity` is related to instances of type `Property`, while any `GuardCondition` has an opposite one, connected to each other via the bidirectional property `is_opposite_of`. Finally, any `Activity` is connected to an `Actor`, an `Action`, and an `Object` via the corresponding properties `has_actor`, `activity_has_action`, and `activity_has_object`.

Extracting Dynamic Flow from Storyboards As already discussed, we require a representation for the dynamic view of the system that is compliant with the RESTful paradigm. Therefore, we design this representation in the form of storyboards. Storyboards are dynamic system scenarios that describe flows of actions in software systems. A storyboard diagram is structured as a flow from a *start node* to an *end node*. Between the start and the end node, there are *actions* with their *properties*, and *conditions*. All nodes are connected with edges/paths. We have designed and implemented a tool for creating and editing storyboard diagrams, as an Eclipse plugin using the Graphical Modeling Framework (GMF), named Storyboard Creator. A screenshot of the tool including a storyboard is shown in Figure 7.

Storyboard Creator is a graphical editor for files with .sbd format. The tool includes a canvas for drawing storyboards, a palette that can be used to create nodes and edges, and an outline view that can be used to scroll when the diagram does not fit into the canvas. It also supports validating diagrams using several rules, such as the fact that each diagram must have at least one action node, each property must connect to exactly one action, etc.

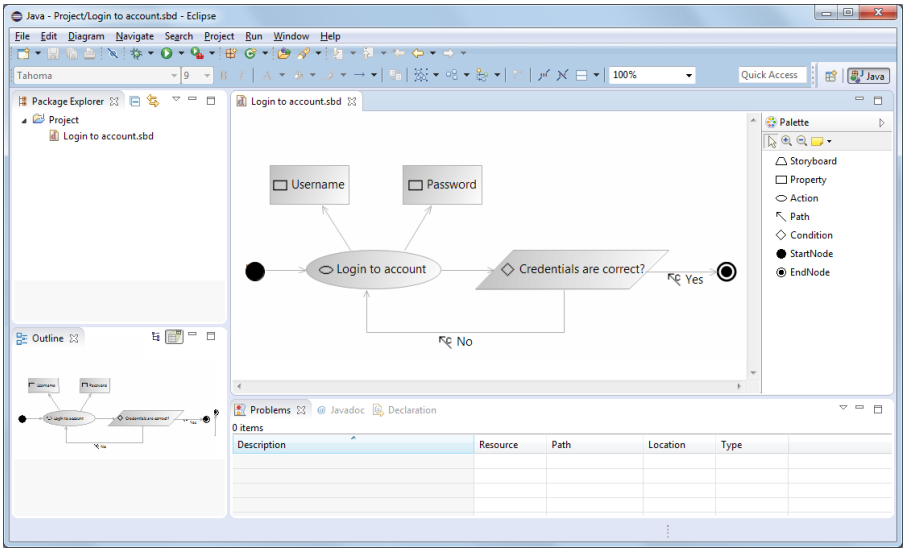

Fig. 7 Screenshot of the Storyboard Creator

The storyboard of Figure 7 includes an action "Login to account" which also has two properties that define the elements of "account", a "username" and a "password". Conditions have two possible paths. For instance condition "Credentials are correct?" has the "Yes" path that leads to the end node and the "No" path that leads back to the login action requiring new credentials.

Mapping storyboard diagrams to the dynamic ontology is straightforward. Storyboard actions are mapped to the OWL class `Activity` and they are further split into instances of `Action` and `Object`. Properties become instances of the `Property` class and they are connected with the respective `Activity` instances via the `has_property` relation. The paths and the condition paths of storyboards become instances of `Transition`, while the storyboard conditions split into two opposite `GuardConditions`. An example instantiation for the storyboard of Figure 7 is shown in Table 1.

Table 1 Example Instantiated OWL Classes for the Storyboard of Figure 7

OWL Class	OWL Instances
`Activity`	`Login_to_account`
`Property`	`Username`,`Password`
`Transition`	`FROM__StartNode__TO__Login_to_account`, `FROM__Login_to_account__TO__EndNode` `FROM__Login_to_account__TO__Login_to_account`
`GuardCondition`	`Credentials_are_correct__PATH__Yes`, `Credentials_are_correct__PATH__No`
`Action`	`login`
`Object`	`account`

3.3 Aggregated Ontology of Software Projects

Upon having constructed the two ontologies for the dynamic and static view of the system, an aggregated ontology that composes the concepts of those ontologies is needed. The elements of the aggregated ontology actually form an initial version of the *Computationally Independent Model (CIM)* of the software project that is provided as input to the MDE engine module. Thus, the main building block of this ontology is the RESTful resource. Additionally, since resources are created, retrieved, and deleted via actions, the ontology includes the main actions that are performed on resources, as well as any parameters required for these actions. The class hierarchy of the aggregated ontology is shown in Figure 8.

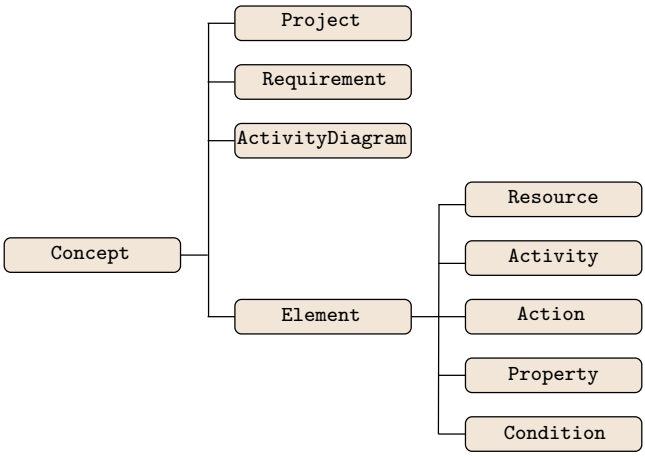

Fig. 8 Aggregated Ontology of Software Projects

Instances of `Concept` are divided in the classes `Project`, `Requirement`, `ActivityDiagram`, and `Element`. `Project` refers to the software project instantiated, while `Requirement` and `ActivityDiagram` are used to hold the requirements and diagrams of the static and the dynamic ontology respectively. These instances ensure that the results of the ontology are traceable.

Any other ontology `Concept` is an `Element` of the project. Instances of type `Element` are divided into the subclasses `Resource`, `Activity`, `Action`, `Property`, and `Condition`. The `Resource` is the building block of any RESTful system, while `Action` is used to hold the CRUD actions performed on resources. `Activity` refers to an activity of the system (e.g. "create bookmark") that is connected to a `Resource` (e.g. "bookmark") and a CRUD `Action` (e.g. "create"). `Property` refers to a parameter required for a specific activity (e.g. "bookmark name" may be required for the "create bookmark"). Finally, an instance of `Condition` holds criteria that have to be met for an `Activity` to be executed. The properties of the aggregated ontology are shown in Table 2.

Table 2 Properties of the Aggregated Ontology

OWL Class	Property	OWL Class
Project	has_requirement	Requirement
Requirement	is_requirement_of	Project
Project	has_activity_diagram	ActivityDiagram
ActivityDiagram	is_activity_diagram_of	Project
Project	has_element	Element
Element	is_element_of	Project
Requirement/ ActivityDiagram	contains_element	Element
Element	element_is_contained_in	Requirement/ ActivityDiagram
Resource	has_activity	Activity
Activity	is_activity_of	Resource
Resource	has_property	Property
Property	is_property_of	Resource
Activity	has_action	Action
Action	is_action_of	Activity
Activity	has_condition	Condition
Property	is_condition_of	Activity
Activity	has_next_activity	Activity
Activity	has_previous_activity	Activity

The properties of the aggregated ontology cover the main structure of the software project. The instances of the class `Element` along with these properties have to be as expressive as possible since they will be used to form the YAML file and subsequently the CIM of the project.

The properties including `Project`, `Requirement`, `ActivityDiagram`, and `Element` are used to ensure that any element of the diagram is traceable in the other two ontologies. The relations of the ontology classes are formed around two main subclasses of `Element`, `Resource` and `Activity`. This is quite expected since these two elements form the basis of a RESTful system. Any system `Resource` may be connected to instances of type `Property` and `Activity`, using `has_property/is_property_of` and `has_activity/is_activity_of` respectively. The class `Activity` is connected to instances of type `Action` (via the properties `has_action/is_action_of`) and of type `Condition` (via the properties `has_condition/is_condition_of`), since it is necessary to keep track of the CRUD verbs to be used as well as any conditions that have to be met in order for the activity to be valid. Transitions are handled using the properties `has_next_activity/has_previous_activity`.

The properties are also visualized in Figure 9 (excluding the properties relevant to `Project`, `Requirement`, and `ActivityDiagram` and including only one of the inverse properties for simplicity), where it is clear that `Resource` and `Activity` have central roles in the aggregated ontology.

The aggregated ontology is instantiated using the information provided by the static and dynamic ontologies of software projects. The static ontology contains several classes that refer to the static view of the system. Among them, we focus on actions performed on objects and any properties of these

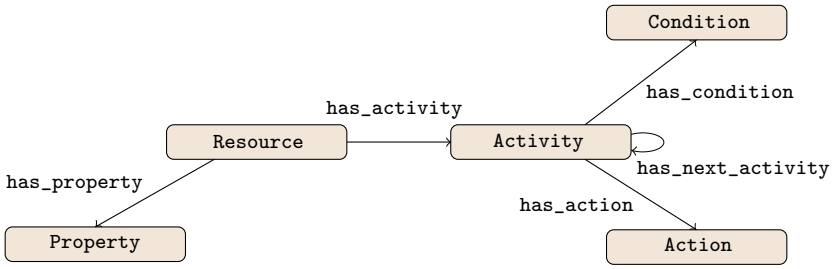

Fig. 9 Properties of the Aggregated Ontology

objects. In the static ontology, these elements are represented by the OWL classes `OperationType`, `Object`, and `Property`. Concerning the dynamic elements of a software system, the corresponding ontology covers not only actions, objects, and properties, but also the conditions of actions. The corresponding OWL classes are Action, `Object`, `Property`, and `GuardCondition`. Apart from the above classes, we also keep track of the `Project` that is instantiated, as well as the instances of type `Requirement` and `ActivityDiagram` derived from the static and dynamic ontologies respectively. These three classes ensure that our ontologies are traceable and strongly linked to one another. The mapping of OWL classes from the static and dynamic ontologies to the aggregated ontology is shown in Table 3.

Table 3 Classes Mapping from Static and Dynamic Ontologies to Aggregated Ontology

OWL Class of Static Ontology	OWL Class of Dynamic Ontology	OWL Class of Aggregated Ontology
`Project`	`Project`	`Project`
`Requirement`	-	`Requirement`
-	`ActivityDiagram`	`ActivityDiagram`
`OperationType`	`Action`	`Activity`
-	`GuardCondition`	`Condition`
`Object`	`Object`	`Resource`
`Property`	`Property`	`Property`

As shown in this Table, instances of `Requirement` and `ActivityDiagram` are propagated to the aggregated ontology, while `Project` is used to ensure that the two ontology instantiations refer to the same project. Concerning the remaining classes of the aggregated ontology, several of them require merging the elements from the two ontologies. Thus, any `Object` of the static ontology and any `Object` of the dynamic ontology are added to the aggregated ontology one after another. If at any point an instance already exists in the aggregated ontology then it is simply not added. However, any properties of this instance are also added (again if they do not exist); this ensures that the ontology is fully descriptive, yet without any redundant information. The mapping for OWL properties is shown in Table 4.

Table 4 Properties Mapping from Static and Dynamic Ontologies to Aggregated Ontology

OWL Property of Static Ontology	OWL Property of Dynamic Ontology	OWL Property of Aggregated Ontology
project_has	-	has_requirement
is_of_project	-	is_requirement_of
-	project_has_diagram	has_activity_diagram
-	is_diagram_of_project	is_activity_diagram_of
consists_of	diagram_has	contains_element
consist	is_of_diagram	element_is_contained_in
receives_action	is_object_of	has_activity
acts_on	has_object	is_activity_of
has_property	has_property*	has_property
is_property_of	is_property_of*	is_property_of
-	has_action	has_action
-	is_action_of	is_action_of
-	has_condition*	has_condition
-	is_condition_of*	is_condition_of
-	has_target	has_next_activity
-	has_source	has_previous_activity

* derived property

Note that some properties are not directly mapped among the ontologies. In such cases, properties can be derived from intermediate instances. For example, the aggregated ontology property `has_property` is directed from `Resource` to `Property`. In the case of the dynamic ontology, however, properties are connected to activities. Thus, for any `Activity`, e.g. "Create bookmark", we have to first find the respective `Object` ("bookmark") and then upon adding it to the aggregated ontology, we have to find the `Property` instances of the `Activity` (e.g. "bookmark name") and add them to the ontology along with the respective connection. This also holds for the condition properties `has_condition` and `is_condition_of`, which are instantiated using the instances of `GuardCondition` of the preceding `Transition`.

3.4 Ontology Software Artefacts to a YAML Representation

Upon instantiating the aggregated ontology, the next step is the transformation from the ontology to the CIM of the envisioned service. Since a CIM may be overwhelming for the end-user, we first design a representation in YAML[14], which shall effectively describe the conceptual elements of the CIM and provide the user with the ability to modify (fine-grain) the model. YAML was selected as the representation since it is intuitive, human-readable (i.e. in comparison to XML) and is supported by several tools. YAML supports several well-known data structures, since it is designed to be easily mapped to programming languages. In our case, we use lists and associative arrays (i.e. key-value structures) to create a structure for resources, their properties and the different types of information that have to be stored for each resource. The schema of our representation is shown in Figure 10, where the main element is the RESTful resource (`cim.Resource`). A project consists of a list of resources.

[14] `http://yaml.org/`

```
- !!cim.Resource
  Name: String
  IsAlgorithmic: Boolean
  CRUDActivities: List of Create, Read, Update, and/or Delete
  Properties:
  - Name: String
    Type: Integer/Float/String/Boolean/null
    Unique: Boolean
    NamingProperty: Boolean
  - ...
  RelatedResources: List of String
```

Fig. 10 Schema of the YAML Representation

Several fields are defined for each resource, each with its own type and allowed values. At first, each resource must have a name, which also has to be unique. Additionally, every resource may be either algorithmic (i.e. requiring some code to be written or some external service to be called) or non-algorithmic. This is represented using the `IsAlgorithmic` boolean field. CRUD verbs/actions (or synonyms that can be translated to a CRUD verb) are usually not applied on algorithmic resources. There are four types of activities that can be applied to resources, in compliance with the CRUD actions (Create, Read, Update, and Delete). Each resource may support one or more of these actions, represented as a list (`CRUDActivities`).

Resources also have properties, which are defined as a list of objects. Each property has a `Name`, which is alphanumerical, as well as a `Type`, which corresponds to the common data types of programming languages, i.e. integers, float, strings, and booleans. Furthermore, each property has two boolean fields: `Unique` and `NamingProperty`. The former denotes whether the property has a unique value for each instance of the resource, while the latter denotes whether the resource is named after the value of this property. For example, a resource "user" could have the properties "username" and "email account". In this case, the "username" would possibly be unique, while "email account" could or could not be unique (e.g. a user may be allowed to declare more than one email accounts). Any instance of "user", however, should also be uniquely identified in the system. Thus, if we do not allow two users to have the same username, we could declare "username" as a naming property. Finally, each resource may have related resources. The field `RelatedResources` is a list of alphanumerical values corresponding to the names of other resources.

Extracting information from the aggregated ontology and creating the corresponding YAML file is a straightforward procedure. At first, instances of the OWL class `Resource` can directly be mapped to YAML objects of type `cim.Resource`. Each resource is initially considered non-algorithmic. The flow of activities and conditions is used to find the types of verbs that are used on any resource as well as the related resources. Thus, for example, given an activity "Add bookmark" followed by an activity "Add tag", one may identify two resources, "bookmark" and "tag", where "tag" is also a related resource for "bookmark". Additionally, both "bookmark" and "tag" must have the

"Create" CRUD activity enabled, since the verb "add" implies creating a new instance. The type for each verb is recognized using a lexicon.

Whenever an action verb cannot be classified as any of the four CRUD types, a new algorithmic resource is created. Thus, for example, an activity "Search user" would spawn the new algorithmic resource "userSearch", and connecting it as a related resource of the resource "user". Upon applying the transformations of the MDE engine, it would then be possible to manually write code for the function of this algorithmic resource. Finally, the properties of the resources are mapped to the `Properties` list field.

4 Model-Driven Engineering Module

Model Driven Engineering (MDE) is the cornerstone technology used in the presented two-fold mechanism to introduce automation. More specifically, our MDE engine follows our novel 2D MDE architecture, which is illustrated in Figure 11 (its full theoretical definition is out of the scope of this paper).

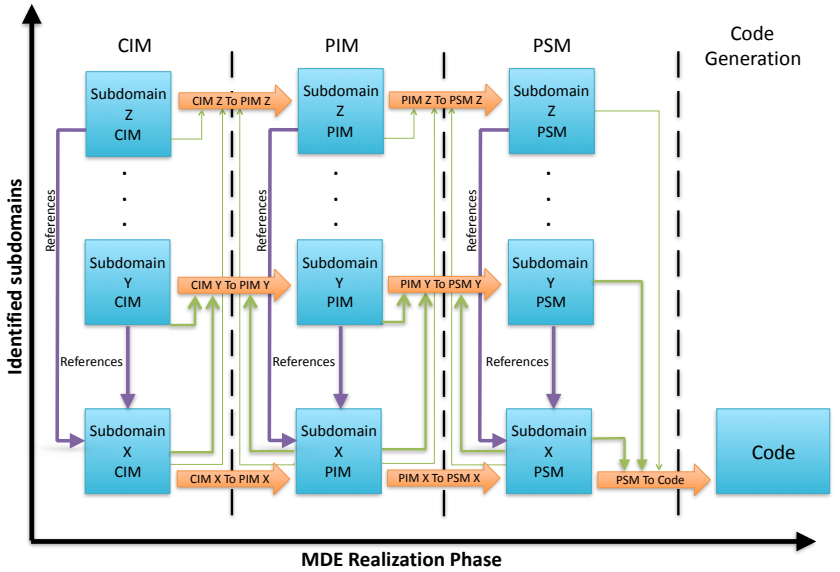

Fig. 11 Abstract 2D MDE Engine Architecture.

Our engine embeds end-to-end the benefits of problem space Domain Compartmentalization, which allows MDE designers and users to work with smaller models, meta-models and transformations, hence lowering the perceived complexity during the construction of MDE engines as well as their use (Störrle, 2014; Moody, 2009). The need for cross domain expertise is also decreased, while meta-modeler effectiveness is increased, since the initial complex domain is split in simpler congruent ones (Sweller, 1994). Productivity is further improved by introducing parallelism in both the MDE engine development phase as well as its usage, because independent subdomains yield independent meta-models and corresponding transformation chains.

In summary the 2D MDE architecture evolves over two axis. The horizontal one, the Realization Phase Axis, concerns the phases of Model Driven Engineering to be included in the 2D MDE engine. The connecting elements along this Axis are the model-to-model or model-to-text transformations. The type of the desired MDE methodology to be used determines the number of the realization phases. In this case MDA is used, as introduced by the Open Management Group (OMG) and up-to-date is the most popular case of MDE.

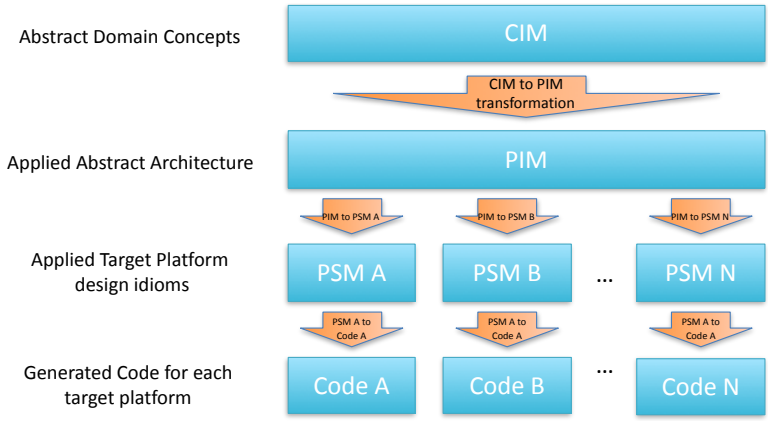

Fig. 12 Model Driven Architecture Phases

Hence, the Realization Phase Axis (or in short the Realization Axis) comprises the four MDA phases (illustrated in Figure 12), namely:

- *Computational Independent Model (CIM):* During the first phase the developer models the system using an abstract domain specific language, which hides the design and implementation details. The outcome of this process is the CIM model of the envisioned system that contains all the domain concepts to be included in it as well as their conceptual interconnections.
- *Platform Independent Model (PIM):* Once the developer concludes with the CIM model of the system, an automated chain of transformations takes place. Firstly, the MDA engine performs a Model-to-Model (M2M) transformation in order to produce the corresponding PIM model. This CIM-to-PIM transformation refines all the CIM model elements by applying the envisioned system design and architecture. However, this is done in an abstract way, without targeting any specific implementation platform at this stage. That is, an architectural style is applied, such as *Client-Server*, without any specific programming language/platform design idioms. Usually, there is one to one relationship between CIM and PIM models; every CIM model is transformed to one PIM model of a specific architecture.
- *Platform Specific Model (PSM):* Once the PIM model is in place, a second M2M transformation takes place, the PIM-to-PSM. This time the architecture of the system is further refined by taking into account the target

execution platform of the envisioned system e.g. Java Application with an underlying SQL database. For example, some PIM elements are further revised using some of the target language design idioms, available libraries etc. At this stage, there is usually a one-to-many relationship between one PIM and several possible target platform PSMs.
– *Code generation:* The final step is the production of the envisioned system's code. Hence, a Model-to-Text (M2T) transformation takes place. Its input is a PSM model and its output is the generated code, where may need to fill in extra code for additional/case specific functionality. The level of completeness, is greatly affected by how effectively the envisioned system can be modeled using the CIM meta-model (domain specific language).

On the other hand, the vertical axis or the Subdomain Axis comprises the meta-models of each subdomain that are derived after applying Domain Compartmentalization to a complex domain. In Figure 11 for example, there are three such meta-models X, Y and Z that together constitute the initial complex domain. Along the Subdomain Axis, the connecting elements are references from one meta-model to the others. Concepts that belong to meta-models Y and Z refer to concepts of meta-model X, while there are no references towards or among them. That is, meta-model X is independent of Y and Z since its definition does not need their concepts, Y and Z depend on X, while they are independent to each other. Of course, there can be many simpler or more complex scenarios. For example some subdomains may depend on multiple other ones, referencing multiple other meta-models. The conceptual clarity of each meta-model and the population of cross-meta-model relationships depend on successful domain compartmentalization.

The domain upon which our MDE engine brings automation is compartmentalized to four subdomains. The first one is the REST subdomain, which provides the structural concepts for the producible systems. The second is the Basic-Authentication that alters the behavior of the underlying referenced REST components so as to embed authentication capabilities or not. The third subdomain concerns common database keyword searching, whilst the fourth automates interconnection of envisioned systems with existing Web Services in the web. The following subsections present in more depth these aforementioned meta-models in more depth.

4.1 The REST architectural Style

Since this two-fold mechanism aims to produce RESTful services, it embeds to the produced services primarily the REST architectural style. In this case, the Richardson's Maturity Model (RMM) is used as a model of the REST architectural Style. According to this model, a web service lies at the 3rd level of RMM (in other words it is said to be RESTful) if and only if the following design rules are not violated (illustrated in Figure 13).

– Rule 1: The web service building block has to be a *resource* with a unique URI. Hence, in this resource oriented approach each service comprises several resources, each of which models a small part of the service functionality.

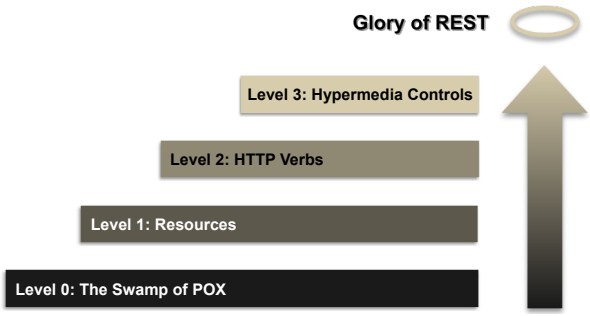

Fig. 13 Richardson's Maturity Model[15]

- Rule 2: The web API of each resource respects the semantics of each HTTP verb. That is, the *POST* verb is used to create new resources, *PUT* to update existing ones, *GET* to retrieve existing ones and *DELETE* to delete existing resources.
- Rule 3: The resources have to be semantically interwoven to each other with *hypermedia links*. That is, every time the server sends back a response to its client, it embeds into this response some URIs with possible follow-up actions for this client.

The last RMM rule, the so called HATEOAS, is the feature of REST style that differentiates it the most from other Web Architectures and greatly helps to avoid coupling between services and their clients. Namely, the service does not need to publish a predefined interface, but rather responds to client requests that also embed hypermedia links, linking to resources the client can access. Hence, by complying to the RMM REST model, the server may change its interface, e.g. by adding or removing hypermedia links to/from its responses, however the client will still be able to find these available links that match its communication protocol and advance the application state. The trade-off of this reduced coupling is that server/client communication can be sometimes too interactive, however such side effects can be contemplated using server-side optimizations (Newman, 2015).

4.2 Modeling RESTful Services

The CIM meta-models that comprise the domain specific language of the presented 2D MDE engine, are four specific purpose meta-models. As already discussed, our MDE engine models RESTful services that may embed essential non-CRUD functionality, such as Basic Authentication, interoperability with existing 3rd party services as well as database keyword-searching functionality. That non-CRUD functionality is meta-modeled within the corresponding three meta-models that refer to the core REST one, which provides the structure of the produced services.

[15] `http://martinfowler.com/articles/richardsonMaturityModel.html`

Our mechanism includes this essential non-CRUD functionality in order to increase the developer productivity in several ways. Firstly, the selected non-CRUD functionality is wide-spread and very common in most Web Services, hence automating its implementation reduces the development effort. Secondly, it reduces the need for cross-domain expertise, since developers that do not know how to implement searching mechanisms or are not acquainted with Authentication mechanisms, can embed such functionality in their services with minimal domain knowledge. Lastly, since this essential functionality part is automated, it always embeds the same code qualities and its correctness and validity does not depend of day-to-day developer programming habits. The following subsections present these meta-models and explain their elements and their semantics.

4.2.1 REST CIM Meta-Model

The primal meta-model has to embed the needed concepts to model resources, their common web API and their web interweaving. Figure 14 illustrates the principal REST CIM meta-model part elements, which are explained below, modeled with the Ecore meta-model[16].

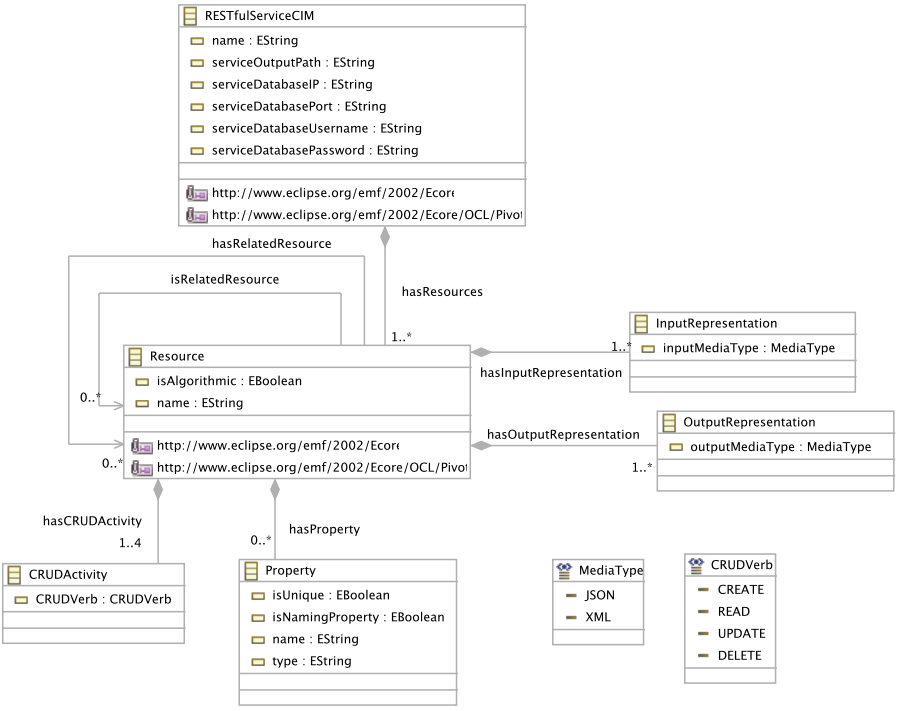

Fig. 14 S-CASE MDE primal CIM Meta-Model.

[16] `https://eclipse.org/modeling/emf/`

The *RESTfulServiceCIM* element is the root element of the core metamodel. It pretty much embeds the whole CIM. It comprises the envisioned system's name, the output code folder and the necessary login information for the database scheme that is going to be created and used as a local data repository. That is, a valid username/password combination of the database server that runs at the provided Port:IP combination.

The *Resource* element models an RMM resource. Obviously, this is the most important concept to allow resource-oriented modeling, and is associated with concepts that model the resource data, representations and possible actions. It embeds the resource's name and a boolean value to indicate whether the resource will model data and its primitive *CREATE*, *READ*, *UPDATE* or *DELETE* operations, or otherwise will embed in RESTful wrapper some sort of non-CRUD algorithm (e.g. calculate shortest path, pay by credit card etc.). One may notice the absence of a URI attribute for every resource. It is not necessary, since the MDE engine automatically computes the service's URIs, whilst the developer simply has to decide only where the service will be deployed. Each resource may also have some *Representation* elements that model various acceptable media formats available in the *MediaType* enumeration e.g. *JSON* or *XML*.

The *Property* element represents an attribute of a resource. Each such attribute must have a *name* and a *type* of type string and a boolean value that indicates whether the property is single-valued (multiplicity 1) or multivalued. Moreover, each resource must have exactly one *naming property*. That value of the naming property is included in resource lists retrieved by the clients, in order to be able to differentiate among different resources and pick the ones they wish to fully retrieve or act upon. Table 5 illustrates some hypermedia links of a shopping list. Without the naming property attribute, only the HTTP Verb and the Link would be included in the response. However, in this case, the developer who designed this resource selected its name as a naming property, thus the "Black/Red T-shirt" values are also included in the server response and help the client to decide on which item to act.

Table 5 Illustrative RESTful service's hypermedia links, included in response to a shopping list *GET* Request

HTTP Verb	Hypermedia Link	Naming Property Value
`GET`	`http://www.example.com/list/85/item101`	`Black T-Shirt`
`DELETE`	`http://www.example.com/list/85/item101`	`Black T-Shirt`
`GET`	`http://www.example.com/list/85/item102`	`Red T-Shirt`
`PUT`	`http://www.example.com/list/85/item102`	`Red T-Shirt`

The *CRUDActivity* element models an abstract form of each *CRUD* verb and can take any value modeled with the *CRUDVerb* enumeration. These elements form the common web API of any resource, strictly following the appropriate REST style semantics. Table 6 provides the intuitive mapping between the CRUD and HTTP verbs.

Table 6 CRUD to HTTP verbs mapping

CRUD Verb	HTTP Verb
CREATE	POST
READ	GET
UPDATE	PUT
DELETE	DELETE

The *Hypermedia* links, as dictated by the 3rd rule of RMM, are modeled with the two kinds of associations a resource element might have. These are the *hasRelatedResource* and *isRelatedResource* associations. The former models the capability of one resource to have zero or more related resources, whilst the latter models the capability of a resource to be related to zero or more other resources. Based on these relationships, when a client retrieves a representation of one resource, it also gets a list of hypermedia links of that resource's related resources, which allow the application state to forward in the RESTful manner. Table 5 illustrates for such hypermedia links each of which comprise the needed HTTP verb, the unique URI of the resource to be reached and its naming property as already discussed. Table 7 summarizes the core concepts of this CIM meta-model that model the principal REST design concepts.

Table 7 Modelling of principal REST design concepts

REST Concept	Satisfied by CIM element(s)
Resource Oriented Design	Resource and Representations
Common Web API	CRUDActivity
Hypermedia	hasRelatedResource and isRelatedResource

Apart from the structural constraints embedded in the presented meta-model, our MDE engine embeds also behavioral constraints designed using First Order Predicate Logic (FOL[17]), so as to be able to reason about the well formed-ness of instance models and impose validation checks to runtime instances. These behavioral constraints have been developed using the equivalent, to the FOL formulas, OCL[18] expressions within the Ecore meta-models.

Table 8 presents a few of the FOL formulas used in our MDE engine, the predicates of which are defined in Table 9. For example, the first one defines the uniqueness of each CRUD activity type a Resource may have. In FOL terminology it unambiguously states that: *"For any x, for any y, x is a resource and y is a Create Activity and y is Create Activity of Resource x, if and only if for any* y_1*, if* y_1 *is a Create Activity and* y_1 *is Create Activity of Resource x, implies that* y_1 *equals y"*. That way, the lower CASE of our mechanism is able to actively identify inconsistencies in developer input and guide him/her to correct them.

[17] https://en.wikipedia.org/wiki/First-order_logic

[18] https://en.wikipedia.org/wiki/Object_Constraint_Language

Table 8 Illustrative First Order Predicate Logic formulas

Nr.	First Order Predicate Logic
1	$(\forall x)(\forall y)(Rx \,\&\, Cy \,\&\, Fyx \;\equiv\; (\forall y_1)(Cy_1 \,\&\, Fy_1x \;\supset\; y_1 = y))$
2	$(\forall x)(Rx \,\&\, (\exists y)(Py \,\&\, Hyx) \;\equiv\; \neg Ax)$
3	$(\forall x)(Rx \,\&\, \neg Ax \;\equiv\; (\exists y)(Py \,\&\, Hyx \,\&\, Iy \,\&\, (\forall y_1)(Py_1 \,\&\, Hy_1x \,\&\, Iy_1 \;\supset\; y_1 = y)))$
4	$(\forall x)(\forall y)(\forall z)(Mx \,\&\, Ry \,\&\, Kyx \,\&\, Nz \,\&\, Lzy \;\equiv\; (\forall y_1)(Ry_1 \,\&\, Ky_1x \,\&\, Lzy_1 \;\supset\; y_1 = y))$

Table 9 Predicate Symbol Explanation

Predicate Symbol	Meaning
`R`	`is Resource`
`C`	`is Create Activity`
`F`	`is Create Activity of`
`P`	`is Property`
`H`	`is Property of`
`A`	`is Algorithmic Resource`
`I`	`is Naming Property`
`M`	`is RESTful CIM`
`K`	`is Resource of`
`N`	`is Resource Name`
`L`	`is Resource Name of`

On the other hand, Table 10 defines the corresponding OCL expressions that model this FOL formula. For example, OCL constraint 1 validates that each resource has unique activity types, constraint 2 checks whether all non-algorithmic resources have at least one property, constraint 3 makes sure that every non algorithmic resource has at least one naming property and finally constraint 4 validates the uniqueness of resource names a CIM may contain.

Table 10 Illustrative OCL constraints

Nr.	OCL constraint of CIM element	OCL constraint definition
1	`Resource`	`self.hasCRUDActivity->isUnique(CRUDVerb)`
2	`Resource`	`(self.isAlgorithmic = false) implies self.hasProperty->notEmpty()`
3	`Resource`	`self.hasProperty->notEmpty() implies self.hasProperty->one(isNamingProperty = true)`
4	`RESTfulServiceCIM`	`self.hasResources->isUnique(name)`

4.2.2 Basic Authentication Meta-Model

The Basic Authentication CIM meta-model part models the necessary security concepts in order to be able to embed automatically an authentication mechanism to the envisioned system. Specifically, these concepts allow the

developer to incorporate a password/username type authentication scheme. Since RESTful services are stateless (e.g. there is no session cookie), the password/username combination is communicated using the authorization HTTP header of client requests. Figure 15 illustrates the authentication meta-model. The grayed elements are the referenced REST CIM meta-model ones, already presented in Figure 14, hence are simplified in this diagram.

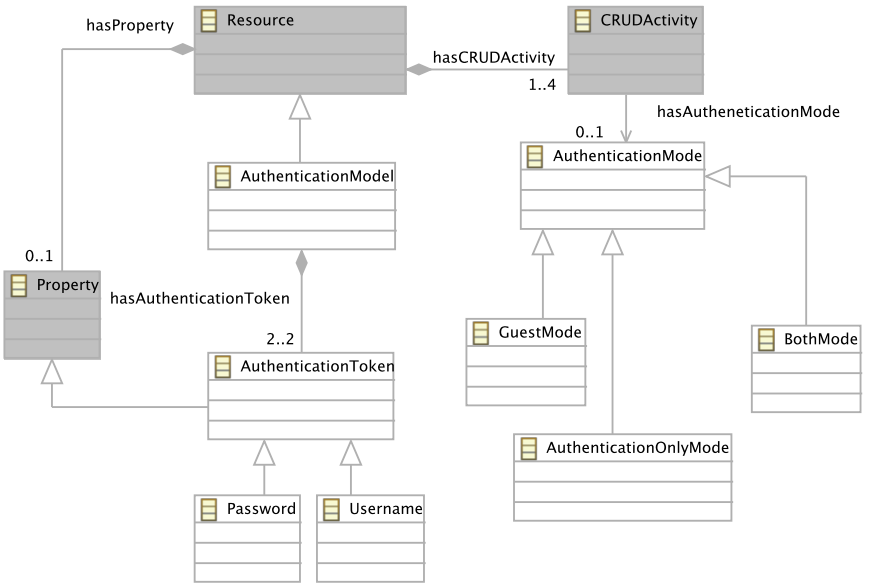

Fig. 15 Basic Authentication CIM Meta-Model.

The core authentication concept is the *AuthenticationModel* one. A RESTful service with an authentication scheme must have exactly one AuthenticationModel, which is a specialization of the Resource concept. The AuthenticationModel dictates that one of the service resources and two of its properties will be used as authentication tokens. Thus, since the Basic Authentication scheme is used, each AuthenticationModel has exactly two *AuthenticationTokens*, which are a specialization of the aforementioned Property concept. Moreover, AuthenticationTokens are further specialized to be either a *Password* or a *Username*. Table 11 illustrates such an AuthenticationModel.

Table 11 Illustrative Resource that is used as AuthenticationModel

Resource Name	Property Name	Basic Authentication Specialization
`UserAccount`	`username`	`Username AuthenticationToken`
`UserAccount`	`pass`	`Password AuthenticationToken`
`UserAccount`	`email`	`-`

One should note that an AuthenticationModel may also have other properties beyond its AuthenticationTokens, such as an email property etc., as in the example of Table 11.

Since the web API of each Resource, as already discussed, comprises its CRUDActivities, the remaining concepts concern the fine tuning of the Basic Authentication scheme to be used per CRUDActivity. If Basic Authentication is applied to a RESTful service, then each CRUDActivity must have exactly one *AuthenticationMode*. This mode precisely defines whether a client has to be authenticated or not in order to use the functionality of an underlying CRUDActivity. In our CIM meta-model 3 modes have been defined. The *GuestMode* does not perform any authentication checks and allows any client to make requests. The *AuthenticationOnlyMode* performs authentication checks and allows only successfully authenticated clients to make requests. Finally, there is a mixed mode (*BothMode*), which allows guest clients to make requests but also performs authentication checks if a client has its authorization HTTP header set. Table 12 illustrates some CRUDActivities of a Product resource and their Authentication Mode.

Table 12 Authentication modes of several CRUDActivities example

Resource Name	CRUDActivity	CRUDVerb	Authentication Mode
`Product`	`createProduct`	`CREATE`	`BothMode`
`Product`	`updateProduct`	`UPDATE`	`AuthenticationOnlyMode`
`Product`	`readProduct`	`READ`	`BothMode`
`Product`	`deleteProduct`	`DELETE`	`AuthenticationOnlyMode`

Like in the core REST meta-model, OCL constraints are defined to check the well-formedness of an instance CIM model regarding its Basic Authentication concepts. Such constraints ensure that only one AuthenticationModel exists, which has exactly two AuthenticationTokens, one of which is a Username and the other one is a Password. Other constraints validate whether there is exactly one AuthenticationMode for every CRUDActivity of the envisioned system etc.

4.2.3 Database Searching Meta-Model

The database searching CIM meta-model part, models the necessary concepts to automate widely-used keyword searching. It comprises concepts that allow the developer to model some resources to be able to search other resources' properties given a keyword. Hence, the generated service will embed automatically the needed functionality to handle client search requests and send back links of any matched artefacts. Figure 16 illustrates these Search meta-model. Again, the referenced REST CIM meta-model concepts, already presented in previous subsection, are grayed out and simplified.

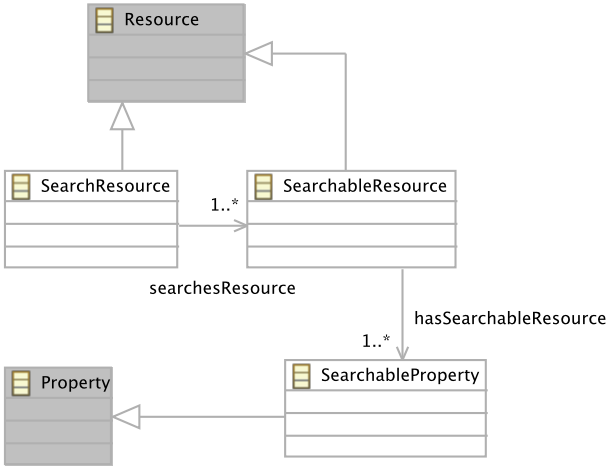

Fig. 16 Database Searching CIM Meta-Model.

The most important concept of this non-CRUD functionality layer is the *SearchResource* one. This concept is a specialization of Resource and models algorithmic resources that embed a non-CRUD algorithm, which is able to handle incoming keyword search requests, make the appropriate queries to the database and bundle a list of links to matched artefacts. The *SearchableResource* is also a specialization of the Resource concept that models a resource, whose properties (some or all) will be searchable by some SearchResource. These properties are modeled with the *SearchableProperty* concept, which is a specialization of the Property one. The defined OCL constraints validate that every SearchResource searches at least one SearchableResource, which in turn has at least one SearchableProperty.

Table 13 illustrates two SearchResources and the SearchableResources' SearchableProperties they search. One may note that a SearchResource may search properties of different SearchableResources and that a SearchableProperty might be searchable by more than one SearchResource.

Table 13 Database-Searching modelling example

SearchResource Name	SearchableResource Name	SearchableProperty Name
`ProductSearch`	`Product`	`description`
`ProductSearch`	`Product`	`name`
`ProductSearch`	`Tag`	`tagName`
`TagSearch`	`Tag`	`tagName`

4.2.4 3rd Party Wrapper Meta-Model

The remaining concepts of the CIM meta-model, concern the interoperability with existing 3rd party RESTful services. In other words, this part of the

CIM meta-model allows the developer to model some of his/her envisioned system resources as RESTful clients, each of which will make requests to one specific 3rd party RESTful service. Additionally, it is possible to model the input/output of each service and decide whether its responses should be stored in the local database. Figure 17 illustrates these concepts and their associations. The REST CIM meta-model referenced concepts are grayed out and simplified.

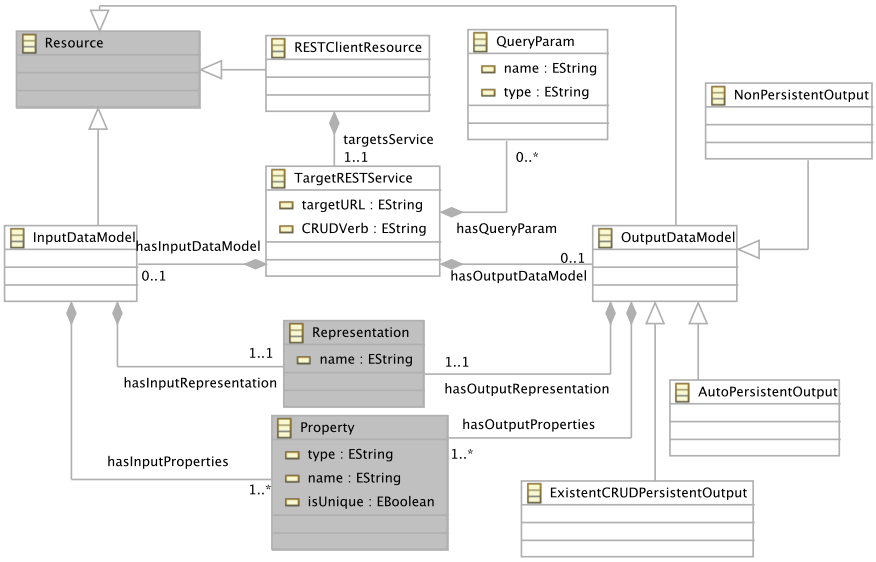

Fig. 17 3rd Party Services Wrapper CIM Meta-Model.

The core concept of this part is the *RESTClientResource*. It is a specialization of the Resource concept with exposed web API. Each such RESTClientResource has a *TargetRESTService*, which embeds the URL of the 3rd party service and the expected CRUD verb that has to be used. Should there exist query parameters, they are modeled with the *QueryParam* concept. The *InputDataModel* and *OutputDataModel* concepts model the input and output data of the target RESTful service. Both of them are specializations of the Resource concept and as such, they have a media format Representation and some Properties that model the data of the input or output model. Finally, there are three specializations of the OutputDataModel. The first one, the *NonPersistentOutput* models output data that will not be persisted in the local database but will only be sent back to the client. The *AutoPersistentOutput* uses the OutputDataModel meta-data to automatically create a fully persistent resource with a web API, whilst the third specialization, the *ExistentCRUDPersistentOutput* uses an existing Resource of the system as an OutputDataModel. Table 14 provides an example of an instantiated model.

Table 14 3rd Party Interoperation Modelling example

CIM Meta-Model Concept	Value
RESTClientResource	LocalWeather
TargetRESTService - targetURL	http://www.example.com/LocalWeather/REST
TargetRESTService - CRUDVerb	READ
QueryParam	Country
QueryParam	City
InputDataModel	WeatherDate
InputDataModel - Property	Year
InputDataModel - Property	Month
InputDataModel - Property	Day
InputDataModel - Representation	application/JSON
NonPersistentOutput	Forecast
NonPersistentOutput - Property	Temperature
NonPersistentOutput - Property	Humidity
NonPersistentOutput - Representation	application/XML

4.3 Transformation Chain: From YAML input to code

Having parsed the static and dynamic view of the service all information is stored to the aggregated ontology (see Subsection 3.3). As already said, this information is then used to produce the service's YAML file. The next step is to instantiate the envisioned system CIM model from the YAML input. Figure 18 illustrates the straightforward mapping between the YAML input file and the CIM meta-model concepts.

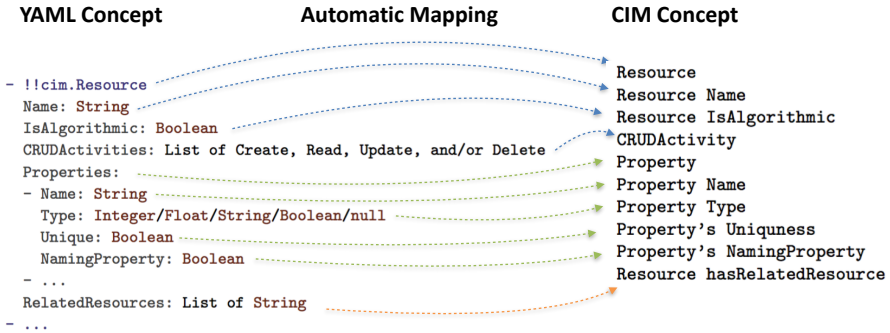

Fig. 18 YAML to CIM Metamodel concept mapping.

This mapping creates a semi-complete CIM meta-model that provides the developer with a head start that decreases the necessary manual modeling effort. Moreover, the aforementioned OCL validation mechanisms support the whole modeling process by indicating errors in the envisioned system CIM model, thus guiding the developer towards a valid CIM instance model of his/her envisioned service. This will be further illustrated in the case study that follows in the next section.

4.4 M2M ATL Transformations

After the developer has provided the missing information to the CIM meta-model of the envisioned RESTful service, the common model-to-model MDA transformation chain takes place. This chain comprises two ATL[19] transformations, one for the CIM-to-PIM transformation and another for the PIM-to-PSM one. Both of them follow the transformation scheme shown in Figure 19.

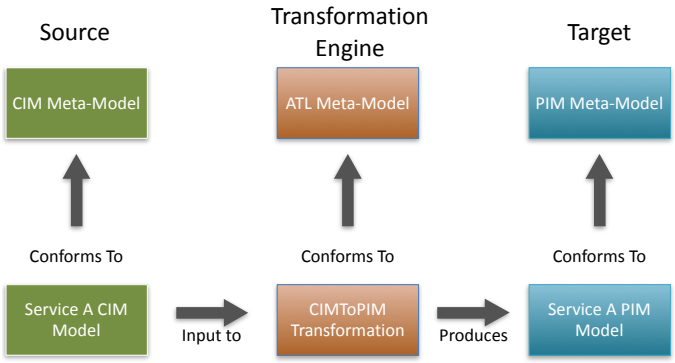

Fig. 19 Overview of the CIM-to-PIM Model-to-Model ATL transformation.

The CIM-to-PIM transformation needs as input the envisioned system's CIM model, which conforms to the presented CIM meta-model and by applying a chain of ATL rules to CIM concepts, it produces the envisioned system's PIM model. Figure 20 illustrates such an ATL rule which transforms a CIM Property to its PIM counterpart.

```
unique lazy rule  addRModelProperties{
  from
    CIMProperty: CIM!Property
  to
        PIMRModelProperty: PIM!PIMComponentProperty(
          name <- CIMProperty.name,
          type <- CIMProperty.type,
          isUnique <- CIMProperty.isUnique,
          isNamingProperty <- CIMProperty.isNamingProperty,
          isPrimaryIdentifier <- false,
          isMappedToRDBMSColumn <- thisModule.
                                   createRDBMSTableColumn(CIMProperty)
    )
}
```

Fig. 20 ATL Transformation Rule of the CIM Property concept

In this case the name, type, multiplicity and naming attributes remain unchanged. Moreover, the transformation rule marks these properties as non-

[19] https://eclipse.org/atl/

identifiers, since they are not used as database primary keys and maps them to the corresponding database table column by calling the *createRDBMSTableColumn* rule. The full definition of these transformations and the PIM/PSM metamodels is omitted due to space limitations. Both can be retrieved online[20].

4.5 M2T Acceleo Transformation

The final step the MDE engine takes is the PSM-to-Code Model-to-Text transformation. This is usually done by employing some sort of code templates. The proposed engine employs the Acceleo[21] template language. In total there are 33 template files, each of which uses several PSM concepts to produce one java code file or a web service configuration/maven file. Figure 21 illustrates the simplest template used to produce the *web.xml* file of the envisioned system.

```
[module webXML('../PSMMetamodel.ecore')]
[template public webXMLConfigurationFile(aPSM : PSM)]
[file (aPSM.name + '/src/main/webapp/WEB-INF/web.xml', false, 'UTF-8')]
<web-app id="WebApp_ID" version="2.4"
 xmlns="http://java.sun.com/xml/ns/j2ee"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
 http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
 <display-name> [aPSM.name/] </display-name>
 <servlet>
    <servlet-name>jersey</servlet-name>
    <servlet-class>com.sun.jersey.spi.container.servlet.
              ServletContainer</servlet-class>
        <init-param>
            <param-name>javax.ws.rs.Application</param-name>
            <param-value> eu.fp7.scase.[aPSM.name.toLowerCase()/].
                                utilities.JAXRSPublisher</param-value>
        </init-param>
 </servlet>
 <servlet-mapping>
    <servlet-name>jersey</servlet-name>
    <url-pattern>/*</url-pattern>
 </servlet-mapping>
</web-app>
[/file]
[/template]
```

Fig. 21 Acceleo template that produces the envisioned system's web.xml file.

The only PSM concept needed to produce the output is the name of the service (highlighted with blue). The whole package of the acceleo templates can be found at the MDE engine's online source code repository[22].

[20] http://s-case.github.io/publications/ase2015/S-CASE_D2.2.pdf

[21] https://eclipse.org/acceleo/

[22] https://github.com/s-case/mde/tree/master/eu.scasefp7.eclipse.mde.m2t/src/LayeredPSMToText/files

The output of this Model-to-Text transformation is a folder structure that contains all the java source code of the envisioned system alongside the *pom.xml* Maven[23] build file, a *web.xml* service configuration file and the persistence storage with Hibernate[24] and Lucene[25] configuration files *hibernate.cfg.xml* and *persistence.xml* respectively. Figure 22 illustrates the MDE engine schema output. All these together form a ready to compile bundle.

```
-[serviceName]
        |
        pom.xml
        src/main/java/eu/fp7/scase/[serviceName]
                |                       |
                |                  [utilities]
                |                  [resourceName] (one per resource)
                |                     |
                |                      *.java (resource's java files)
               webapp
                |
                WEB-INF
                  |
                  web.xml
                  classes
                           |
                           hibernate.cfg.xml
                           META-INF
                                   |
                                   persistence.xml
```

Fig. 22 MDE engine output folder structure outline

However, in most cases the developer will have to add manual code for functionality that is not automatable by this engine. The amount of manual code needed varies, of course, and depends on how well the service is modeled using the CIM meta-model domain specific language. For example envisioned services that are pure data-handling, with Basic Authentication, alongside simple database keyword-searching and interoperation with existing 3rd party services are fully automatable. Moreover, the aforementioned functionality is implemented using technologies that lie in the concrete technology set of the PSM. While currently the PSM set of the proposed engine is being expanded with another *.NET*[26] and *Doxygen*[27] PSM, the primal PSM used comprises:

- Java[28] language as programming language
- JAXB[29] framework for XML to java classes binding

[23] https://maven.apache.org
[24] http://hibernate.org
[25] https://lucene.apache.org
[26] http://www.microsoft.com/net
[27] http://www.stack.nl/~dimitri/doxygen/
[28] http://www.oracle.com/technetwork/java/index.html
[29] https://jaxb.java.net

– JAX-RS[30] framework to expose resource web API and Jersey[31] as its implementation
– Hibernate framework for Object-Relational-Mapping (ORM)
– Lucene

On the other hand, any other functionality is wrapped in a Java template-file placeholder, that is compilable and executable, but without any functionality within it. The developer has to fill in these "empty" template files in order to create a service that is fully compliant with the input user requirements. In this task, the developer may reuse any common, automatically produced, functionality that lies within the utilities folder. This is especially helpful for local database data-handling.

As a final note, the automation achieved by our mechanism depends on the desired functionality that the envisioned service must embed. All services that model a data structure that needs to be exposed through a web API, including authentication capabilities, keyword searching and/or interoperation with existing 3rd party services, are expected to be fully automatable. In cases where common web service functionality is desired alongside some custom algorithmic functionality, our mechanism produces the largest part of the envisioned service and creates RESTful wrapper templates for any manual code that the developer has to add. In the extreme case of web services that do not embed any data structures and only comprise custom functionality that is also not available as 3rd party service, our mechanism is expected to create a set of RESTful templates to which manual code has to be added in order to be fully functional. However, in any of the aforementioned cases, the outcome is a compilable and executable service that is ready for deployment. Table 15 illustrates these cases.

Table 15 Outcome of the proposed mechanism in various functionality scenarios

	REST Model	Basic Auth	Keyword Searching	3rd party interoperation	Custom Functionality
Automated Work	REST API and Database	Auto login mechanism	Auto Searching mechanism	Wrapper that interoperates with 3rd party	Template REST Wrappers
Manual Work	-	-	-	-	Fill in custom code

5 Illustrating the Functionality of the Mechanism

The modules of our system include two tools for entering multimodal requirements, the Requirements Editor and the Storyboard Creator, which also instantiate the ontologies to construct the first model representation, as well as

[30] https://jax-rs-spec.java.net
[31] https://jersey.java.net

an MDE engine that applies model-to-model and model-to-text transformations to finally produce the envisioned service's executable source code. All tools, including installation and usage instructions, can be found at:

`http://s-case.github.io/`

In the following subsections, we assess the effectiveness of our methodology for constructing fully functional RESTful web services using multimodal input from software requirements. At first, we provide an overview of the action flow a developer has to follow in order to produce a web service. Then, we illustrate the application of our methodology using an actual case study. Finally, we provide some metrics of using our mechanism either in our case examples or in user studies that span various application domains.

5.1 Flow of Actions

The flow of actions of our system is shown in Figure 23.

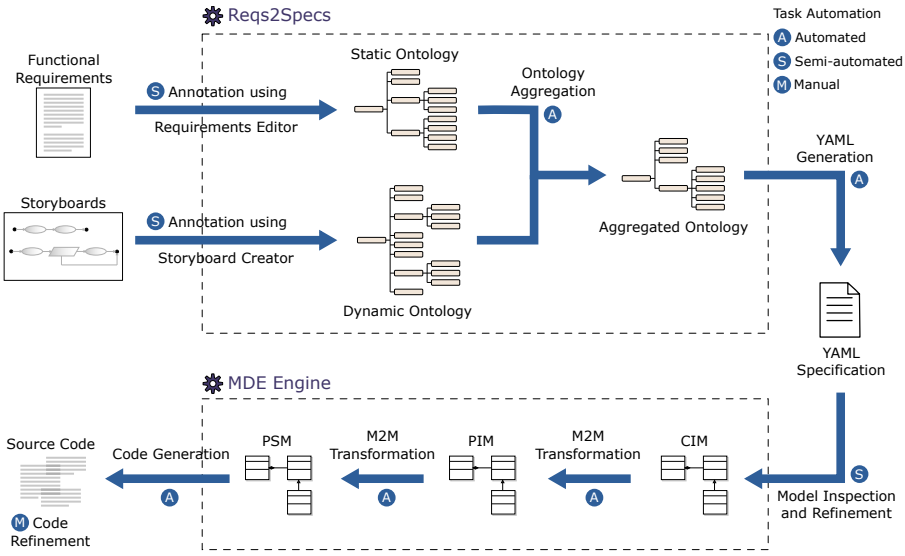

Fig. 23 Overview of the Action Flow of our System

At first, the developer uses the Requirements Editor and the Storyboard Creator plugins in order to construct the functional requirements and the storyboards that describe the envisioned service. These artefacts are syntactically and semantically annotated and the annotations are subsequently refined by the user. After that, the static ontology and the dynamic ontology of the system are automatically instantiated and aggregated to produce an instance of the REST-compliant aggregated ontology. The YAML file is automatically generated from that instance and provided to the developer, so that he/she

may inspect it and make any refinements using the wizard of the MDE plugin. The MDE Engine performs automated M2M transformations from CIM to PIM and from PIM to PSM, and finally produces the source code of the envisioned service given the PSM. The generated service is fully functional, including also a database for its resources as well as an API for its endpoints. The source code for all resources of the service is generated automatically, while the developer may only need to refine the code for algorithmic resources.

5.2 RESTMarks Case Study

In this subsection, we provide a case study for the example project *Restmarks*. Consider Restmarks as a service that allows users to store and retrieve online their bookmarks, share them with other users and search for bookmarks by using tags. One could think of it as a social service for bookmarks. In the following paragraphs, we illustrate how one can use our system to create such a product easily, while at the same time ensure that the produced service is fully functional and traceable. All elements required to reproduce our case study are provided at `http://s-case.github.io/publications/ase2015`, including requirements, storyboards, MDE models, and the produced code.

The first step of the process includes entering and annotating functional requirements. An illustrative part of the requirements of Restmarks, annotated and refined using the Requirements Editor, are depicted in Figure 24.

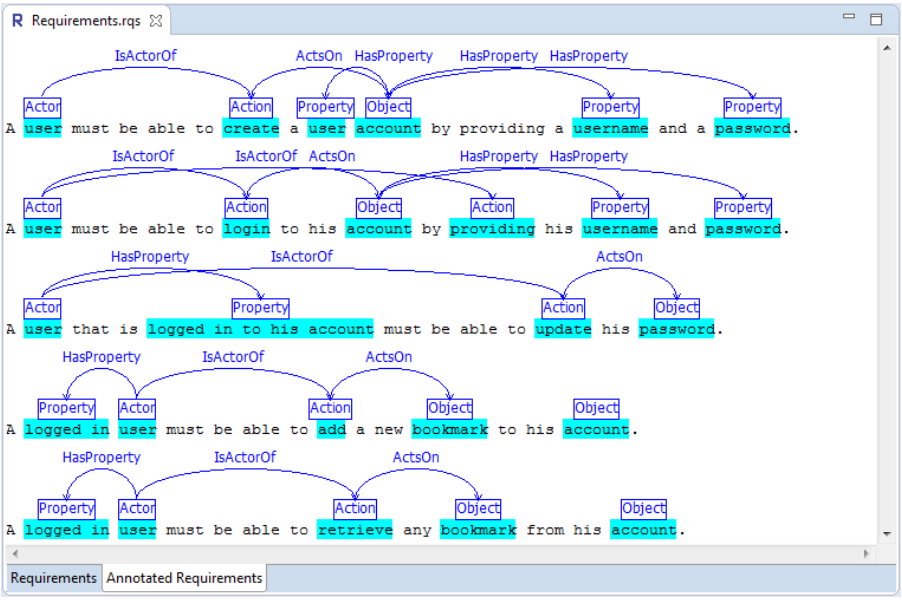

Fig. 24 Excerpt of the Annotated Functional Requirements of Project Restmarks

Upon adding the requirements, the user also has to enter information about the dynamic view of the system. In this example, dynamic system represen-

tation is given in the form of storyboards. Let us assume Restmarks has the following dynamic scenarios:

1. Add Bookmark: The user adds a bookmark to his/her collection and optionally adds a tag to the newly added bookmark.
2. Create Account: The user creates a new account.
3. Delete Bookmark: The user deletes one of his/her bookmarks.
4. Login to Account: The user logs in to his/her account.
5. Search Bookmark by Tag System Wide: The user searches for bookmarks by giving the name of a tag. The search involves all public bookmarks.
6. Search Bookmark by Tag User Wide: The user searches for bookmarks by giving the name of a tag. The search a user's public and private bookmarks.
7. Show Bookmark: The system shows a specific bookmark to the user.
8. Update Bookmark: The user updates the information on one of his/her bookmarks.

The storyboards of the above scenarios are created using the Storyboard Creator. An example storyboard diagram using the tool is shown in Figure 25. The storyboard of this Figure refers to the first scenario of adding a new bookmark. The rest of the requirements and storyboards of the project are omitted due to space limitations; the reader is referred to `http://s-case.github.io/publications/ase2015` for the storyboards of the project.

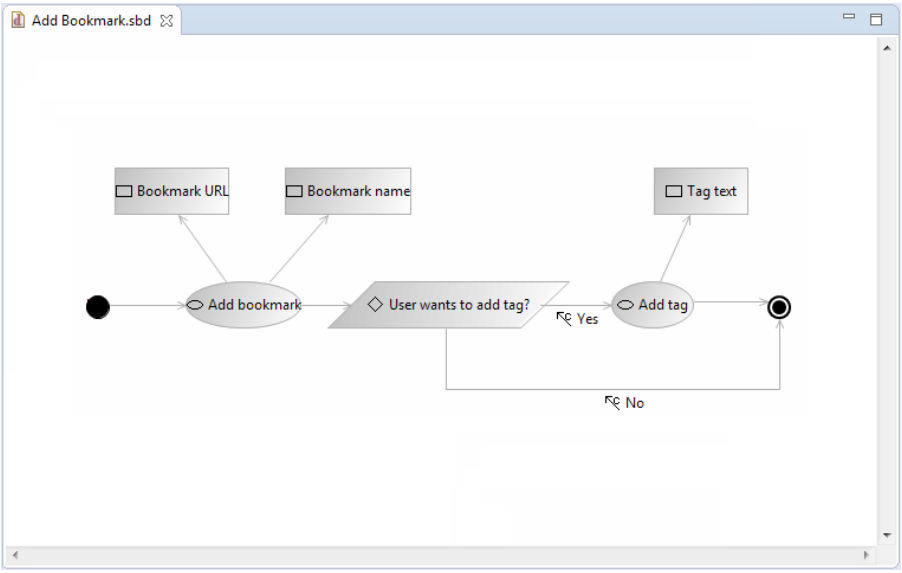

Fig. 25 Storyboard Diagram "Add bookmark" of Project Restmarks

Upon having composed the annotated requirements and the storyboards, the next step involves creating the static and dynamic ontologies. The two ontologies are combined to provide the instances of the aggregated ontology.

An illustrative instantiation of the OWL classes `Resource`, `Property`, and `Activity` of the aggregated ontology is shown in Table 16.

Table 16 Instantiated Classes `Resource`, `Property`, and `Activity` for Project Restmarks

OWL Class	OWL Instances
`Resource`	`bookmark`, `tag`, `account`, `password`
`Property`	`username`, `password`, `private`, `public`, `user`
`Activity`	`search_bookmark`, `add_tag`, `add_bookmark`, `delete_bookmark`, `update_tag`, `update_password`, `login_account`, `retrieve_bookmark`, `update_bookmark`, `get_bookmark`, `delete_tag`, `mark_bookmark`, `create_account`

Finally, the YAML representation that describes the project is exported from the aggregated ontology. For Restmarks, the YAML file is shown in Figure 26, tweaked using the CIM wizard to produce a more complete CIM.

```
- !!cim.Resource
  Name: account
  IsAlgorithmic: false
  CRUDActivities: [Create, Read, Update, Delete]
  Properties: [username, password]
  RelatedResources: [bookmark]
- !!cim.Resource
  Name: tagSearch
  IsAlgorithmic: true
  CRUDActivities: []
  Properties: []
  RelatedResources: []
- !!cim.Resource
  Name: tag
  IsAlgorithmic: false
  CRUDActivities: [Create, Read, Update, Delete]
  Properties: [name, description]
  RelatedResources: [tagSearch]
- !!cim.Resource
  Name: bookmark
  IsAlgorithmic: false
  CRUDActivities: [Create, Read, Update, Delete]
  Properties: [url, scope]
  RelatedResources: [tag]
```

Fig. 26 Example YAML File for Project Restmarks

Once the YAML representation of the envisioned system is produced, the MDE takes action by giving the developer the capability to refine the parsed software artefacts and/or define new functionality. The MDE engine module provides a set of self-explanatory screens, each of which modifies concepts of the CIM (principal REST part, Basic Authentication, etc.). Figure 27 depicts the principal REST CIM meta-model part screen and some of the re-

finements the developer has done in order to produce a valid model of the system, e.g. adding new resources (bookmarkShare), specifying datatypes, input/output representation media formats and inter-resource relationships.

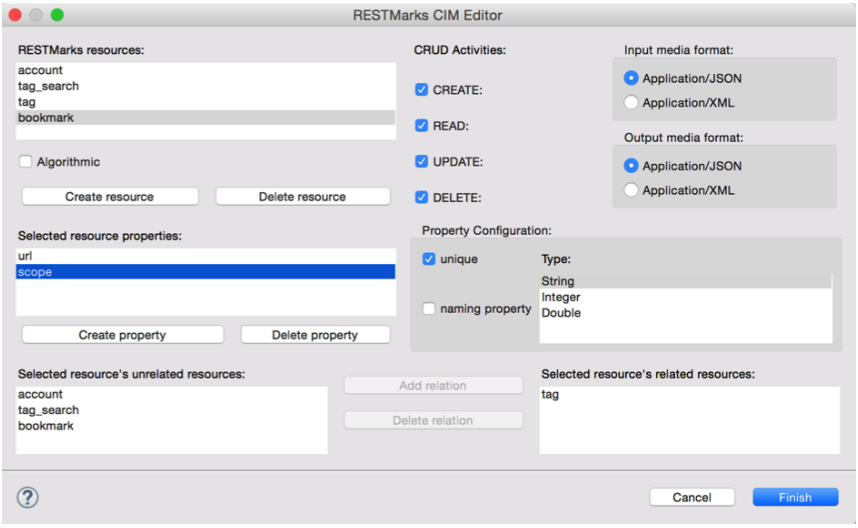

Fig. 27 REST CIM Meta-model UI.

Figures 28 and 29 present the Authentication UI's through which the developer can choose his envisioned system's Authentication Model and Tokens and then fully define the Authentication Mode for the service's CRUDActivities.

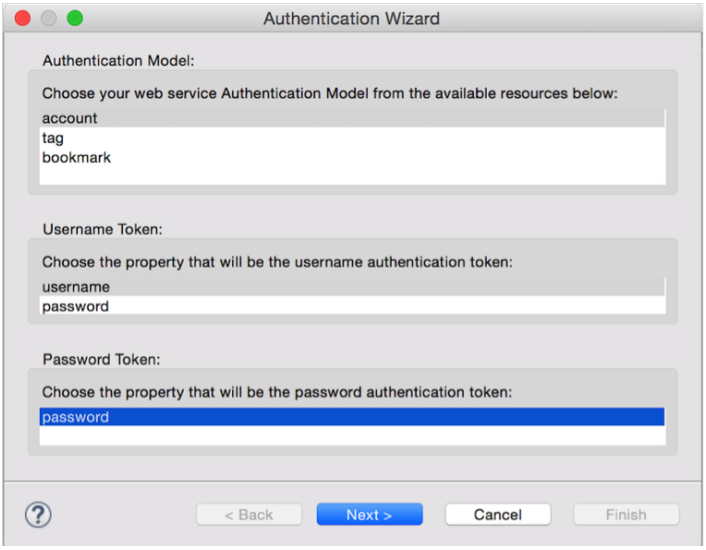

Fig. 28 Selecting Authentication Model with the Basic Authentication UI.

Fig. 29 Defining the desired authentication of each CRUDActivity

Figure 30 depicts the UI for offering search capabilities. In this case, the developer specialized the TagSearch resource as a Search Resource, which searches the *description* property of the *Tag* resource.

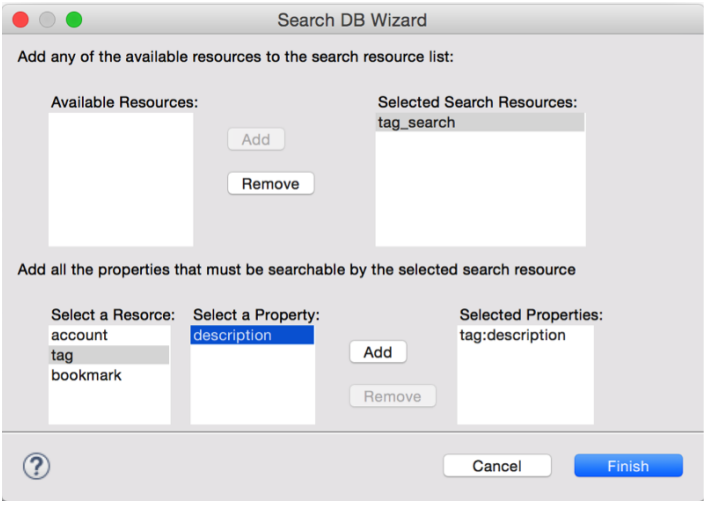

Fig. 30 Defining the TagSearch Search Resource.

Finally, Figure 31 depicts the UI for defining interoperation with 3rd party services. In this case, the developer specialized the bookmarkShare Resource to be able to interoperate with Facebook and share a bookmark.

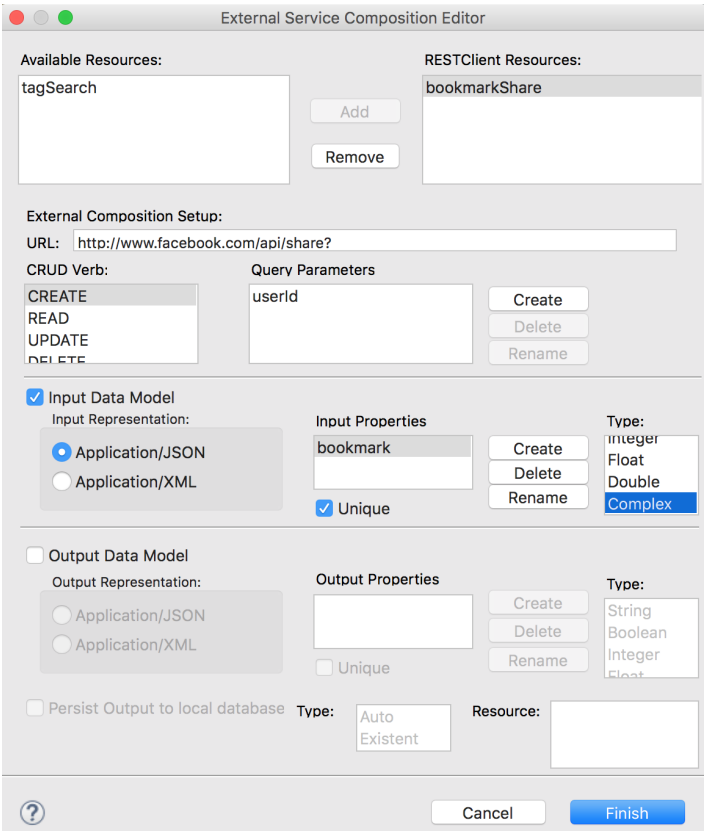

Fig. 31 Defining the bookmarkShare Resource.

Once the developer steps through all the aforementioned screens, the MDE engine Module lower CASE produces the services code. For this illustrative example, it created 38 Java files that sum up to 3843 lines of code plus the maven pom.xml file to build the service.

5.3 Evaluation

Apart from Restmarks which serves as a case study presented in the previous subsection, we further evaluate our methodology and specifically assess the effort reduction using our mechanism in four more projects. One of them is project RESTReviews, which we created as a sample online shop that supports buying products, writing reviews, etc. For the three other projects, we conducted a study based on the methodology that follows.

Study Goal The goal of our study is to assess whether our mechanism results in developer productivity gain in development teams within a corporate environment.

Study Design We provided our mechanism to three distinct teams of professional developers, each designing and producing its own domain application in its corporate environment, in the context of their work in the S-CASE project. These three user studies span various application domains: one of them is an Internet of Things (IoT) application, one is a mini Social Network and the last is an Internet as a Service (IaaS) application. More information on the applications can be retrieved online[32].

Initially, we trained each development team using webinars (2 hours), specialized topic tutorials (2 hours) and interactive coaching sessions (2 hours) for a total of 6 hours of training per team. Then each of the teams developed its application using our mechanism (Phase A) and then without it (Phase B), using their preferred methodology. Our goal was to measure the effort reduction. All the teams used the Goal Question Metric method (Basili and Weiss, 1984) to define a methodology for evaluating our mechanism and then report the measured software development process improvements.

Study Results Table 17 summarizes the evaluation results for the two sample projects and the three user studies.

Table 17 Effort reduction using the presented mechanism

Metric	RESTMarks	RESTReviews	IoT app	Social Network app	IaaS app
Assessment Type	Authors	Authors	User Study	User Study	User Study
Number of Requirements/ Storyboards	21	10	13	14	12
Java LoC generated	3843	2888	7430	8115	6466
MU Coefficient (hours)	N/A	N/A	2	2.4	1.4
MD Coefficient (hours)	N/A	N/A	10.5	10.8	9.3
Effort Reduction (%)	N/A	N/A	23.81%	22.2%	20.4%
Effort Reduction without Learning Curve (%)	N/A	N/A	80.9%	77.8%	85%

The first row provides the type of assessment performed, namely either assessment as case studies by the authors or assessment in user studies. The second row of the table provides information on the number of software requirements that the developers had to add to our mechanism in the form of

[32] http://s-case.github.io/publications/ase2015/S-CASE_D6.2.pdf

functional requirements and/or storyboards. The third row presents the automatically produced lines of code (LoC) in the Java language. The fourth row presents the MU coefficient, which stands for the time needed to generate the desired service using our mechanism, while the fifth row presents the MD coefficient that stands for the Manual Development effort required to provide the same functionality that has been automatically generated with our mechanism, e.g. creating the interface of the service, setting up its database, embedding searching capabilities and building interoperations with external services, testing and debugging respectively. The last two rows present the effort reduction. In the second-to-last row, the effort reduction value for the provided case projects is calculated by taking into account also the learning curve of our mechanism. In the last row, the effort reduction is also provided without the learning curve, to illustrate the productivity gain once a developer has learned to use our mechanism.

The Effort Reduction metric (ER) is computed as a percentage using the following formula:

$$ER = \frac{(LC + MU) - MD}{MD} \cdot 100\% \tag{1}$$

where the LC coefficient stands for the Learning Curve estimation in hours, which includes the aforementioned activities performed in order to train the developer teams. For example, for the team developing the IoT app, the reported effort has been 10.5 hours. Hence, using equation (1) and given that the values for MU and MD are 2 hours and 10.5 hours respectively, the effort reduction including the LC coefficient (with value equal to 6 hours) is 23.81%, whilst excluding it (LC set to 0) concludes that the effort reduction is 80.9%.

Threats to validity A threat to validity of the aforementioned study results is that the three developer teams have self-reported the results for the productivity improvement. However, we consider the self-reporting threat negligible since the teams established and applied a solid methodology to measure the productivity gain, based on the Goal Question Metric method. Another possible threat is that the teams used their preferred methodology for the manual round of building the system (Phase B). However, since each developer team used technologies and tools that they are highly familiar with, they peaked their productivity during this phase, thus minimizing this threat. Finally, the ordering of the development experiment, first using our mechanism (Phase A) and then without it (Phase B), does not favor our methodology either; on the contrary, it might favor the manual round, as the development teams had to re-solve an already familiar development problem to them.

As a final note, in comparison to other popular frameworks such as RAML[33], our mechanism enhances developer productivity in two ways. At first, our mechanism supports the developer through the typical Requirements to Design phase of Software Engineering, since it receives input in the form of software requirements and automates the production of the design model. By

[33] `http://raml.org`

contrast, in other popular frameworks, the developer has to manually create an initial model (e.g. RAML model) by thoroughly examining the envisioned system's requirements, which requires a significant amount of effort. Secondly, the outcome of such Web-API automators is usually a scaffolding of resources, rather than fully deployable code that may also embed common web service functionality, as is the case with our mechanism.

6 Conclusion

Lately, the problem of automating the software development process has attracted the attention of several researchers. Although MDE has been proven effective for constructing prototypes of REST-compliant web services, current tools do not fully cover the features of the RESTful paradigm. In this paper, we have designed a methodology that allows developers to model their envisioned system using software requirements from multimodal formats, and an MDE engine that applies model-to-model transformations in order to produce a ready-to-deploy RESTful web service.

Our system receives input in the form of functional requirements and action flows in the form of storyboards, thus covering both the structural and the behavioral views of the envisioned system. The use of NLP and semantics facilitates the extraction of specifications from these requirements, while the designed ontologies produce a traceable model that is highly compliant to the RESTful paradigm. The produced model (YAML file) comprises all the required elements of the service, including resources, CRUD actions and properties, and support for hypermedia links.

Additionally, the proposed MDE lower CASE, in contrast with most other approaches, goes beyond REST modeling by also embedding common functionality that is needed in many web services. It supports Basic Authentication functionality, popular database keyword-searching and interoperability with existing 3rd party RESTful services. Moreover, it models the necessary, compilable and executable, placeholder code in order to guide the developer in his manual programming effort for the parts of his envisioned system that cannot be automated.

Summarizing the contributions of this work, we have created a set of tools for developers to extract specifications from functional requirements and storyboards and transform these specifications to the source code of the envisioned system. Upon assessing our tools and our methodology in a case study for project Restmarks (see Section 5), we may conclude that our system provides an intuitive traceable solution for semi-automatically producing a web service prototype from software requirements. Further evaluating our methodology in user studies with regard to reducing development time, it is indicated that our system can greatly facilitate the development of RESTful web services.

Future work on our methodology may lie in several directions. At first, concerning the Reqs2Specs module, the continuous improvement of the tools by receiving feedback from users is in our immediate plans. Concerning the MDE

lower CASE, further evolution includes exploration of developer's manual code handling after the first MDE engine run, automation of non-functional aspects as well as the automated production of a matching web client. Finally, future research includes further assessing our methodology in industrial settings and evaluating its effectiveness both for the quality of the produced service and for reducing development time.

Acknowledgements Parts of this work have been supported by the FP7 Collaborative Project S-CASE (Grant Agreement No 610717), funded by the European Commission. S-CASE aims to provide a cloud-based realm of services and tools for software developers to enable rapid software prototyping based on user requirements and system models, provided in multimodal formats. S-CASE provides automated solutions for (a) the extraction of system specifications and architecture, (b) the transformation of these specifications (models) to the source code of RESTful web services, and (c) the discovery and synthesis of composite workflows of software artefacts from distributed open source and proprietary resources in order to fulfill the system requirements. More info on S-CASE can be found at `http://s-case.github.io/`.

References

Abbott RJ (1983) Program design by informal english descriptions. Commun ACM 26(11):882–894

Basili VR, Weiss DM (1984) A methodology for collecting valid software engineering data. IEEE Transactions on Software Engineering SE-10(6):728–738, DOI 10.1109/TSE.1984.5010301

Booch G (1986) Object-oriented development. IEEE Trans Softw Eng 12(1):211–221

Castañeda V, Ballejos L, Caliusco ML, Galli MR (2010) The use of ontologies in requirements engineering. Global Journal of Researches In Engineering 10(6)

Dermeval D, Vilela J, Bittencourt I, Castro J, Isotani S, Brito P, Silva A (2015) Applications of ontologies in requirements engineering: a systematic review of the literature. Requirements Engineering pp 1–33

Ed-Douibi H, Izquierdo JLC, Gómez A, Tisi M, Cabot J (2015) EMF-REST: generation of restful apis from models. CoRR abs/1504.03498

Fielding RT (2000) Architectural styles and the design of network-based software architectures. PhD thesis, University of California, Irvine

Happel HJ, Seedorf S (2006) Applications of ontologies in software engineering. In: Proceedings of the 2nd International Workshop on Semantic Web Enabled Software Engineering (SWESE 2006), held at the 5th International Semantic Web Conference (ISWC 2006), pp 5–9

Harmain HM, Gaizauskas R (2003) Cm-builder: A natural language-based case tool for object-oriented analysis. Automated Software Engg 10(2):157–181

Hernández AG, García MNM (2010) A formal definition of restful semantic web services. In: Proceedings of the First International Workshop on RESTful Design, ACM, New York, NY, USA, WS-REST '10, pp 39–45

Hutchinson J, Whittle J, Rouncefield M, Kristoffersen S (2011) Empirical assessment of mde in industry. In: Proceedings of the 33rd International Conference on Software Engineering, ACM, New York, NY, USA, ICSE '11, pp 471–480

Kaindl H, Smialek M, Svetinovic D, Ambroziewicz A, Bojarski J, Nowakowski W, Straszak T, Schwarz H, Bildhauer D, Brogan JP, Mukasa KS, Wolter K, Krebs T (2007) Requirements specification language definition: Defining the redseeds languages, deliverable d2.4.1. Public deliverable, ReDSeeDS (Requirements Driven Software Development System) Project

Liebel G, Marko N, Tichy M, Leitner A, Hansson J (2014) Assessing the state-of-practice of model-based engineering in the embedded systems domain. In: Dingel J, Schulte W, Ramos I, Abraho S, Insfran E (eds) Model-Driven Engineering Languages and Systems, Lecture Notes in Computer Science, vol 8767, Springer International Publishing, pp 166–182

Mich L (1996) Nl-oops: From natural language to object oriented requirements using the natural language processing system lolita. Nat Lang Eng 2(2):161–187

Moody D (2009) The physics of notations: Toward a scientific basis for constructing visual notations in software engineering. Software Engineering, IEEE Transactions on 35(6):756–779, DOI 10.1109/TSE.2009.67

Mylopoulos J, Castro J, Kolp M (2000) Tropos: A framework for requirements-driven software development. In: Information Systems Engineering: State of the Art and Research Themes, Springer-Verlag, pp 261–273

Newman S (2015) Building Microservices. O'Reilly Media Inc

North D (2003) Jbehave: A framework for behaviour driven development. URL `http://jbehave.org/`

Parastatidis S, Webber J, Silveira G, Robinson IS (2010) The role of hypermedia in distributed system development. In: Proceedings of the First International Workshop on RESTful Design, ACM, New York, NY, USA, WS-REST '10, pp 16–22

Porres I, Rauf I (2011) Modeling behavioral restful web service interfaces in uml. In: Proceedings of the 2011 ACM Symposium on Applied Computing, ACM, New York, NY, USA, SAC '11, pp 1598–1605

Rauf I, Ruokonen A, Systa T, Porres I (2010) Modeling a composite restful web service with uml. In: Proceedings of the Fourth European Conference on Software Architecture: Companion Volume, ACM, New York, NY, USA, ECSA '10, pp 253–260

Richardson L, Ruby S (2007) Restful Web Services, 1st edn. O'Reilly

Roth M, Diamantopoulos T, Klein E, Symeonidis A (2014) Software requirements: A new domain for semantic parsers. In: Proceedings of the ACL 2014 Workshop on Semantic Parsing, Association for Computational Linguistics, Baltimore, MD, pp 50–54

Roth M, Diamantopoulos T, Klein E, Symeonidis A (2015) Software requirements as an application domain for natural language processing. Language Resources and Evaluation Under review

Saeki M, Horai H, Enomoto H (1989) Software development process from natural language specification. In: Proceedings of the 11th International Conference on Software Engineering, ACM, New York, NY, USA, ICSE '89, pp 64–73

Schreier S (2011) Modeling restful applications. In: Proceedings of the Second International Workshop on RESTful Design, ACM, New York, NY, USA, WS-REST '11, pp 15–21

Siegemund K, Thomas EJ, Zhao Y, Pan J, Assmann U (2011) Towards ontology-driven requirements engineering. In: Workshop semantic web enabled software engineering at 10th international semantic web conference (ISWC), Bonn

Smialek M (2012) Facilitating transition from requirements to code with the redseeds tool. In: Proceedings of the 2012 IEEE 20th International Requirements Engineering Conference (RE), IEEE Computer Society, Washington, DC, USA, RE '12, pp 321–322

Störrle H (2014) Model-Driven Engineering Languages and Systems: 17th International Conference, MODELS 2014, Valencia, Spain, September 28 – October 3, 2014. Proceedings, Springer International Publishing, Cham, chap On the Impact of Layout Quality to Understanding UML Diagrams: Size Matters, pp 518–534. DOI 10.1007/978-3-319-11653-2_32, URL http://dx.doi.org/10.1007/978-3-319-11653-2_32

Sweller J (1994) Cognitive load theory, learning difficulty, and instructional design. Learning and Instruction 4(4):295 – 312, DOI http://dx.doi.org/10.1016/0959-4752(94)90003-5, URL http://www.sciencedirect.com/science/article/pii/0959475294900035

Tavares NAC, Vale S (2013) A model driven approach for the development of semantic restful web services. In: Proceedings of International Conference on Information Integration and Web-based Applications & Services, ACM, New York, NY, USA, IIWAS '13, pp 290:290–290:299

Wynne M, Hellesoy A (2012) The Cucumber Book: Behaviour-Driven Development for Testers and Developers. Pragmatic Bookshelf

Yu ESK (1995) Modelling strategic relationships for process reengineering. PhD thesis, University of Toronto, Toronto, Ont., Canada, Canada

Zhao H, Doshi P (2009) Towards automated restful web service composition. In: Web Services, 2009. ICWS 2009. IEEE International Conference on, pp 189–196

Zuzak I, Budiselic I, Delac G (2011) Formal modeling of restful systems using finite-state machines. In: Auer S, Daz O, Papadopoulos G (eds) Web Engineering, Lecture Notes in Computer Science, vol 6757, Springer Berlin Heidelberg, pp 346–360