Seventh Framework Programme

CallFP7-ICT-2013-10

Project Acronym:	**S-CASE**
Grant Agreement N^{o}:	**610717**
Project Type:	**COLLABORATIVE PROJECT**
Project Full Title:	**Scaffolding Scalable Software Services**

D3.2.2 Module for extracting software artefacts from storyboards

Nature:	**R**
Dissemination Level:	**PU**
Version #:	**1.0**
Date:	**30 January 2015**
WP number and Title:	**WP3 Multimodal Information Processing**
Deliverable Leader:	**AUTH**
Author(s):	**Themistoklis Diamantopoulos (AUTH)**
Revision:	**Davide Tossi (INS), Ciro Formisano (ENG), Andreas Symeonidis (AUTH)**
Status:	**Submitted (Draft, Peer-Reviewed, Submitted, Approved)**

Document History

Version[1]	Issue Date	Status[2]	Content and changes
0.1	10 December 2014	Draft	TOC
0.2	22 December 2014	Draft	Added Section 1 and subsection 2.1
0.3	29 December 2014	Draft	Added Section 3
0.4	3 January 2015	Draft	Added subsection 2.2
0.5	5 January 2015	Draft	Added Section 4
0.6	9 January 2015	Draft	Added Sections 5 and 6, minor corrections
0.7	22 January 2015	Peer-Reviewed	Minor corrections noted by reviews
0.8	23 January 2015	Peer-Reviewed	Added subsection 2.3
0.9	26 January 2015	Peer-Reviewed	Added subsection 5.4
1.0	30 January 2015	Submitted	Final Proof-reading

Peer Review History[3]

Version	Peer Review Date	Reviewed By
0.6	16 January 2015	Ciro Formisano (ENG)
0.6	22 January 2015	Davide Tosi (INS)
1.0	30 January 2015	Andreas Symeonidis (AUTH)

[1]Please use a new number for each new version of the deliverable. Use “0.#” for Draft and Peer-Reviewed. “x.#” for Submitted and Approved”, where x>=1.Add the date when this version was issued and list the items that have been added or changed.

[2]A deliverable can be in one of these stages: Draft, Peer-Reviewed, Submitted and Approved.

[3]Only for deliverables that have to be peer-reviewed

Table of contents

Abbreviations and Acronyms

API	Application Programming Interface
CIM	Computationally Independent Model
CRUD	Create, Read, Update, Delete
EMF	Eclipse Modeling Framework
GEF	Graphical Editing Framework
GMF	Graphical Modeling Framework
HTTP	HyperText Transfer Protocol
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
MDE	Model-Driven Engineering
OWL	Web Ontology Language
RAML	RESTful API Modeling Language
REST	REpresentational State Transfer
RDF	Resource Description Framework
UML	Unified Modeling Language
URI	Uniform Resource Identifier
XML	eXtensible Markup Language
YAML	YAML Ain't Markup Language

Executive Summary

The module for extracting software artefacts from storyboards converts dynamic scenarios expressed in the form of storyboards into artefacts that map to the S-CASE ontology. This is achieved by using a storyboard diagram editor that allows the developer to create and edit his/her scenarios and a dynamic ontology to store the dynamic aspects of the system.

In the context of the S-CASE architecture, this module provides an initial analysis of the dynamic view of software projects, which includes storyboards and dynamic-view UML diagrams (e.g. activity diagrams). This information is used in conjunction with information on the static view of software projects (text requirements, use case diagrams, etc.), as covered in task T3.1, to provide a unified view of a software project. Additionally, this view will populate the S-CASE registry, and will serve as the basis for a query mechanism that goes beyond keyword search.

This deliverable (D3.2.1 Module for extracting software artefacts from storyboard) describes the various components of the module developed for the task outlined above, including (1) an ontology that defines a hierarchy of concepts and relations for representing dynamic elements of software projects, (2) a diagram editor for storyboards that allows developers to define system scenarios that are mapped to the ontology, and (3) an aggregated ontology which provides a unified view of the system including all the static and dynamic concepts of a software system as defined in Task 3.1 and Task 3.2, respectively. It is accompanied by the respective prototype, D3.2.2 Module for extracting software artefacts from storyboard, that is the software module implemented.

1 Introduction

Deliverable D3.2 (Module for extracting software artefacts from storyboards) describes the components implemented for converting dynamic aspects of a software system to a formal representation that maps to the S-CASE ontology. This deliverable is part of Work Package 3 (WP3), which aims to extract requirements from multi-modal input.

1.1 WP3 Objectives

The main goal of WP3 (Multi-modal information processing) is to design the mechanisms for efficiently extracting requirements from formal models such as UML diagrams, as well as from text and images. Additionally, WP3 will design and implement the Question-Answering mechanism that will serve as the user interface for querying on software artefacts. The WP has four specific objectives:

- To recognize software requirements expressed in unstructured and semi-structured English text and formally represent them as static system aspects (T3.1).
- To analyse dynamic scenarios in the form of storyboards and provide a representation of the dynamic view of software projects (T3.2).
- To transform XMI-based UML diagrams into the S-CASE ontology and to semantically analyse images of UML diagrams (T3.3).
- To develop a question answering system that will allow developers to pose queries about the software components in the S-CASE repository (T3.4).

This deliverable focuses on the second objective. We describe the scope of the corresponding task in more detail in the following subsection.

1.2 Scope of Task 3.2

This deliverable reports on work performed for Task 3.2, which comprises the following subtasks:

- analysis of the dynamic features of software projects,
- definition of a structure to represent these features, and
- definition of a storyboard representation in order to allow the developer to describe dynamic system scenarios.

Additionally, a unified view of the static and dynamic concepts of a software system has to be defined, which includes the features recognized in Task 3.1 and Task 3.2 respectively. Work on these tasks has resulted in the following contributions described in this deliverable: (1) an ontology that defines a hierarchy of concepts and relations for representing dynamic elements of software projects, (2) a tool for creating and editing storyboards that are mapped to the ontology, and (3) an aggregated ontology which provides a unified view of the system including all the static and dynamic concepts of a software system.

Since Task 3.2 provides an overall view of the outcome of the first three tasks of WP3, it covers the main scope of the Reqs2Specs module of S-CASE and provides the specifications of the software project added by the developer. These specifications shall be communicated to both the CIM of the MDE engine and the components for finding functionally equivalent web services, which are part of tasks T2.3 and T4.3, respectively.

1.3 Structure of this Deliverable

The document is structured as follows. Section 2 describes the state-of-the-art on UML diagrams for the dynamic view of software projects and the graphical editors for creating these diagrams. Section 3 presents an ontology developed for formally representing dynamic artefacts of software projects. Section 4 provides information on the Storyboard Creator, a tool designed for creating and editing storyboards. Section 5 describes the aggregated ontology of S-CASE which includes a unified view of software projects. Finally, Section 6 summarizes our progress on Task 3.2 focusing on the dynamic and unified system representations.

2 State-of-the-art: UML Diagrams and Graphical Editors

This Section describes the diagramming techniques of the Unified Modeling Language (UML) for representing dynamic scenarios of software projects. Additionally, we provide a state-of-the-art on graphical diagram editors to justify the need for creating a new diagram editor for storyboards.

2.1 UML Diagrams for the Dynamic View of Software Projects

2.1.1 Structure and Behaviour Diagrams

According to the UML specification [1], there are two major kinds of diagrams: Structure diagrams and Behaviour diagrams. Structure diagrams are used to present the static structure of the objects in a software system. In specific, these types of diagrams include the objects of a system and possibly the relations among them, without however including any details for the scenarios in which these relations appear. Examples of Structure diagrams include Class diagrams, Component diagrams, etc.

The dynamic behaviour of the objects in a software system is illustrated using Behaviour diagrams. These types of diagrams actually depict system scenarios where objects may interact, thus describing a series of events for the system. Note that these events may or may not change the state of the objects, i.e. even a stateless system may have to be described using Behaviour diagrams to ensure its usage scenarios are clear. Examples of Behaviour diagrams include Activity diagrams, State diagrams, etc.

In the context of this deliverable, we focus on the dynamic view of software projects, which is described using Behaviour diagrams. Although there are several diagrams that cover the dynamic aspects of a system, WP3 concerns only diagrams at requirements' level, hence including representations of high level entities, e.g. Activity diagrams, and excluding scenarios of low level software entities, e.g. Sequence diagrams. Consequently the following paragraph describes the usage of an Activity diagram and examines whether it fits the RESTful paradigm of S-CASE.

2.1.2 Activity Diagrams

Activity diagrams are graphical representations of workflows that describe usage (and system) scenarios in the form of consecutive actions. They also support conditions, iterative flows, and concurrency. An example of an Activity diagram for project Restmarks [2] is shown in Figure 2.1. In this diagram, the scenario "Create bookmark" is described. Restmarks is a service that can be seen as a social network where each user can share his internet bookmarks. Additionally, the user can add tags to his/her bookmarks, create, modify, or delete existing bookmarks and search for his/her private bookmarks and/or public bookmarks of other users[4].

[4] Throughout this deliverable, we will use project Restmarks as an example software project that is expected to be prototyped using the tools provided by S-CASE.

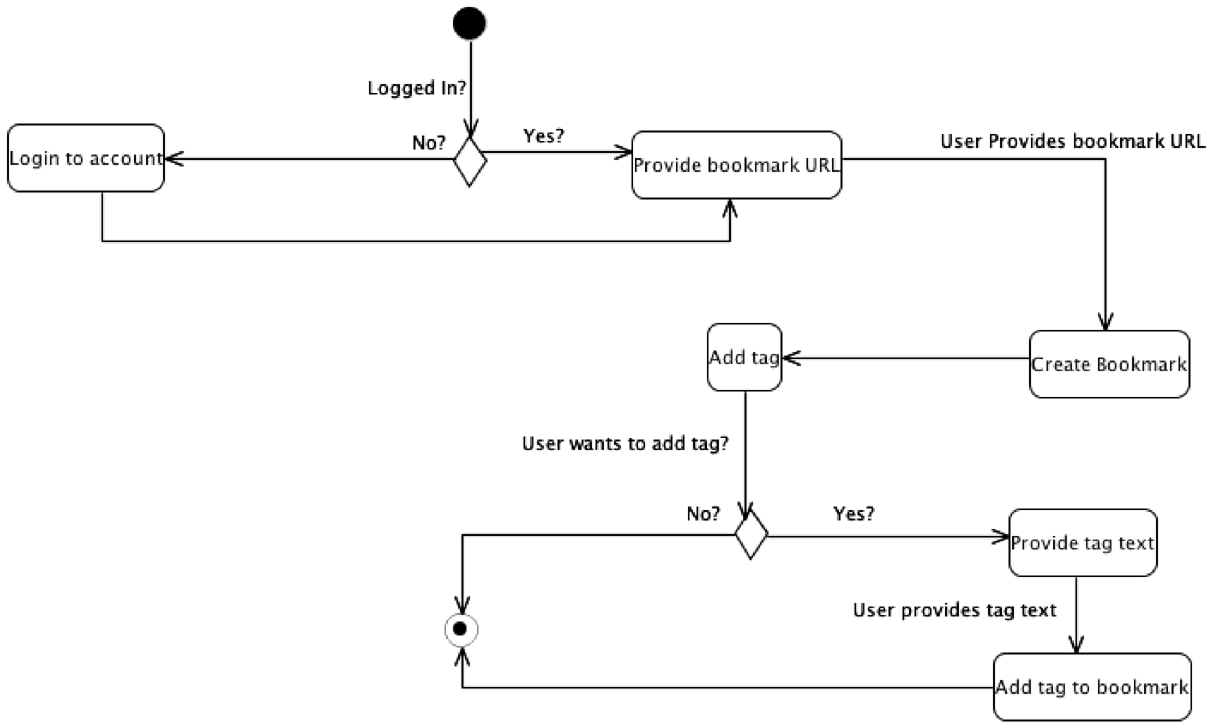


Figure 2.1 Example activity diagram for the use case "Create bookmark" of Restmarks

According to the scenario depicted in Figure 2.1, the user has to initially be logged in to the system. After that, creating a bookmark requires providing its URL and then the user is asked to optionally add tags to the newly added bookmark.

The activity diagram of Figure 2.1 is certainly a valuable representation for a Requirements engineer. However, in the context of RESTful web services, the diagram may contain several pieces of redundant or unclear information. For instance, note that the prompt of the system for providing the URL of a bookmark is followed by the user action (depicted as an arrow in an activity diagram) of actually providing the URL. This information is therefore redundant. Additionally, RESTful resources and properties of these resources are both included in the same types of activities. Thus, distinguishing among these types is hard. For example, if one isolates the activities of the diagram, they would end up with the following:

- Login to account
- Provide bookmark URL
- Create bookmark
- Add tag
- Provide tag text
- Add tag to bookmark

In this case, we would not be able to determine whether "tag" is a resource or even whether "tag text" is a property. Note also that this diagram is generally well defined. The recognition of "text" as a property of "tag" may be much more difficult if the developer provides an activity "Provide text".

Finally, as already mentioned, activity diagrams may include several other concepts that do not conform to the RESTful paradigm. For instance, an activity diagram may depict concurrent flows of activities. Concurrency is not actually supported by RESTful web services.

In specific, although it is possible for an interface to send two or more queries to a service and expect all the responses in order to continue, in practice the requests will be handled one at a time. Thus, it might be better to avoid such requirements representations since they may be misleading as to the functionality of the service.

2.1.3 Other Dynamic Representations

As noted in the previous subsection, activity diagrams provide a useful representation for the dynamic view of a system, without however totally fitting the RESTful paradigm. The dynamic view of a system, however, can be described using several other representations, either graphical or textual.

The main element that is required to be described is actually the dynamic *scenario*. A scenario describes the flow of actions between two specific states of the system. An example textual representation for a dynamic scenario is given in Figure 2.2.

Feature: Create bookmark	
In order to create a new bookmark	
As a user	
I want to create a new bookmark	
Scenario: User also wants to add a tag	
	Given that the user is logged in When the user selects to create a bookmark Then the system adds a new bookmark When the new bookmark is added Then the user is asked to add a tag When the user adds a tag Then the system adds the tag to the bookmark And gives the new bookmark to the user
Scenario: User does not want to add a tag	
	Given that the user is logged in When the user selects to create a bookmark Then the system adds a new bookmark And gives the new bookmark to the user

Figure 2.2 Example Cucumber scenario for the use case "Create bookmark" of Restmarks

The representation shown in Figure 2.2 is a system behaviour scenario described in the language of Cucumber [3]. In the language of Cucumber, certain keywords are used to provide instruction-based scenarios. *Given* is used to define a condition, *When* is used for a user action, *Then* for a system response, and *And* for connecting two or more actions or responses. Additionally, the structure allows: defining the purpose of each scenario via the phrase *In order to*; the actor of the scenario, via the phrase *As a*; the desired final state using the phrase *I want to*.

Cucumber provides an interesting paradigm for creating scenarios based on behaviour driven development. Its language is strict enough so that it can be easily parsed by NLP tools, while the scenarios are quite descriptive. However, in our case, the use of Cucumber scenarios seems unnatural; alternative flows in scenarios are rather verbose and the language does not really conform to the RESTful paradigm.

2.1.4 Dynamic Representations in the RESTful Domain

Concerning the dynamic representation presented in the previous subsections, they can under certain circumstances be used for engineering RESTful web services. In fact, the S-CASE front-end is expected to handle activity diagrams as part of WP3 (specifically task T3.3). However, the definition of another type of representation is preferred in order to ensure that the developer provides the required information in a more RESTful-compliant manner. In this Section, we describe the main elements of this representation not covered by current literature.

At first, concerning the RESTful paradigm, the main building blocks of services are *resources*. Resources may seem similar to objects in the Object-Oriented world, however their nature is different in the way they are processed. Resources are processed in four specific ways, they are created, read, updated, and deleted using the four common HTTP verbs (Post, Get, Put, and Delete). Objects, on the other hand, are handled using any possible action verb since they are constrained by any architectural paradigm. Additionally, the concept of *properties* or *parameters* of resources does not fit well the Object-Oriented point of view. Resources, on the other hand, utilize properties as parameters of HTTP verbs, e.g. retrieving a “bookmark” may require to issue a “Get” command and providing its “id” as a parameter.

Thus, in our case, we require a representation that is highly descriptive for the elements of RESTful web services. Furthermore, the designed representation has to be concise to avoid cluttering the main elements of the simple RESTful scenarios. In specific, we focus on creating a diagram type that covers the resources, the actions on them as well as the parameters of these actions. Additionally, our representation must allow action flows, as well as multiple alternative scenario flows via the use of conditions. This representation is covered by Storyboard diagrams which will be analysed in Section 4.

2.2 Graphical Editors

In this Section, we present current UML graphical editor tools and discuss whether they are compatible with our needs as tools for known UML diagrams and specifically for creating storyboards.

Although there are several UML tools [4], most of them do not fit the paradigm of S-CASE. In specific, the selected tool must have the following prerequisites:

1. The tool must be open-source;
2. It must be cross-platform;
3. The modelling capabilities of the tool must support importing and exporting models in XMI format (i.e. draw-only tools are not sufficient);
4. The support for the tool and its community should be broad.

Given that several popular UML tools are commercial, prerequisite 1 is not always easy to cover. Additionally, there are several tools that cover the first two prerequisites but fall short on current support, i.e. their latest stable release is more than 3 years before. These tools are also not preferable since they may not cover the latest editions of UML.

So, for instance, ArgoUML [5] is a rather popular tool, however it has not been updated since 2011 [4]. StarUML [6], on the other hand, has been updated recently (2014), yet its reliance on Delphi makes it difficult to cover the cross-platform criterion.

Two well-known tools that fit our requirements are Papyrus [7] and Modelio [8]. They are both open-source and cross-platform since they are built as extensions to the Eclipse platform. We can also safely assume that a well-established community of developers is accustomed to Eclipse-based UML tools, given that the modelling capabilities of Eclipse are known to a fair share of developers. Furthermore, since S-CASE is going to be connected to the Eclipse platform, using a similar look and feel tool is certainly preferable.

Both Papyrus and Modelio cover all the prerequisites posed in the previous paragraphs. They are also both continuously supported and their latest stable releases at the time of the writing are within 2014. Additionally, both tools support XMI representations and are generally sufficient for most UML diagram types. The similarities and differences between those tools are actually out of the scope for this deliverable[5], since the main question is whether they are suitable for creating and editing storyboards.

Although these tools are quite useful for known UML diagram types, they are not a good fit for creating special-purpose storyboards. The diagram types supported by Papyrus and Modelio are used to describe a large variety of systems. In our RESTful paradigm, storyboards are designed to be simple dynamic scenarios of the system; hence using a fully complex UML tool would be an overstatement. Additionally, storyboards (as any other diagram types) have their own rules as to the available model elements and the relations among them (see Section 4). These rules are not supported by any of the aforementioned UML tools.

Given that we create a new type of diagram, finding a tool to support creating these types of diagrams out-of-the-box is actually impossible. From a technical point of view, when creating a new diagram type, the description of the diagram is what we call the diagram *meta-model*. A meta-model provides a focused detailed description of the model of a diagram type. In fact, any type of diagram has one such meta-model, use case diagrams, activity diagrams, etc. Thus, in our scenario, what we require is not a diagramming tool since it would not have the meta-information to “understand” our diagrams; we need a framework that allows us to provide this meta-information so that the produced diagram editor can handle storyboards.

[5] The interested reader is referred to the wiki discussion for the UML tools used by S-CASE [6].

2.2.1 Creating Graphical Editors

Since common UML tools do not support the features described in the previous subsections, we decided to design a new tool for creating storyboards. In this Section, we discuss the main alternatives for creating graphical editors.

Since S-CASE has integration with the Eclipse IDE, a rational choice for creating a diagram editor is to use the capabilities of Eclipse for creating diagrams. Currently, the most well-known infrastructure for developing graphical editors in Eclipse is the Graphical Modeling Framework (GMF). The GMF runtime of Eclipse provides a generative component and runtime infrastructure for developing graphical editors based on the Eclipse Modeling Framework (EMF) and the Graphical Editing Framework (GEF). We analyse these frameworks in the next paragraph.

While selecting GMF we had to choose between using the “plain” framework, and using some more high-level infrastructures provided by other projects. In specific, two projects that certainly draw the attention of the respective community are Graphiti [10] and EuGENia [11]. However, using GMF itself allowed us better finegraining of the editor. Additionally, at the time of writing, Graphiti is in the incubation phase while EuGENia provides even more abstraction by using a language (called Emfatic) to provide all models in one large file.

2.2.2 Eclipse Graphical Modeling Project

2.2.2.1 Overview

As noted in the previous paragraph, the GMF runtime relies on two frameworks, EMF and GEF. EMF is a framework and code generation facility that allows building applications based on a *meta-model* [12]. The meta-model is actually the core of an EMF project, thus it is called *ecore*. Thus, an ecore file describes how the data is structured in packages, classes, enumerations, types, etc. After that, any model created by the user of the application has to comply with the rules defined by the meta-model.

GEF is a framework used to create graphical editors [13]. It requires a model that has to be designed beforehand, usually using EMF. GEF provides several editing capabilities, including canvas as well as tooling. Note that GEF does not actually validate any model. This has to be accomplished by the underlying model itself (EMF). A GEF projects is defined by two parts: the *graphical definition*, and the *tooling definition*. The former contains the main rules about diagram editing, including e.g. the allowed shapes of each diagram node, the width of the diagram edges, the diagram layout, etc. The tooling definition involves the toolbox, i.e. the palette, with the shapes of a diagram. Both parts have to be defined carefully since they comprise the final view of the user.

Creating an EMF model and then attaching it to a GEF editor is a difficult procedure. It involves writing several lines of code for every simple connection between the two models. Even if one could accomplish this, the final application would have several transparency problems, e.g. changing the model would result in a non-compliant GEF that would have to be changed manually. GMF, provided by Eclipse, is an interesting solution to the above problem. The framework provides a straightforward way of combining the two models, EMF and GEF. The framework allows creating the models in an isolated manner and combining them by providing a *mapping* between the elements of the two models.

2.2.2.2 Workflow of Creating Diagram Editors

The components and models used during GMF-based development are shown in Figure 2.3.

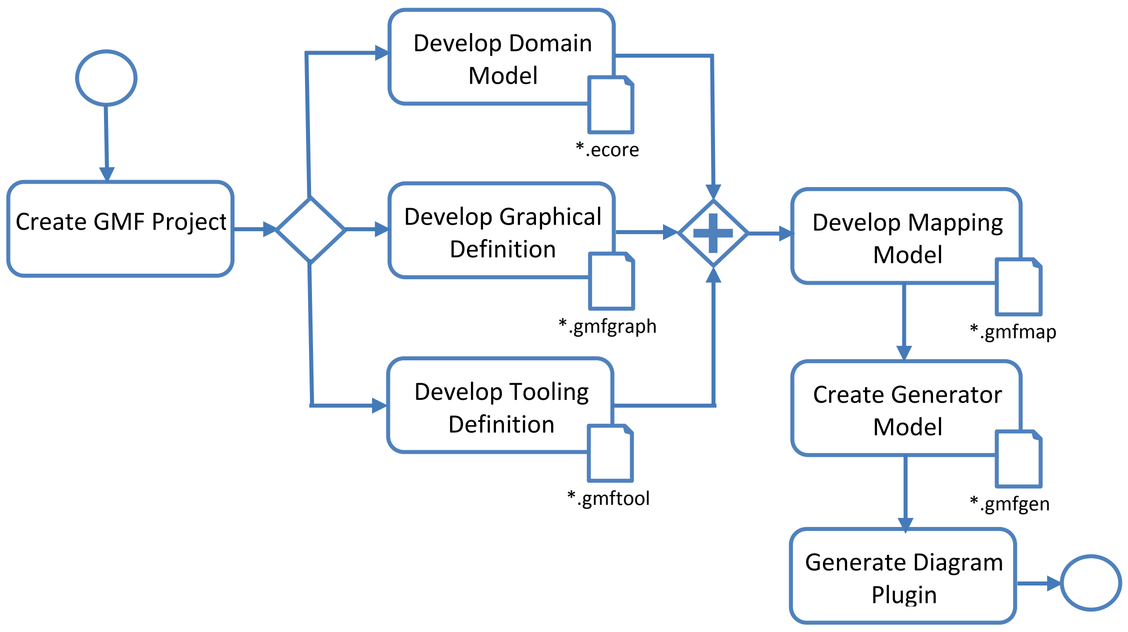


Figure 2.3 Workflow of using GMF for creating Diagram Editors, as shown in [14]

As shown in this diagram, upon creating a GMF Project, we have to develop three different parts of the system. The *Domain Model* is the core of the system; it is actually the ecore model of the EMF. The two other parts, the *Graphical Definition* and the *Tooling Definition*, are the two parts of the GEF analyzed in the previous subsection. Given the domain model, GMF creates initial files for these two parts that have to be edited in order to provide the necessary information for the diagram editor.

Upon creating the EMF and GEF parts, the most difficult step of this procedure is to develop the *Mapping Model*. This is, however, the main facility provided by GMF. Using the framework, the mapping model is created semi-automatically. Of course, the developer has to be very careful about the mapping provided by the system. Generally, simple objects and relations of the meta-model are safely automated. However, any slightly complex object (e.g. an edge with a label) is usually not correctly mapped.

The *Generator Model* is the final model that has to be created. The model is initiated by the framework according to the mapping model. In short, the generator model is the mapping model including options for generating the plugin. Thus, any configurations such as file extensions of the plugin, context menus, etc. have to be defined in this model.

The final step is to generate the diagram plugin. Although this is easily accomplished, any other improvements have to be made on generated code so the models of the GMF should be carefully designed.

2.3 Task Contributions and Progress beyond the State-of-the-art

As noted in the previous subsections, dynamic representations found in current literature are not always capable of effectively describing the dynamic view of software systems. Modelling the dynamic view of a system and designing a representation that allows developers to describe it are two quite important contributions of this task. Additionally, the design of a unified view of software projects represented using an aggregated ontology is another meaningful contribution that effectively illustrates the expected outcome of WP3. As part of working on these directions, we had to achieve significant progress beyond the current state-of-the-art. In specific, the work on this task includes the following main contributions:

- The design of a dynamic ontology that is used to represent the dynamic view of a software project. This ontology covers the possible action-flow representations as long as they are compatible with the RESTful paradigm.
- The development of a dynamic representation that is oriented towards the main concepts of the RESTful paradigm. This representation is designed in the form of storyboards, i.e. system scenarios described in the form of diagrams. As part of this contribution, we can also distinguish the following progress points:
    - Upon analysing the current state-of-the-art on dynamic representations, we claim that this new diagram type is required since no other representation, either graphical or textual, can describe the dynamic view of RESTful services effectively. Current UML diagram types and textual representations are oriented towards the Object Oriented paradigm; thus, they cannot describe the resources of a RESTful service, or the actions and the properties on these resources, without compromising their semantics or introducing verbosity.
    - Since current graphical editors are not capable of handling the creation and editing of storyboards, we design and implement a new tool to sufficiently meet this requirement. Our tool allows designing storyboards based on a robust non-verbose meta-model, since it is based on the well-known Eclipse IDE. Additionally, the main prerequisites for the S-CASE tools are also met, as our tool is open-source and cross-platform, while it allows coupling with components supporting the XMI representations of Eclipse.
- The design of an aggregated ontology of software projects. This contribution is actually a central element of S-CASE since it focuses on the main scope of this work package. Using this ontology, we are now able to describe the main elements of a RESTful system, while also preserving the connection between the system and its requirements. Furthermore, this unified system representation shall form the basis for designing the system using the Model-Driven Engineering (MDE) components of S-CASE and the components for finding functionally equivalent web services.

3 Ontology for the Dynamic View of Software Projects

This Section concerns the design of an ontology for storing information derived from the dynamic view of software projects. This ontology shall include information from storyboards, and generally any dynamic information derived from other types of input (e.g. activity diagrams).

Since ontologies provide a structured means of storing information and linked data, the use of an ontology for our scenario seems well justified. System objects, actions, properties, etc. can be mapped to *Web Ontology Language (OWL) classes* and *properties*[6]. Similarly to previous deliverables, we use Protégé for visualizing and designing our ontology [15]. OWL classes and individuals are drawn as rounded squares (with different colours), and properties are drawn as arrows. The shapes and arrows have labels that hold the name of each class or property, except from the `has_subclass` property (continuous arrow), which is given unlabelled in order to avoid cluttering the visualizations.

The following subsections present the ontology for the dynamic view of the system and illustrate its instantiation using examples.

3.1 Ontology Overview

The main elements of dynamic system representations are flows of actions among system objects. Given that the OWL includes classes and properties, actions can be represented as resources and flows can be described using properties. Additionally, the properties of system objects can also be mapped to OWL properties.

3.1.1 Ontology Class Hierarchy

The class hierarchy of the ontology is shown in Figure 3.1. Anything entered in the ontology (any `owl:Thing`) is a `Concept`. Instances of class `Concept` are further divided in the types of `Project`, `ActivityDiagram`, `AnyActivity`, `Actor`, `Action`, `Object`, `Condition`, `Transition` and `Property`.

`Project` refers to the project analyzed while `ActivityDiagram` stores each diagram of the system. Note that `ActivityDiagram` covers all dynamic view diagrams of the system, i.e. it is not limited to activity diagrams, but also storyboards and generally any diagrams with dynamic flows of action. When instantiating the ontology, `Project` and `ActivityDiagram` can be used to keep its structure reversible. Since each project has several diagrams and each diagram has several other concepts (see next subsection for relations), one can reconstruct the diagrams of the project with their respective concepts.

`AnyActivity` is one of the most central OWL classes of the ontology. It involves all activities shown in a diagram. This class is further distinguished in the following OWL subclasses:

[6] Note that in the context of S-CASE, we use OWL since it is a well-known established standard of current research and industry communities. For an extensive review of OWL languages and tools, the reader is referred to the deliverable 4.1.

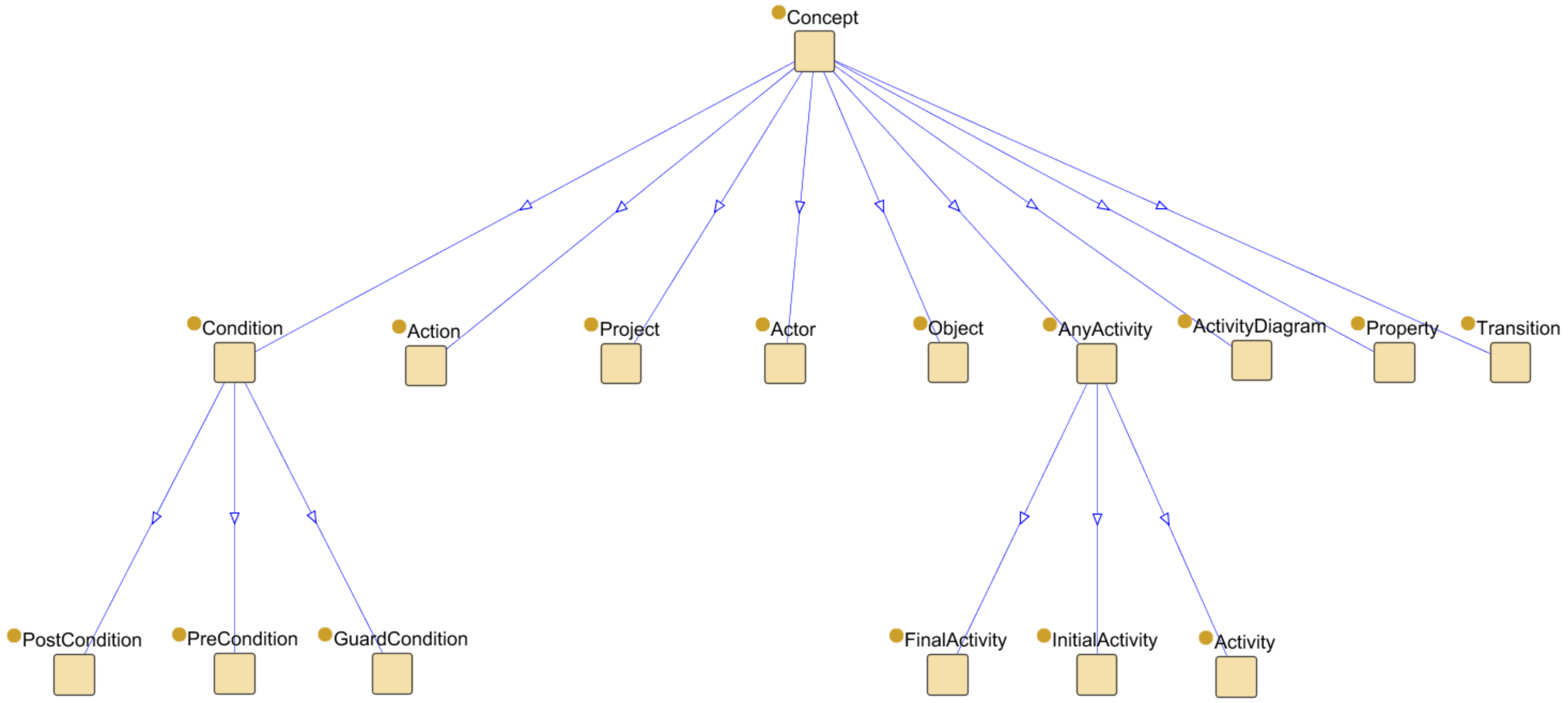


Figure 3.1 Dynamic Ontology Class Hierarchy

- `InitialActivity`: refers to the initial state of the diagram. It does not have a name but it holds the precondition of the diagram.
- `Activity`: any activity of the system, e.g. "Create bookmark"
- `FinalActivity`: the final state of the diagram that holds any postconditions

As noted, activities are the main building blocks of dynamic system representations. In most cases, the information contained in an activity can be stored in several other concepts. These concepts further instantiate the OWL classes `Actor`, `Action`, and `Object`. So, for example, an `Activity` "Create bookmark" instantiates also the `Action` "create" performed on the `Object` "bookmark" by the (implicit) `Actor` "user". This derivation is performed using NLP methods as described in deliverable 3.1 of this work package.

Furthermore, the OWL class `Property` is closely related to `Activity`. Any action of the system may require one or more input properties. For instance, performing a "Create bookmark" may require the newly added bookmark's "name" or its "id" or both. In this case, "name" and "id" are instances of class `Property`.

Another important building block of activity diagrams and storyboards are transitions. Transitions are actually the elements that describe the flow of activities. 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` (more on the connection of these OWL classes in the next subsection). Finally, an instance of `Condition` can be one of the following classes:

- `PreCondition`: refers to the condition that has to be met for the diagram flow to be possible. For example, "the user has to be logged in" in order to create a bookmark.
- `GuardCondition`: a condition that "guards" the execution of an activity of the system along with the corresponding positive answer, e.g. "Create tag" may be guarded by the condition "does bookmark belong to the user? Yes", while the opposite `GuardCondition` "does bookmark belong to the user? No" shall not allow executing the "Create tag" activity.
- `PostCondition`: includes criteria that have to be met after the final state of the diagram

The aforementioned OWL classes cover all elements present in dynamic system representations. The next subsection focuses more on the relations among these elements.

3.1.2 Ontology Properties

The properties of the ontology define the possible interactions between the different classes. In the context of the ontology defined in this Section, we distinguish between two types of properties: *high-level properties* and *low-level properties*. The former involve interactions at inter-diagram level, whereas the latter involve relations between elements of a diagram.

3.1.2.1 High-Level Ontology Properties

Concerning diagram-level, we define the properties shown in Table 3.1. As shown in that Table, several properties are bidirectional.

Table 3.1 High-level Properties of the Dynamic Ontology

OWL Class	Property	OWL Class
`Project`	`project_has_diagram`	`ActivityDiagram`
`ActivityDiagram`	`is_diagram_of_project`	`Project`
`ActivityDiagram`	`diagram_has`	`Actor`, `AnyActivity`, `Transition`, `Property`, `Condition`
`Actor`, `AnyActivity`, `Transition`, `Property`, `Condition`	`is_of_diagram`	`ActivityDiagram`
`ActivityDiagram`	`diagram_has_condition`	`PreCondition`, `PostCondition`
`PreCondition`, `PostCondition`	`is_condition_of_diagram`	`Requirement`

The high-level properties shown in Table 3.1 cover the interactions among the main classes of the ontology. In specific, 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`. Note that `Action` and `Object` are not included in this high-level view of the system since they are covered by the low-level properties of the next paragraph. This is quite rational since they are not actually elements of the diagram; instead, they are derived from its elements. In the case of `Actor`, it is possible that it is given or not given by the diagram. For example, in activity diagrams it is common to assume that the activities performed by the system are shaped as rectangles whereas user activities are defined as labels on the diagram arrows. So, we keep `Actor` as one of the main diagram classes to cover this case.

3.1.2.2 Low-Level Ontology Properties

With the term "low-level properties" we define the properties that cover the interactions among the different ontology classes, excluding `Project` and `ActivityDiagram`. We can further refine these properties in the ones defining the flow of activities in a diagram and the ones defining the relations among the rest elements (including implicitly derived elements, e.g. `Action` or `Object`). These properties are given in Table 3.2 and Table 3.3, respectively.

Table 3.2 Low-level Properties of the Dynamic Ontology, for the Flow of a Diagram

OWL Class	Property	OWL Class
Activity	activity_has_property	Property
Property	is_property_of_activity	Activity
Transition	has_source	Activity, InitialActivity
Activity, InitialActivity	is_source_of	Transition
Transition	has_target	Activity, FinalActivity
Activity, FinalActivity	is_target_of	Transition
Transition	has_condition	GuardCondition
GuardCondition	is_condition_of	Transition
GuardCondition	is_opposite_of	GuardCondition

As shown in Table 3.2, 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`. This flow is also illustrated in Figure 3.2.

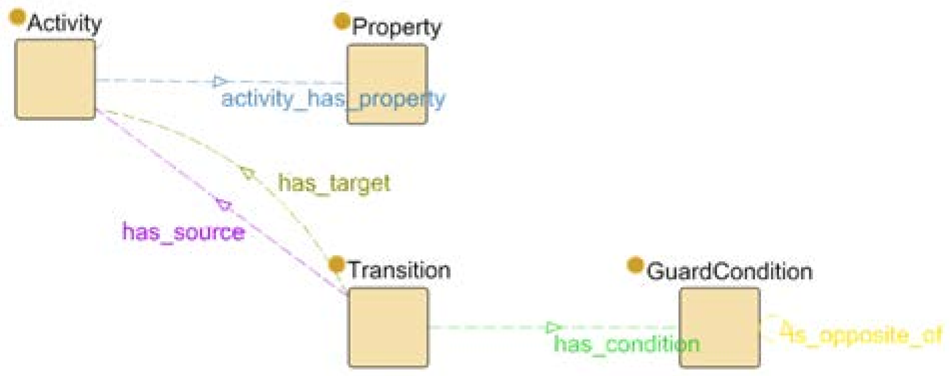


Figure 3.2 Low-level Ontology Properties depicting the Flow of Activities

Finally, the properties (and their reverse) of the implicitly derived elements for each diagram are shown in Table 3.3.

Table 3.3 Low-level Properties of the Dynamic Ontology, for the Implicit Elements of a Diagram

OWL Class	**Property**	**OWL Class**
`Activity`	`activity_has_action`	`Action`
`Action`	`is_action_of_activity`	`Activity`
`Activity`	`activity_has_object`	`Object`
`Object`	`is_object_of_activity`	`Activity`
`Activity, Condition`	`has_actor`	`Actor`
`Actor`	`is_actor_of`	`Activity, Condition`

As shown in Table 3.3, 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`. Note also that the `has_actor` property connects `Condition` to `Actor`, since certain diagrams may also imply an actor for a condition, e.g. "Does the user want to continue?" has a user actor, whereas "Does the database contain x?" implies a system actor.

3.2 Example Instances

This Section illustrates the use of the ontology for storing information derived from an activity diagram. Note that information extraction from activity diagrams is handled in deliverable 3.3, so in this Section we provide an example for a manually extracted diagram. In Section 4, one can find a full example of deriving information from the storyboards of a software project and instantiating the ontology.

The example we use is the activity diagram of Figure 2.1. In this diagram, one can find several common elements of activity diagrams. At first, we can see there are 6 activities and 2 conditions, so we can add these to the ontology. Additionally, we add 2 more activities for the classes `InitialActivity` and `FinalActivity`. Note that the diagram has no preconditions or postconditions.

Transition naming follows the `FROM__SourceActivity__TO__TargetActivity` convention. Additionally, the instances of `GuardCondition` follow the convention `Condition__PATH__Predicate`. Finally, we derive the instances of `Actor`, `Action`, and `Object`. The instantiation of the ontology for the activity diagram of Figure 2.1 is shown in Figure 3.3.

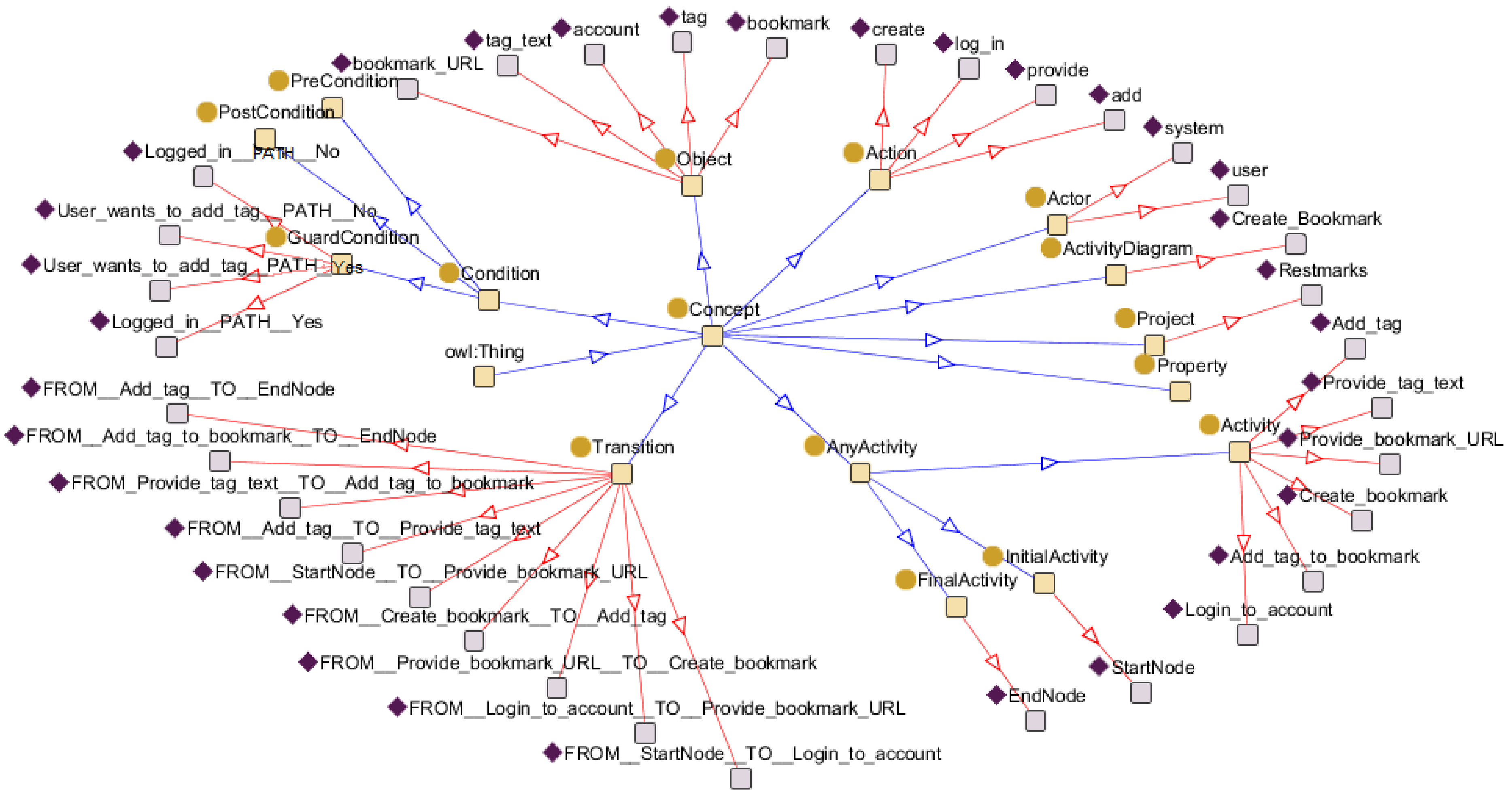

tag_text
account
tag
bookmark
create
log_in
provide
add
system
user
Create_Bookmark
Restmarks
Add_tag
Provide_tag_text
Provide_bookmark_URL
Create_bookmark
Add_tag_to_bookmark
Login_to_account
StartNode
EndNode
bookmark_URL
PreCondition
PostCondition
Logged_in__PATH__No
User_wants_to_add_tag__PATH__No
GuardCondition
User_wants_to_add_tag__PATH__Yes
Logged_in__PATH__Yes
Condition
Object
Action
Actor
ActivityDiagram
Concept
Project
Property
owl:Thing
Activity
Transition
AnyActivity
InitialActivity
FinalActivity
FROM__Add_tag__TO__EndNode
FROM__Add_tag_to_bookmark__TO__EndNode
FROM_Provide_tag_text__TO__Add_tag_to_bookmark
FROM__Add_tag__TO__Provide_tag_text
FROM__StartNode__TO__Provide_bookmark_URL
FROM__Create_bookmark__TO__Add_tag
FROM__Provide_bookmark_URL__TO__Create_bookmark
FROM__Login_to_account__TO__Provide_bookmark_URL
FROM__StartNode__TO__Login_to_account

Figure 3.3 Example Ontology Instantiation of the Activity Diagram "Create bookmark" of Project Restmarks

Concerning the properties given in the previous subsection, we provide here an example for the properties surrounding the action "Provide bookmark URL". These properties are shown in Table 3.4.

Table 3.4 Example Instantiated Properties for the Action "Provide bookmark URL" of the Diagram "Create Bookmark" of Project Restmarks

OWL Individual	**Property**	**OWL Individual**
Provide_bookmark_URL	is_source_of	FROM__Provide_bookmark_URL__TO__Create_bookmark
FROM__Provide_bookmark_URL__TO__Create_bookmark	has_source	Provide_bookmark_URL
Provide_bookmark_URL	is_target_of	FROM__Login_to_account__TO__Provide_bookmark_URL
FROM__Login_to_account__TO__Provide_bookmark_URL	has_target	Provide_bookmark_URL
Provide_bookmark_URL	is_target_of	FROM__StartNode__TO__Provide_bookmark_URL
FROM__StartNode__TO__Provide_bookmark_URL	has_target	Provide_bookmark_URL
Provide_bookmark_URL	activity_has_action	provide
Provide	is_action_of_activity	Provide_bookmark_URL
Provide_bookmark_URL	activity_has_object	bookmark_URL
bookmark_URL	is_object_of_activity	Provide_bookmark_URL
Provide_bookmark_URL	has_actor	user
User	is_actor_of	Provide_bookmark_URL

As shown in Table 3.4, the Activity Provide_bookmark_URL is the source of two transitions (one for each flow coming from the "Logged in" condition) and the target of one transition (towards the "Create bookmark" action). Additionally, the Actor, Action, and Object are found (user, provide, and bookmark_URL) and connected to the activity.

4 Storyboard Creator

As noted in Section 2, we require a representation for the dynamic view of the system that is more compliant with the RESTful paradigm compared to common UML representations. In this Section, we present this representation in the form of *storyboards*. Storyboards are dynamic system scenarios that describe common flows of action in a software system.

As part of work done in Task 3.2, we designed and implemented a tool for creating storyboards. Our tool, created using Eclipse GMF (see Section 2), is named *Storyboard Creator*. The following subsections provide the main architecture of the Storyboard Creator and present the models designed for the application. These models actually provide the meta-model of storyboards, i.e. the elements, the connections, and the rules for creating storyboards. After that, the usage of the tool is illustrated using example storyboards and the respective ontology representations.

Subsection 4.1 presents the requirements for the Storyboard Creator. The information in subsections 4.2 and 4.3 is given thoroughly in the Technical Manual [16] and the User Manual [17] of Storyboard Creator respectively. In this deliverable, we provide an analysis of the architecture and functionality of the tool to illustrate its usage on Task 3.2.

4.1 Requirements of the Storyboard Creator

We mainly focus on REST services, so the basic building blocks of a storyboard are actions performed on objects (resources) of the system. Each action can either be a CRUD action (i.e. Create, Read, Update, Delete) or a non-CRUD action, and it can contain an arbitrary number of properties, which can be interpreted as parameters of this action. Actions connect to each other via directional edges. Finally, storyboards contain conditions that represent preconditions for the actions following them. The functional requirements for the Storyboard Creator are shown in Figure 4.1.

FR1. The user must be able to select CRUD actions of activities
FR2. The user must be able to create new action by providing a non-CRUD verb and the corresponding object of the system.
FR3. The user must be able to change the type of an action, including one of the four CRUD types or a non-CRUD action.
FR4. The user must be able to add one or more properties to each action.
FR5. The user must be able to add conditions that have two output paths.
FR6. The user must be able to add preconditions to each diagram.
FR7. Each storyboard must have an initial and a final node.
FR8. A storyboard can be an action for another storyboard (nested storyboards).
FR9. The user must be able to modify the properties of each entity of the system, including its name, its description, and other relevant to each object type.

Figure 4.1 Functional Requirements of the Storyboard Creator

Apart from its functional part, which is to allow developers to specify dynamic usage scenarios of their system, the tool must also have certain non-functional features, e.g. user friendliness. The non-functional requirements for the Storyboard Creator are summarized in Figure 4.2.

NFR1.	The user interface must be intuitive.
NFR2.	The user must be presented with different options for each node of the storyboard (action, property, or condition), including copying it, modifying it (e.g. altering its multiplicity) and deleting it.
NFR3.	The diagrams must be editable either by using context menus or by using properties panes.
NFR4.	The system must output the storyboards in a comprehensive XML format that will contain the model of the diagram.

Figure 4.2 Non-Functional Requirements of the Storyboard Creator

Since we selected GMF to design our tool, most non-functional requirements are met, thus we focus on the functional aspects of the tool.

4.2 Design of the Storyboard Creator

4.2.1 Architecture

As noted in Section 2, the components used during GMF development reflect the models of EMF and GEF. In the case of Storyboard Creator, we created the models shown in Figure 4.3.

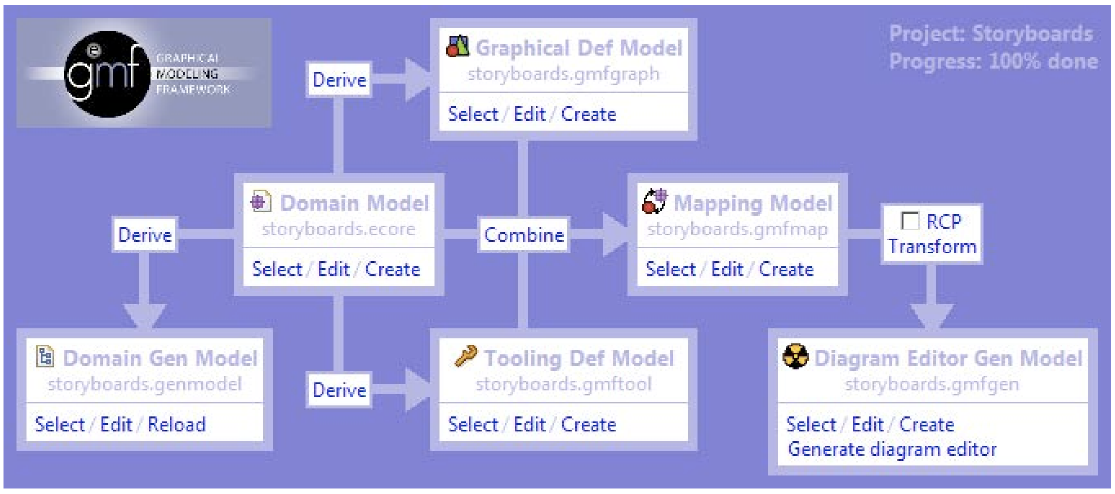


Figure 4.3 GFM Dashboard Architecture for the Storyboard Creator

Figure 4.3 shows the GMF dashboard of the Eclipse IDE. As shown, the core of the GMF development procedure is the Domain Model, which is actually the meta-model to be used for the storyboard models. Using the capabilities of GMF, we have designed this model and generated the Domain Gen(eration) Model. The latter is actually a representation generated by the Domain Model which is however suitable for interacting with the final application. This is actually required to account for model validation.

After that, we designed the Graphical Def(inition) Model and the Tooling Def(inition) Model using also the deriving capabilities of GMF. These two models along with the Domain Model were combined in order to produce the Mapping Model.

After reviewing the Mapping Model and making certain important changes, we transformed it to the final Diagram Editor Gen(eration) Model. Note that the Eclipse GMF allows developing diagram editors either as Eclipse plugins or as standalone RCP applications (i.e. applications including the necessary parts of the Eclipse IDE to execute). We selected to deploy our application as an Eclipse plugin since it is quite flexible and it allows developers to select what to include in their projects. Finally, upon changing certain options concerning the appearance and behaviour of our editor plugin, we generated the diagram editor. The following subsection presents the different models designed for the Storyboard Creator.

4.2.2 Models

4.2.2.1 Domain Model

The domain model is visualized in Figure 4.4.

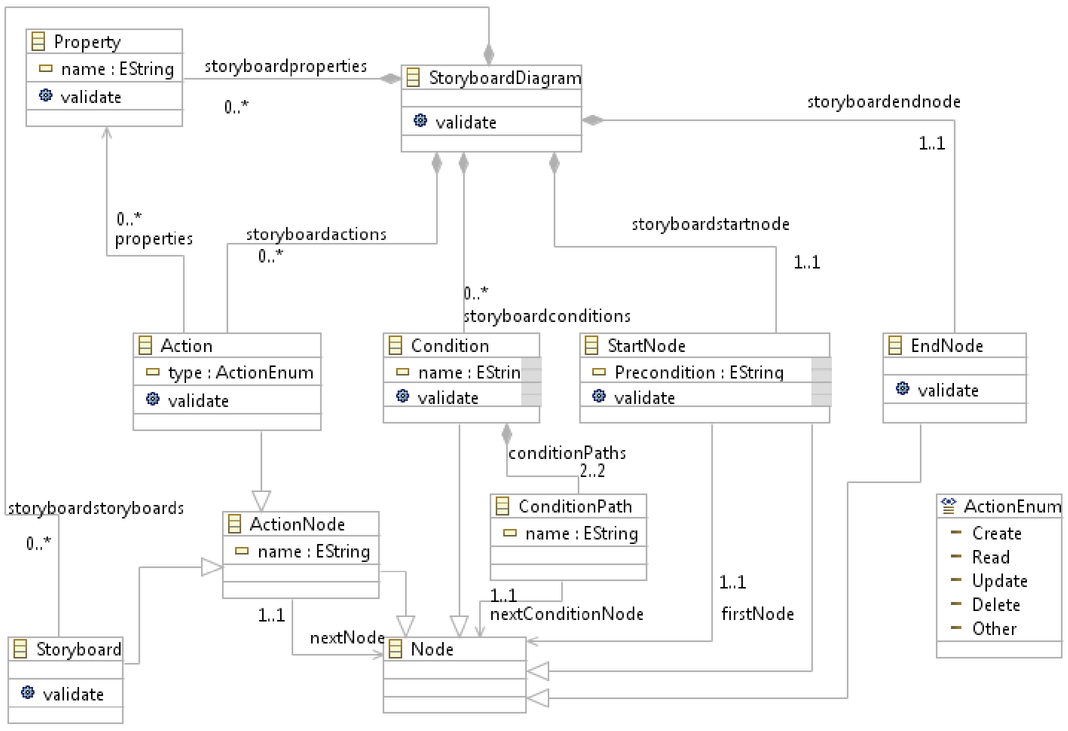


Figure 4.4 Storyboard Creator Domain Model

The most high-level meta-class of the model is the *StoryboardDiagram*. The latter contains all the other classes of the model along with the respective multiplicities. In specific, it contains exactly 1 *StartNode* (*storyboardstartnode*) and 1 *EndNode* (*storyboardendnode*), 0 or more Actions (*storyboardactions*), 0 or more *Properties* (*storyboardproperties*) of Actions, and 0 or more *Conditions* (*storyboardconditions*). Additionally, it contains 0 or more *Storyboards* (*storyboardstoryboards*), allowing nested storyboards.

Most elements of a storyboard belong to one of the subclasses of *Node*. Actions and Storyboards both are subclasses of *ActionNode*, which is a subclass of Node that has a name and connects to exactly one Node of the diagram (*nextNode*). This relationship creates the sequence of Nodes that form a diagram. Note also that Action has a *type*, which can be one of *Create*, *Read*, *Update*, *Delete*, and *Other* (*ActionEnum*). Any Action connects to 0 or more *Properties* (*properties*).

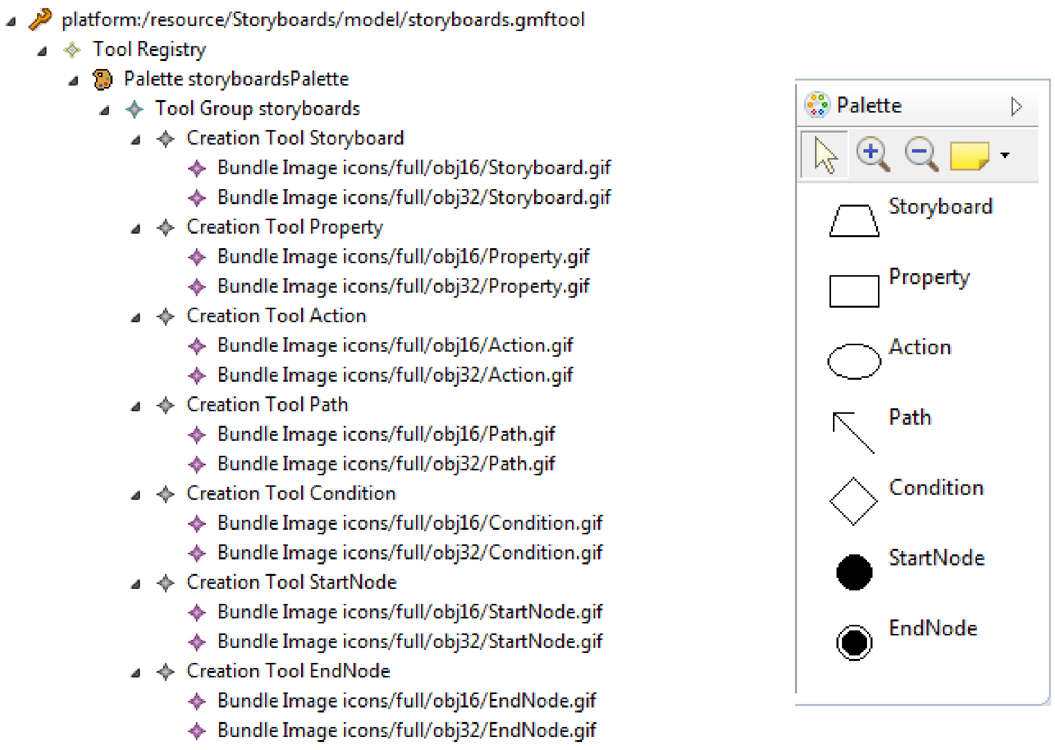
Conditions connect to nodes via *ConditionPaths*. Unlike simple paths, ConditionPaths are defined with a name variable in order to keep the consequent of the Condition. Each condition has exactly 2 ConditionPaths (*conditionPaths*), and each ConditionPath connects to exactly 1 diagram Node (*nextConditionNode*).

Note that EndNode is also a subclass of Node, whereas StartNode is not since no other Node can connect to it. StartNode also contains a string field for any *Precondition* of the diagram and connects to exactly one Node (*firstNode*).

Most elements of the domain model also have a validate function. These functions will be used in order to check whether the diagrams created by the user are valid. Finally, the domain gen model is actually quite similar to the domain model, the main difference being its xml structure.

4.2.2.2 Tooling Model

The tooling model is quite simple. A screenshot of the model in the GMFTool Model Editor of Eclipse is shown in Figure 4.5.



```
platform:/resource/Storyboards/model/storyboards.gmftool
  Tool Registry
    Palette storyboardsPalette
      Tool Group storyboards
        Creation Tool Storyboard
          Bundle Image icons/full/obj16/Storyboard.gif
          Bundle Image icons/full/obj32/Storyboard.gif
        Creation Tool Property
          Bundle Image icons/full/obj16/Property.gif
          Bundle Image icons/full/obj32/Property.gif
        Creation Tool Action
          Bundle Image icons/full/obj16/Action.gif
          Bundle Image icons/full/obj32/Action.gif
        Creation Tool Path
          Bundle Image icons/full/obj16/Path.gif
          Bundle Image icons/full/obj32/Path.gif
        Creation Tool Condition
          Bundle Image icons/full/obj16/Condition.gif
          Bundle Image icons/full/obj32/Condition.gif
        Creation Tool StartNode
          Bundle Image icons/full/obj16/StartNode.gif
          Bundle Image icons/full/obj32/StartNode.gif
        Creation Tool EndNode
          Bundle Image icons/full/obj16/EndNode.gif
          Bundle Image icons/full/obj32/EndNode.gif
```

(a) **(b)**

Figure 4.5 Storyboard Creator Tooling Model and Palette

As shown in Figure 4.5a, creating the tooling model involves selecting which objects shall have a tool in the palette and their respective images. Storyboards, Properties, Actions, Conditions, StartNode and EndNode all have a respective tool. Concerning paths, both Path and ConditionPath objects have the same tool in order to make the diagram editor simple. In Figure 4.5b, one can see the final resulting palette.

4.2.2.3 Graphical Model

The graphical model handles the appearance of all the objects of the diagrams, including their shape, their behaviour (e.g. resizing), their connections with other shapes, either nodes or edges (e.g. where paths are connected), etc. The graphical model contains Nodes and Connections. Each Node or Connection has a respective *Figure Descriptor*.

The Figure Descriptor contains all the information about the figure of each diagram element, i.e. its shape, the size of the element, any labels etc. For example, the Property element has a figure named PropertyFigure, which is of type Rectangle. It also has a Diagram Label named PropertyName. Diagram Labels are actually defined separately and then connected with the other elements. Edges also have their Figure, which is in most cases a simple polyline edge (Polyline Decoration). The ConditionPathFigure, however, also contains the ConditionPathName label. Concerning Storyboard Creator, we defined several figures, while several of these figures are also customized to meet the requirements of our tool. These are analyzed in the following paragraphs.

At first, the Property Figure is a simple Rectangle with a label and the Action Figure is a simple Ellipse with a label. The two elements are shown in Figure 4.6a and Figure 4.6b respectively.

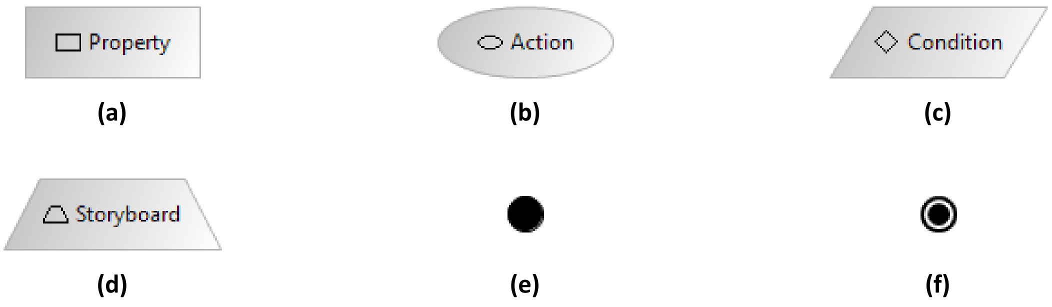


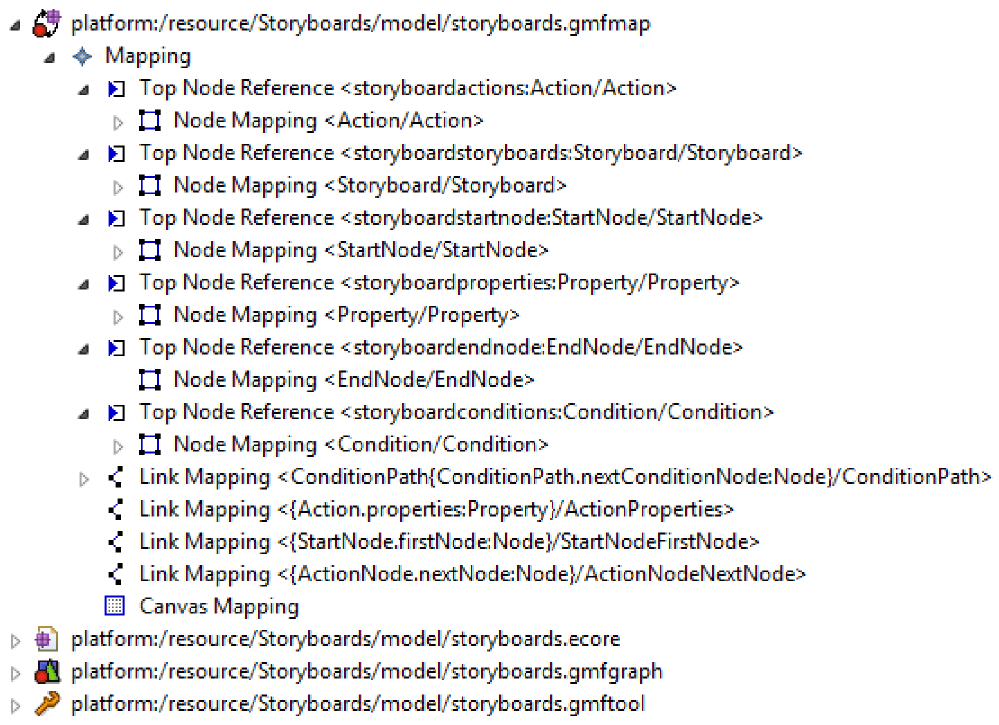
Figure 4.6 Storyboard Creator Figure Descriptors

The Condition Figure and the Storyboard Figure are Scalable Polygons. Scalable Polygons are GEF polygons that allow selecting several points in order to create the shape desired by the developer. In this case, we provided the points so that the Condition Figure is a parallelogram (Figure 4.6c) and the Storyboard Figure is an isosceles trapezoid (Figure 4.6d).

Finally, Figure 4.6e and Figure 4.6f present the StartNode Figure and the EndNode Figure respectively. Note that for the EndNode Figure, we had to create a custom figure since the circle with the dot in the middle is not defined as a default GEF figure.

4.2.3 Combining the Models and Generating the Tool

The mapping model is the result of the combination of the three models, the domain model, the tooling model, and the graphical model. A screenshot of the model in the GMFMap Model Editor of Eclipse is shown in Figure 4.7.



```
platform:/resource/Storyboards/model/storyboards.gmfmap
  Mapping
    Top Node Reference <storyboardactions:Action/Action>
      Node Mapping <Action/Action>
    Top Node Reference <storyboardstoryboards:Storyboard/Storyboard>
      Node Mapping <Storyboard/Storyboard>
    Top Node Reference <storyboardstartnode:StartNode/StartNode>
      Node Mapping <StartNode/StartNode>
    Top Node Reference <storyboardproperties:Property/Property>
      Node Mapping <Property/Property>
    Top Node Reference <storyboardendnode:EndNode/EndNode>
      Node Mapping <EndNode/EndNode>
    Top Node Reference <storyboardconditions:Condition/Condition>
      Node Mapping <Condition/Condition>
    Link Mapping <ConditionPath{ConditionPath.nextConditionNode:Node}/ConditionPath>
    Link Mapping <{Action.properties:Property}/ActionProperties>
    Link Mapping <{StartNode.firstNode:Node}/StartNodeFirstNode>
    Link Mapping <{ActionNode.nextNode:Node}/ActionNodeNextNode>
    Canvas Mapping
platform:/resource/Storyboards/model/storyboards.ecore
platform:/resource/Storyboards/model/storyboards.gmfgraph
platform:/resource/Storyboards/model/storyboards.gmftool
```

Figure 4.7 Storyboard Creator Mapping Model

The mapping model contains the three other models, and also contains a mapping for each element of the diagram. The Canvas Mapping refers to the root of the diagram, while any other elements are shown either as Node Mappings or as Link Mappings. Additionally, any label mappings are included in the respective nodes or links (e.g. the Link Mapping for a ConditionPath includes also a mapping for its name).

Each mapping has a reference to the EMF ecore model and a reference to the GEF element from the graphical model. In addition, the tool for any element is also mapped to it.

Upon creating the mapping model, we generate the Diagram Editor Generation Model. This model is used to generate the source code of our tool. Most elements of our diagram editor have already been defined up to this point. The Diagram Editor Generation Model contains a class for every element of the diagram. Thus, e.g., *ActionEditPart* is a class that refers to Action and functions that correspond to creating, editing, or accessing an Action and its elements (e.g. its name).

Apart from these elements, the model also contains options concerning the palette of the diagram editor, the context menu and the property and preference pages. The default options for most of these settings are usually adequate. In our case, we added two new menu entries for importing and exporting diagrams and we tweaked the file settings so that our tool provides with one file for each storyboard having the .sbd extension.

4.2.4 Refining the Storyboard Creator

Upon generating the tool we had to make some final adjustments. These adjustments concerned the appearance and the validation procedure of Storyboard Creator.

Concerning appearance, most elements have already been defined in paragraphs 4.2.2.2 and 4.2.2.3 that concern the tooling and graphical models of GEF respectively. However, in our case, we wanted to have a personal feel and look in the Storyboard Creator. Thus, we overrode the source code of the two models in order to create the gradient effect of the figures.

Concerning functionality, Storyboard Creator must also have the capability of validating the models (diagrams) created by the user. Certain validation procedures are immediately enforced by GMF. Thus, for example, any ActionNode must connect to exactly one Node. Trying to connect to more Nodes is not allowed by the editor. In several cases, however, this form of validation is not possible. For example, checking that a Property belongs to exactly one Action requires checking the whole diagram model since it cannot be checked using only the domain model. Even when some rules are checked by GMF (e.g. an Action with no name), the default validation messages are sometimes hard to understand, so we overrode them too. Validation rules are checked by overriding the validate function of the generated code model.

4.2.5 Storyboard Creator File Model

Given the domain model and the diagram elements, a file model was defined for the Storyboard Creator. The file extension is .sbd and the file is xml-based. An example diagram and the corresponding .sbd are shown in Figure 4.8 and Figure 4.9 respectively.

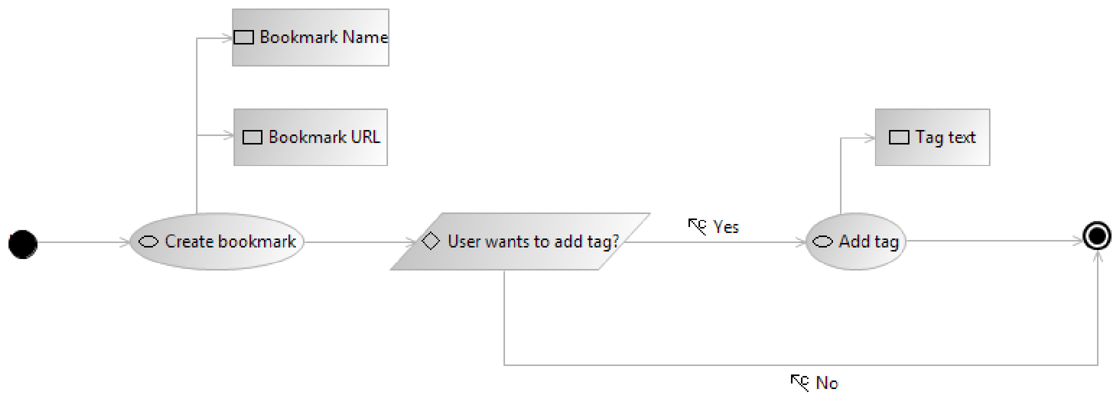


Figure 4.8 Example Storyboard Diagram for the Storyboard “Add bookmark”

As shown in these figures, the model of a storyboard diagram is contained in the xml tag `auth.storyboards:StoryboardDiagram`. Inside this element there are several xml elements, defined by their corresponding tags. These are defined according to the domain model shown in paragraph 4.2.2.1. Thus, e.g., the tag `storyboardactions` is used in order to define an Action, i.e. an object that has an `xmi:type` equal to `auth.storyboards:Action`. Note that any object of the system has a unique `xmi:id`. Any concept of the domain model is simply stored in the sbd following the hierarchy defined in the ecore file. Hence, Actions have `name`, nextNode (defined as xmi:id of the upcoming Node) and properties (defined as array of xmi:ids), while Properties only have a name. Accordingly, StartNode and EndNode are defined with the former having a Precondition and a firstNode, while Condition has a name as well as two conditionPath elements. The conditionPath elements each have a name and a nextConditionNode property.

```
<xmi:XMI xmi:version="2.0" xmlns:xmi=http://www.omg.org/XMI
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:auth.storyboards="http:///auth/storyboards.ecore"
    xmlns:notation="http://www.eclipse.org/gmf/runtime/1.0.2/notation">
    <auth.storyboards:StoryboardDiagram xmi:id="_PjpmkAA8EeSXd67c3thk-A">
        <storyboardactions xmi:type="auth.storyboards:Action"
             xmi:id="_T_8ywAA8EeSXd67c3thk-A"
             nextNode="_YvwJwAA8EeSXd67c3thk-A" name="Create bookmark"
             properties="_VyEQoAA8EeSXd67c3thk-A _wL8C0AdMEeSf0evSNLNfeQ"/>
        <storyboardactions xmi:type="auth.storyboards:Action"
             xmi:id="_bKxasAA8EeSXd67c3thk-A"
             nextNode="_gPGQMAA8EeSXd67c3thk-A" name="Add tag"
             properties="_ccJjAAA8EeSXd67c3thk-A"/>
        <storyboardproperties xmi:type="auth.storyboards:Property"
             xmi:id="_VyEQoAA8EeSXd67c3thk-A" name="Bookmark URL"/>
        <storyboardproperties xmi:type="auth.storyboards:Property"
             xmi:id="_ccJjAAA8EeSXd67c3thk-A" name="Tag text"/>
        <storyboardproperties xmi:type="auth.storyboards:Property"
             xmi:id="_wL8C0AdMEeSf0evSNLNfeQ" name="Bookmark Name"/>
        <storyboardconditions xmi:type="auth.storyboards:Condition"
             xmi:id="_YvwJwAA8EeSXd67c3thk-A" name="User wants to add tag?">
             <conditionPaths xmi:type="auth.storyboards:ConditionPath"
                   xmi:id="_fRW1kAA8EeSXd67c3thk-A" name="Yes"
                   nextConditionNode="_bKxasAA8EeSXd67c3thk-A"/>
             <conditionPaths xmi:type="auth.storyboards:ConditionPath"
                   xmi:id="_hIFGoAA8EeSXd67c3thk-A" name="No"
                   nextConditionNode="_gPGQMAA8EeSXd67c3thk-A"/>
        </storyboardconditions>
    <storyboardstartnode xmi:type="auth.storyboards:StartNode"
        xmi:id="_QcesAAA8EeSXd67c3thk-A" Precondition="User is logged in"
        firstNode="_T_8ywAA8EeSXd67c3thk-A"/>
    <storyboardendnode xmi:type="auth.storyboards:EndNode"
        xmi:id="_gPGQMAA8EeSXd67c3thk-A"/>
    </auth.storyboards:StoryboardDiagram>
    <notation:Diagram xmi:id="_PkGSgAA8EeSXd67c3thk-A" type="Storyboards"
        element="_PjpmkAA8EeSXd67c3thk-A" name="Add Bookmark.sbd"
        measurementUnit="Pixel">
        ...
    </notation:Diagram>
</xmi:XMI>
```

Figure 4.9 SBD file for the Storyboard Diagram "Add bookmark"

Finally, note that the sbd representation contains also information about the diagram (i.e. position of nodes and edges, size, etc.). This information is stored in the elements of the notation:Diagram xml tag. However, this tag contains no information about the model itself. This is very important since it allows for importing and exporting diagrams using the auth.storyboards:StoryboardDiagram of the .sbd file and generating the notation:Diagram.

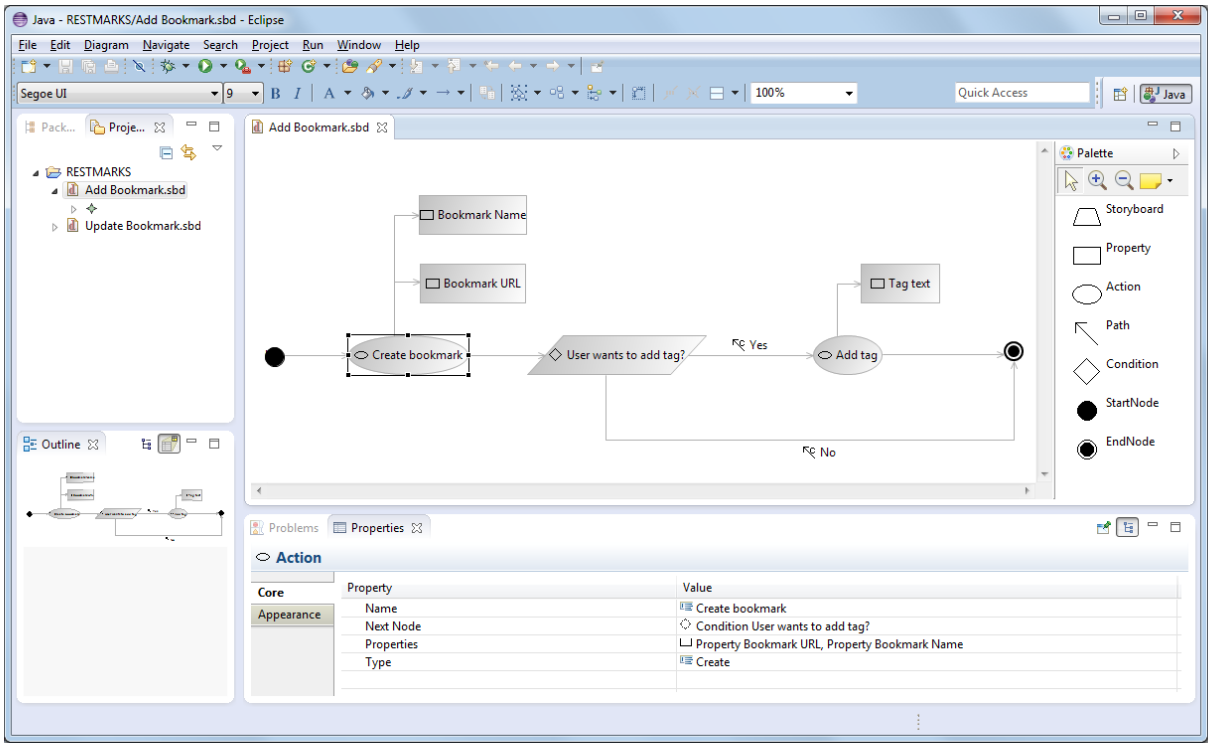
4.3 Usage of the Storyboard Creator

4.3.1 Features

The following website has information about the Eclipse Update Site of Storyboard Creator:

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

One can find detailed instructions on installing and updating the tool in the User Manual [17]. Upon installing the tool, the user is presented with the screen shown in Figure 4.10.



Property	Value
Name	Create bookmark
Next Node	Condition User wants to add tag?
Properties	Property Bookmark URL, Property Bookmark Name
Type	Create

Figure 4.10 Main Screen of Storyboard Creator

As shown in this screenshot, the main views used in Storyboard Creator are:

- The Canvas, in the centre of the screen, where the storyboard diagrams are shown and edited;
- The Palette that includes the possible shapes on the right;
- The Project Explorer on the left that shows the open and closed Storyboard Creator projects;
- The outline in the lower left part of the screen that allows viewing the canvas and navigating (especially when the diagrams are large);
- The Properties tab in the lower part of Eclipse that allows changing specific values for properties of diagram elements;
- The Problems informational tab in the lower part of Eclipse that shows any validation problems for the diagrams.

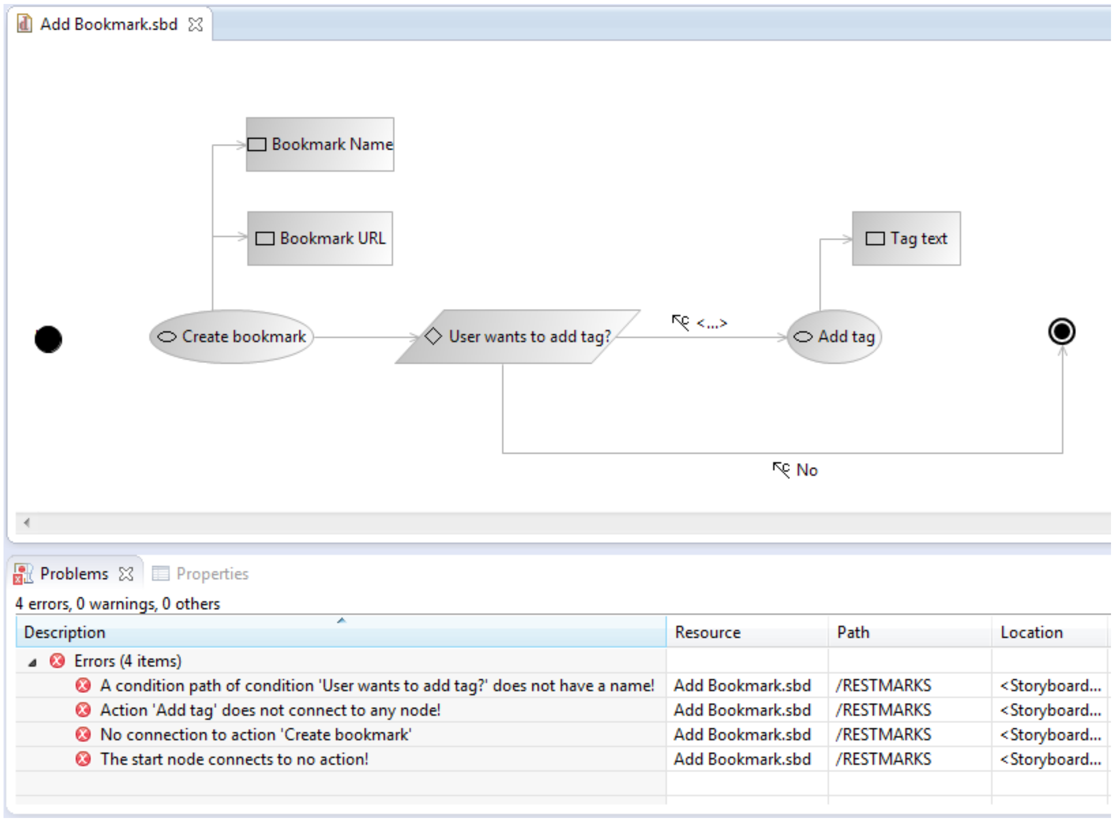
Note that the user is able to change the position of these tabs as he/she would normally do in the Eclipse IDE.

4.3.2 Usage and Validation

Storyboards are stored in Eclipse projects. So at first, one has to create a project by selecting the option *File* and then *New* and *Project…*. The process for creating a storyboard is similar. The user navigates from the *New* menu and selects *File* and *Storyboards Diagram*.

A new diagram is populated with nodes and paths. There are 6 available nodes. Storyboard and Action are similar within the same diagram. Actions, however, represent atomic operations, whereas Storyboards must have their own diagram consisting possibly of several actions. One can also select the type of the action, out of the 4 CRUD types, in the Properties editor of Eclipse. Properties are interpreted as parameters of Actions. They have to be connected to some Action of the diagram. Conditions can be used to split the main flow of a storyboard. Each condition must have exactly two outgoing paths. Each Storyboard Diagram must have exactly one StartNode and one EndNode. The StartNode is the first node of the diagram and the EndNode is the last node of the diagram. Finally, Path is used to connect the nodes of the diagram to one another. Paths have one direction, and in the case of an outgoing Condition path, they also have a label.

Consider an example of a storyboard diagram with errors shown in Figure 4.11.



4 errors, 0 warnings, 0 others

Description	Resource	Path	Location
Errors (4 items)			
A condition path of condition 'User wants to add tag?' does not have a name!	Add Bookmark.sbd	/RESTMARKS	<Storyboard...
Action 'Add tag' does not connect to any node!	Add Bookmark.sbd	/RESTMARKS	<Storyboard...
No connection to action 'Create bookmark'	Add Bookmark.sbd	/RESTMARKS	<Storyboard...
The start node connects to no action!	Add Bookmark.sbd	/RESTMARKS	<Storyboard...

Figure 4.11 Validation Example of Storyboard Creator

In this example, the start node does not connect to any node, the Action "Add tag" does not connect to any node, and there is no connection (i.e. possible path) to the Action "Create Bookmark". In addition, a condition path of Condition "User wants to add tag?" does not have a name. All these errors are reported using messages in the Problems view of Eclipse.

4.3.3 Ontology Instantiation

4.3.3.1 Example Instantiation for a storyboard

Mapping storyboard diagrams to the ontology is straightforward. At first, 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 `activity_has_property` relation. Similarly to activity diagrams, 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 "Add Bookmark" storyboard of Figure 4.8 is shown in Figure 4.12. Table 4.1 shows example instantiated properties for the Action "Create bookmark".

Table 4.1 Instantiated Properties for the Action "Create bookmark" of the Storyboard "Add Bookmark"

OWL Individual	Property	OWL Individual
Create_bookmark	is_source_of	FROM__Create_bookmark__TO__Add_tag
FROM__Create_bookmark__TO__Add_tag	has_source	Create_bookmark
Create_bookmark	is_source_of	FROM__Create_bookmark__TO__EndNode
FROM__ Create_bookmark__TO__EndNode	has_source	Create_bookmark
Create_bookmark	is_target_of	FROM__StartNode__TO__Create_bookmark
FROM__StartNode__TO__Create_bookmark	has_target	Create_bookmark
Create_bookmark	activity_has_action	provide
Provide	is_action_of_activity	Create_bookmark
Create_bookmark	activity_has_object	bookmark_URL
bookmark_URL	is_object_of_activity	Create_bookmark
Create_bookmark	activity_has_property	Bookmark_Name
Bookmark_Name	is_property_of_activity	Create_bookmark

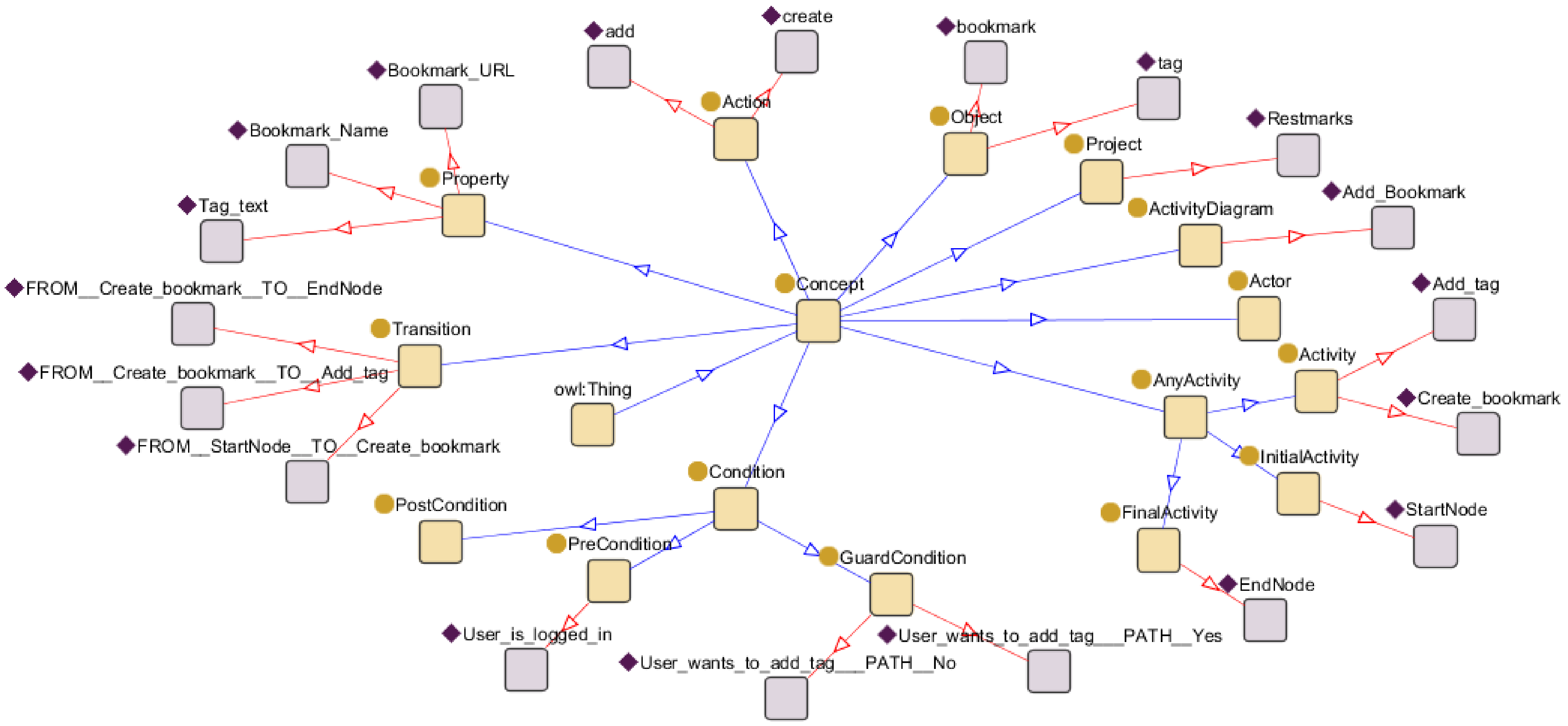


Figure 4.12 Example Ontology Instantiation of the Storyboard "Add Bookmark" of Project Restmarks

One can spot several differences between the instantiated ontology of Figure 4.12 and the one of Figure 3.3. The storyboard instantiation is a better fit to the RESTful paradigm. Properties are now easily identifiable since they are explicitly declared in storyboards. Therefore, actions are also correctly identified since they are also storyboard elements. Additionally, the CRUD verb type of `Action` is also declared in storyboard diagrams.

Finally, note that the ontology instantiation shown in Figure 4.12 is fully reversible. The owl file has a one-to-one mapping to the .sbd file of Figure 4.9. The ontology provides the functionality for retrieving the elements, so the `auth.storyboards:StoryboardDiagram` XMI element of the .sbd file is created. After that, Storyboard Creator allows generating the `notation:Diagram` part of the .sbd to also make it viewable in the graphical editor.

4.3.3.2 Example Instantiation for a software project

Upon providing an example for a specific storyboard diagram, it is straightforward to expand it to cover all the diagrams of a software project. In certain scenarios, the developer could provide the whole dynamic view of his/her project in the form of storyboards. Of course, this may not always be possible, however we shall provide an example including several dynamic scenarios in the form of storyboards to present a clear dynamic view for a project.

For project Restmarks [2], we are able to define 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 logins 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.

Note that these scenarios may involve several conditions, e.g. the user has to be logged in to delete a bookmark. Additionally, the project is complex enough since it also has nested storyboards, e.g. scenario 7 involves logging in – scenario 4. In the context of this deliverable, we collected these storyboards for project Restmarks [2] and instantiated the dynamic ontology. The instantiation is shown in Figure 4.13.

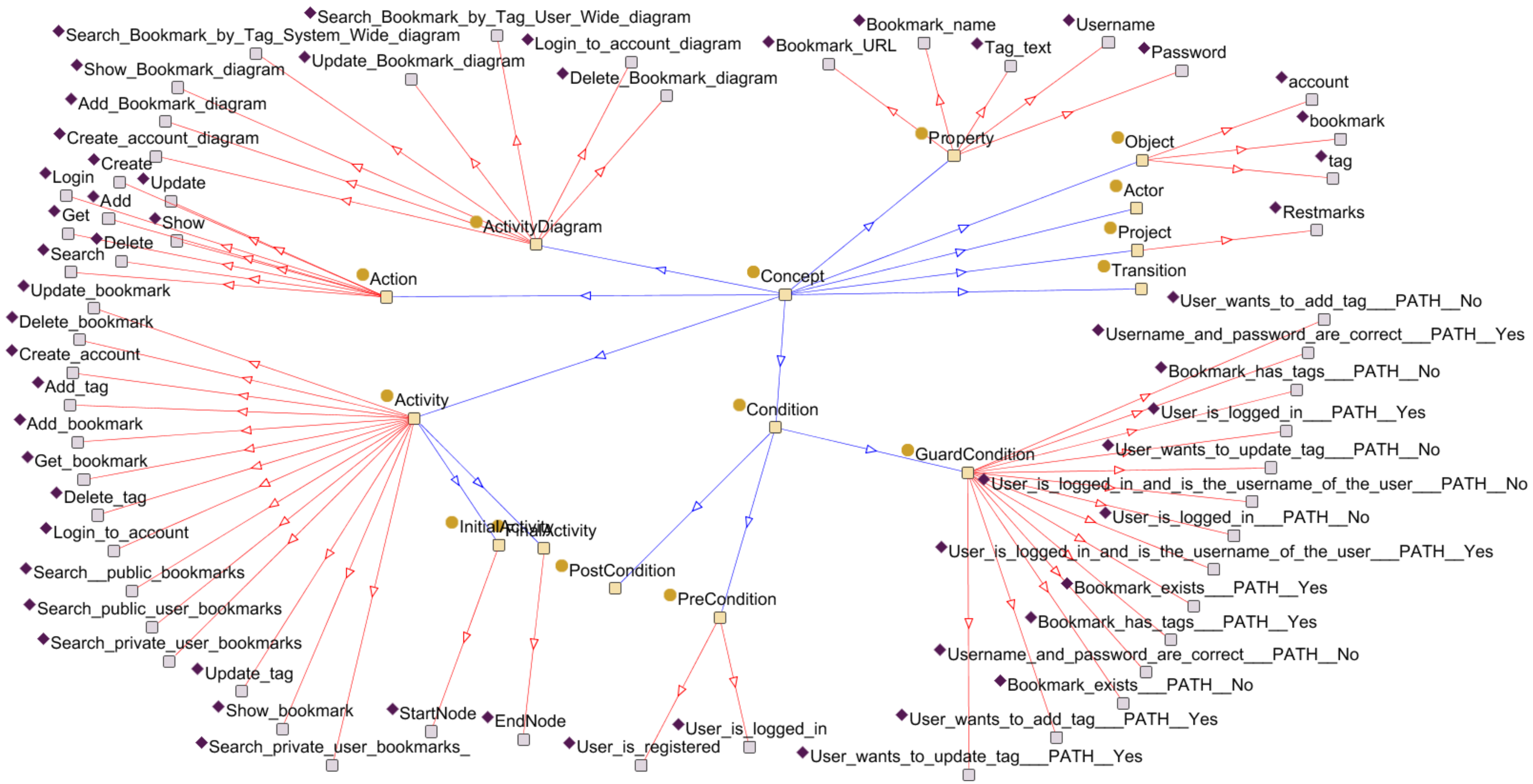

Search_Bookmark_by_Tag_User_Wide_diagram
Search_Bookmark_by_Tag_System_Wide_diagram
Login_to_account_diagram
Show_Bookmark_diagram
Update_Bookmark_diagram
Delete_Bookmark_diagram
Add_Bookmark_diagram
Create_account_diagram
Create
Login
Update
Add
Get
Show
Delete
Search
Update_bookmark
ActivityDiagram
Action
Bookmark_name
Username
Bookmark_URL
Tag_text
Password
account
Property
bookmark
Object
tag
Actor
Restmarks
Project
Transition
Concept
User_wants_to_add_tag___PATH__No
Username_and_password_are_correct___PATH__Yes
Delete_bookmark
Create_account
Add_tag
Add_bookmark
Get_bookmark
Delete_tag
Login_to_account
Search__public_bookmarks
Search_public_user_bookmarks
Search_private_user_bookmarks
Update_tag
Show_bookmark
Search_private_user_bookmarks_
Activity
InitialActivity
FinalActivity
StartNode
EndNode
PostCondition
PreCondition
User_is_registered
User_is_logged_in
Condition
GuardCondition
Bookmark_has_tags___PATH__No
User_is_logged_in___PATH__Yes
User_wants_to_update_tag___PATH__No
User_is_logged_in_and_is_the_username_of_the_user___PATH__No
User_is_logged_in___PATH__No
User_is_logged_in_and_is_the_username_of_the_user___PATH__Yes
Bookmark_exists___PATH__Yes
Bookmark_has_tags___PATH__Yes
Username_and_password_are_correct___PATH__No
Bookmark_exists___PATH__No
User_wants_to_add_tag___PATH__Yes
User_wants_to_update_tag___PATH__Yes

Figure 4.13 Example Ontology Instantiation of the Storyboards of Project Restmarks

The instantiation shown in Figure 4.13 may seem rather complex, however it is in fact not any more complex than the single-diagram example (Figure 4.12). Any project actually has several instances of `Activity` that are further analysed to `Action` and `Object`. Along with the instances of `Condition`, `Transition` (not shown in Figure 4.13 to avoid cluttering the Figure), and `Property`, the elements of the dynamic view of Restmarks are complete. Finally, using the `diagram_has/is_of_diagram` properties, one can easily find the diagram that describes any instance of the ontology.

5 Aggregated Ontology of Software Projects

This Section concerns the design of an ontology for storing information derived from the static and dynamic views of software projects. This ontology is an in-between stage between the work performed in WP3 for multimodal requirements extraction and the rest of S-CASE. In specific, both the MDE components and the components for finding functionally equivalent web services are connected to this ontology.

In the following subsections, we provide an overview of the ontology, and explain how it can be instantiated using the stored ontology data for the static and the dynamic view of the system. An example instantiation of the ontology is provided, and an API for retrieving information from the ontology is described, and tested using a RESTful representation.

5.1 Ontology Overview

The elements of this ontology actually form an initial version of the *Computationally Independent Model (CIM)* of the software project. 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. Finally, note that S-CASE not only allows forming a software prototype using MDE, but it also involves finding and exploiting web service resources, external to the main system. These are also handled by the ontology.

5.1.1 Ontology Class Hierarchy

The class hierarchy of the aggregated ontology is shown in Figure 5.1. Anything entered in the ontology is a `Concept`. Instances of `Concept` are further divided in four main classes: `Project`, `Requirement`, `ActivityDiagram`, and `Element`.

`Project` refers to the software project instantiated in the ontology. Classes `Requirement` and `ActivityDiagram` are used to hold the corresponding requirements and diagrams of the static and the dynamic ontology respectively. Note that the instances of these two classes are also used in order to keep track of the source of each element in the ontology. In other words, they can be used as connectors of the aggregated ontology to the static and dynamic ontologies.

Any other `Concept` of the ontology is an `Element` of the software project. Instances of type `Element` are further divided into the following subclasses:

- `Activity`: an activity of the system, e.g. “Create bookmark”
- `Condition`: a condition that has to be met for an activity to be executed, e.g. “The user must be logged in”
- `Resource`: a resource, which is the building block of any RESTful system. Examples for Restmarks include “bookmark” or “tag”.
- `Action`: a CRUD action, which is performed on a resource, as described by the corresponding activity. For example, the activity “Create bookmark” implies an action “create” on the resource “bookmark”.
- `Property`: a parameter required for a specific activity, e.g. the parameter “bookmark name” may be required for the activity “Create bookmark”.

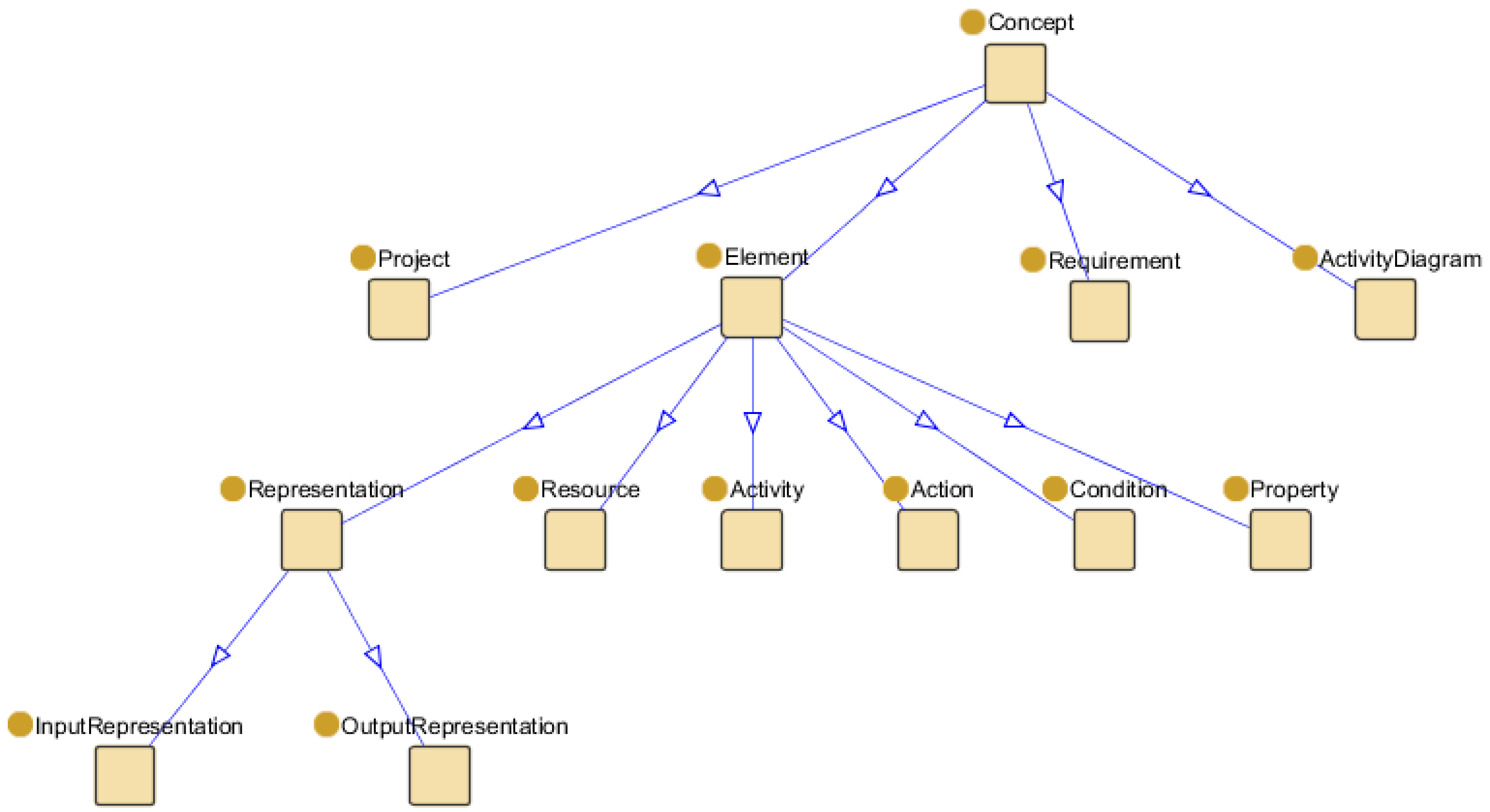


Figure 5.1 Aggregated Ontology Class Hierarchy

- `Representation`: a formal "manual sheet" required for calling external web services to acquire a specific "external" resource. Given, e.g., a resource "wordmap" that is given by an external web service (e.g. "Wordmap Unlimited"), the instance of this class holds two representations for input and output. These representations would belong to the following subclasses:
    - `InputRepresentation`: that holds information about how to call the external web service, e.g. via a GET or a POST? Using what arguments?
    - `OutputRepresentation`: that explains the response of the web service, e.g. does it return JSON or XML?

5.1.2 Ontology Properties

We distinguish among high-level and low-level properties. The former refer to properties defining interactions between classes `Project`, `Requirement`, `ActivityDiagram`, `Element`, whereas the latter refer to interactions among `Element` instances.

5.1.2.1 High-Level Ontology Properties

We define the high-level properties shown in Table 5.1.

Table 5.1 High-level 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`

The high-level properties ensure that the aggregated ontology covers all the requirements and diagrams of the static and the dynamic view of software projects. Additionally, the two properties `contains_element` and `element_is_contained_in` ensure that any element of the diagram is traceable in the other two ontologies. Given, e.g., the activity "Create account", we may trace it back to the corresponding instance of the static ontology (of deliverable 3.1) and find out it has been described by functional requirement FR1.

5.1.2.2 Low-Level Ontology Properties

The low-level properties are very important since they 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 CIM of the project. The low-level properties of the aggregated ontology are shown in Table 5.2.

Table 5.2 Low-level Properties of the Aggregated Ontology

OWL Class	Property	OWL Class
Resource	has_activity	Activity
Activity	is_activity_of	Resource
Resource	has_property	Property
Property	is_property_of	Resource
Resource	has_representation	Representation
Representation	is_representation_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

As shown in Table 5.2, the relations of the ontology classes are formed around two main subclasses, `Resource` and `Activity`. This is quite expected since these two elements form the basis of a RESTful system. Resources and the respective activities are prototyped by the S-CASE MDE engine. Any system `Resource` may be connected to instances of type `Property` and `Activity`, using the properties `has_property/is_property_of` and `has_activity/is_activity_of` respectively. The `Representation` of each `Resource` must also be connected to it (via the properties `has_representation/ is_representation_of`. Finally, 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 `has_next_activity/ has_previous_activity`. These low-level properties are also visualized in Figure 5.2, where it is clear that Resource and Activity have central roles in the aggregated ontology.

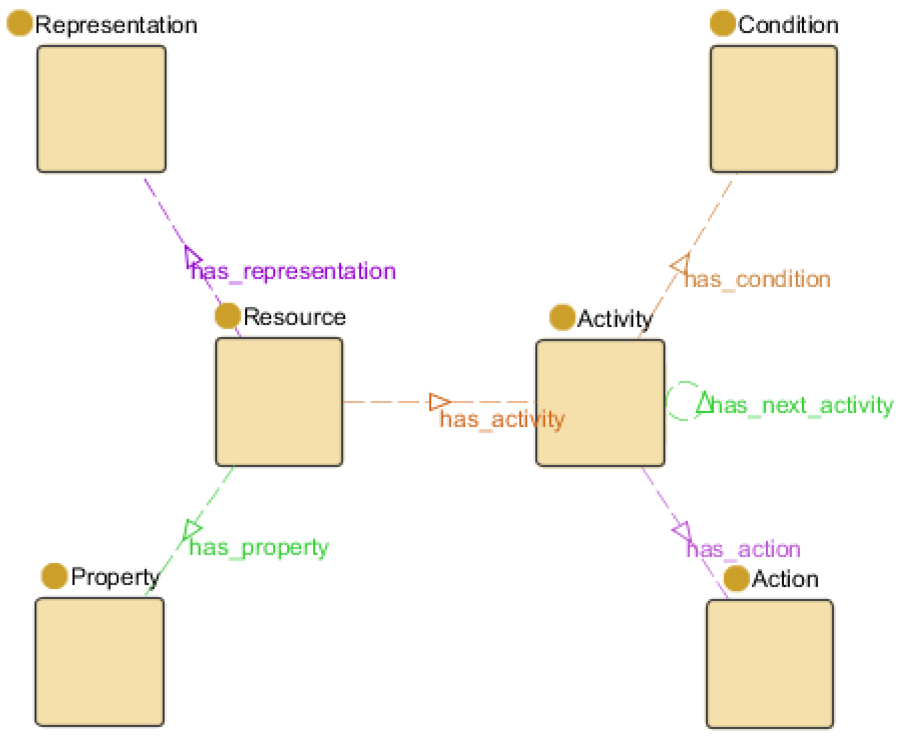


Figure 5.2 Low-level Aggregated Ontology Properties

5.2 Ontology Linking

As already noted in this Section, the aggregated ontology is instantiated using the information provided by the static and dynamic ontologies of software projects. In the following subsections we provide a mapping for instantiating the ontology, while its instantiation is further illustrated using an example.

5.2.1 Linking the Static and Dynamic Ontologies

The static ontology of deliverable 3.1 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 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 the static and the dynamic ontologies to the aggregated ontology is shown in Figure 5.3. As shown in this Figure, `Requirement` and `ActivityDiagram` are simply propagated to the aggregated ontology, while `Project` is used to ensure that the two ontology instantiation refer to the same project.

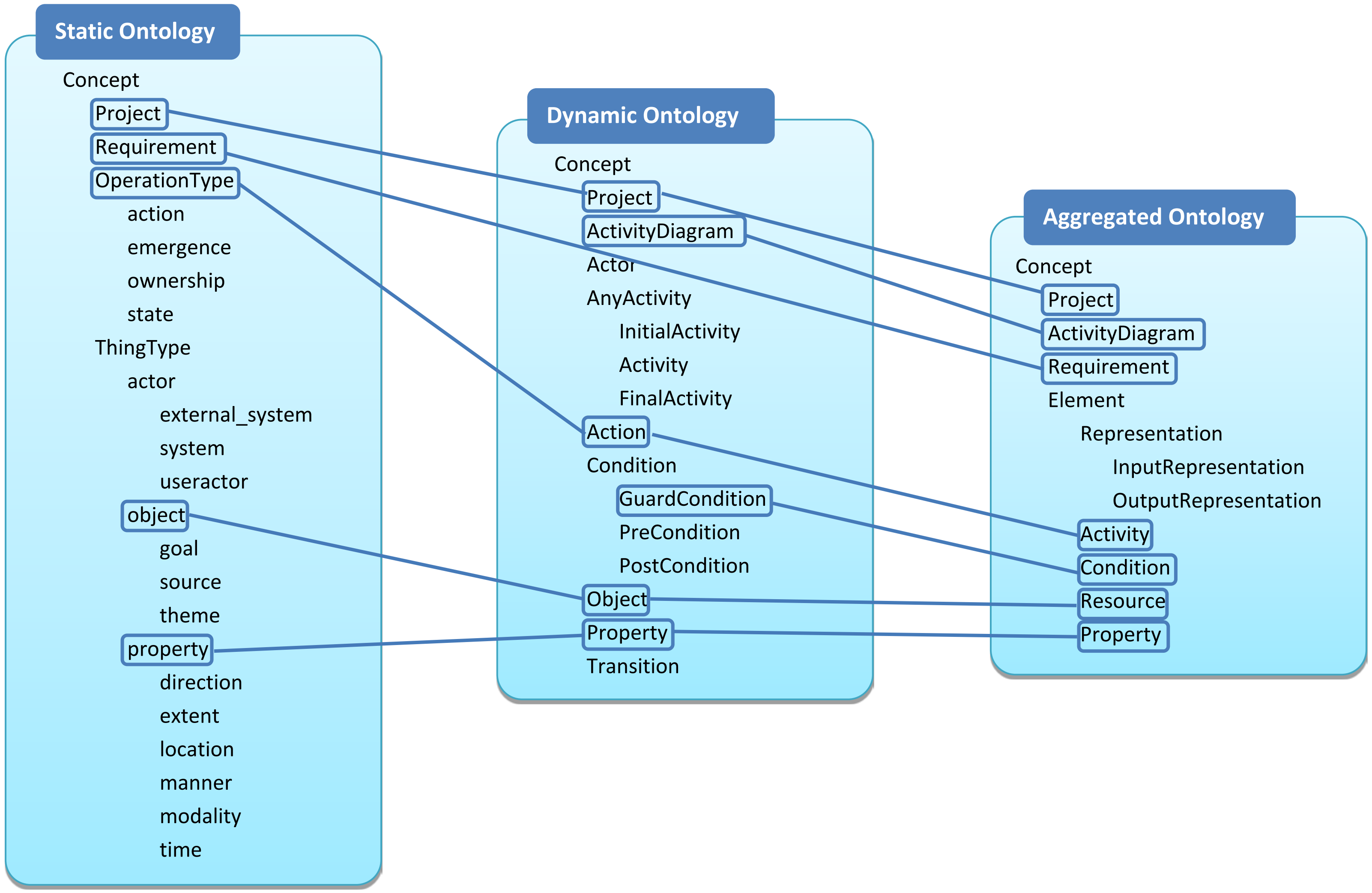


Figure 5.3 Mapping from the Static and Dynamic Ontologies to the Aggregated Ontology

Concerning the remaining classes of the aggregated ontology, these 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 of OWL classes from the static and dynamic ontologies to the aggregated ontology is shown in Table 5.3.

Table 5.3 Mapping of OWL classes from the Static and Dynamic Ontologies to the 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 mentioned above, properties are mapped after the respective classes have been mapped. The respective mapping for OWL properties is shown in Table 5.4.

Table 5.4 Mapping of OWL properties from the Static and Dynamic Ontologies to the Aggregated Ontology

OWL Property of Static Ontology	OWL Property of Dynamic Ontology	OWL Property of Aggregated Ontology
`project_has_requirement`	-	`has_requirement`
`is_of_project`	-	`is_requirement_of`
-	`project_has_diagram`	`has_activity_diagram`
-	`is_diagram_of_project`	`is_activity_diagram_of`
`requirement_consists_of`	`diagram_has`	`contains_element`
`consist_requirement`	`is_of_diagram`	`element_is_contained_in`
`receives_action`	`is_object_of_activity`	`has_activity`

acts_on	activity_has_object	is_activity_of
has_property	activity_has_property*	has_property
is_property_of	is_property_of_activity*	is_property_of
-	activity_has_action	has_action
-	is_action_of_activity	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 they are 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 has_condition/is_condition_of, which are instantiated using the instances of GuardCondition of the preceding Transition.

Finally, the two properties that refer to the connection of Project to Element, has_element and is_element_of, are derived from contains_element and element_is_contained_in at the time of instantiating the ontology. Furthermore, the has_representation/is_representation_of properties are instantiated for external web services, thus they are not mapped to the static and dynamic ontologies.

5.2.2 Example Instantiation

Upon presenting how the two ontologies are connected to form the aggregated ontology of software projects, in this section we further illustrate this connection using an example. For our example, we use project Restmarks [2]. The static ontology for project Restmarks has been provided in deliverable 3.1 of this work package (Figure 7 of Section 2 of D3.1), while the dynamic ontology is shown in Figure 4.13 of Section 4 of this deliverable[7]. Upon applying the mapping of subsection 5.2.1, the aggregated ontology instance is shown in Figure 5.4.

[7] Note that the static and the dynamic ontologies rely each on a single type of input, functional requirements and storyboards respectively. We could also have other modes, e.g. use case diagrams, activity diagrams. In any case, the static and the dynamic view are sufficiently covered for this software project using these modes.

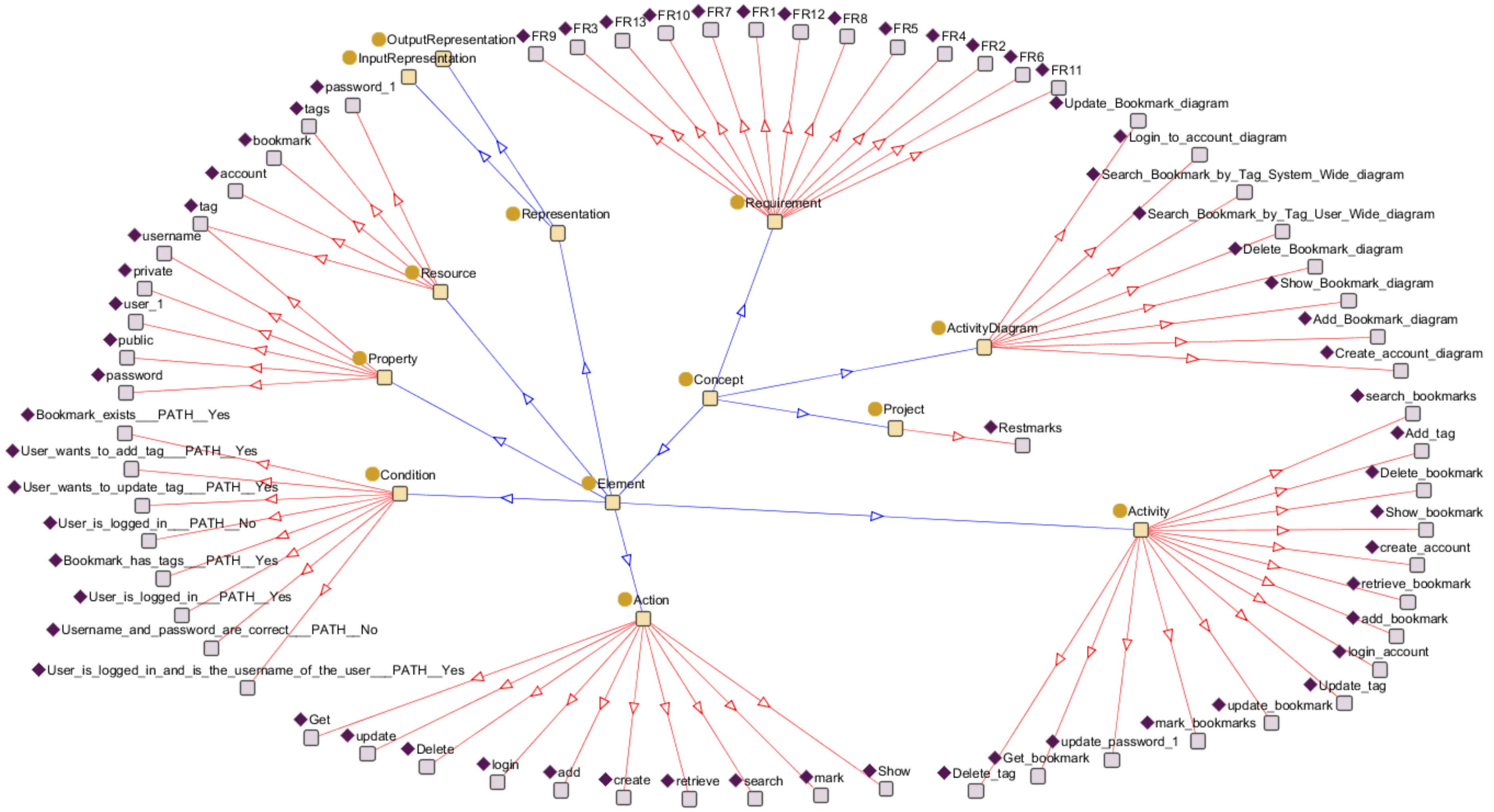


Figure 5.4 Example Instantiation of the Aggregated Ontology for project Restmarks

As shown in Figure 5.4, the aggregated ontology for Restmarks is a representation that could closely resemble its CIM. Although not covered in this deliverable, RESTful architectures imply that the main elements of the CIM are resources, actions for creating, reading, updating, and deleting them, as well as conditions and properties for these actions. Thus, in the case of Restmarks, we can see that several resources, such as “bookmark” or “tag”, have been correctly identified. Additionally, the properties of the instances of the ontology are also correctly identified. Some of the related instances for resource “bookmark” are shown in Table 5.5.

Table 5.5 Related Instances for the resource “bookmark” of Restmarks

OWL Class	**OWL Property**
`Project`	`Restmarks`
`Requirement`	`FR4, FR5, FR6, FR7, FR8, FR9, FR10, FR11, FR12, FR13`
`ActivityDiagram`	`Add_Bookmark_diagram, Delete_Bookmark_diagram, Show_Bookmark_diagram, Update_Bookmark_diagram, Search_Bookmark_by_Tag_System_Wide_diagram, Search_Bookmark_by_Tag_User_Wide_diagram`
`Property`	`private, public, tag`
`Activity`	`Add_bookmark, Delete_bookmark, Get_bookmark, Show_bookmark, Update_bookmark, retrieve_bookmark, search_bookmarks, mark_bookmarks`

At first, the related instances of “bookmark” clearly illustrate how this resource has been detected. The relevant functional requirements and diagrams are all marked. Additionally, the related instances of `Activity` contain all possible actions of “bookmark”. Since they are all connected to `Action`, we can easily deduce which CRUD verb has to be used for each activity.

5.3 Ontology API

As noted in the introduction of this Section, the aggregated ontology has to communicate to both the MDE components and the components for finding functionally equivalent web services. Additionally, it has to be instantiated by the static and dynamic ontologies as shown in the previous subsection. Consequently, we designed a comprehensive API to handle all I/O operations on the ontology. In this subsection, we describe this API.

The API is instantiated given the filename of the ontology and the project name to be added or retrieved. The constructor is shown in Table 5.6. Note also that upon using the API, one has to call the function *close* in order to write it back to disk.

Table 5.6 Constructor of the API for the Aggregated Ontology

Method: LinkedOntologyAPI		
	Return type	(constructor)
	Parameters	String filename, String source, String projectName
	Description	Initializes the connection of this API with the ontology.

Table 5.7 contains API functions used for adding elements to the ontology. At first, it allows instantiating the high-level OWL classes, such as `Requirement` and `ActivityDiagram` (`Project` is already handled in the constructor). After that, any element added in the ontology (e.g. a `Resource`) is connected to the respective requirement or diagram and the project using the function *connectRequirementToElement*. Note also that several of these functions add an element and connect it also to other elements. For example, function *addActivityToResource* does not only add an `Activity` to the ontology but also connects it to the corresponding `Resource`. This is quite important since it ensures that no `Activity` is left without a `Resource`, thus the API enforces some simple rules for the ontology.

Table 5.7 Input Functions of the API for the Aggregated Ontology

Method: addActionToActivity		
	Return type	void
	Parameters	String activityName, String actionName
	Description	Adds an action to a specific activity of the ontology.
Method: addActivityDiagram		
	Return type	void
	Parameters	String activityDiagramName
	Description	Adds an activity diagram and connects it to the project.
Method: addActivityToResource		
	Return type	void
	Parameters	String resourceName, String activityName
	Description	Adds an activity to a specific resource of the ontology. The type of the activity is set to “Other”.

Method: addActivityToResource		
	Return type	void
	Parameters	String resourceName, String activityName, String activitytype
	Description	Adds an activity to a specific resource of the ontology. Overloaded function to include the type of the activity.
Method: addConditionToActivity		
	Return type	void
	Parameters	String activityName, String conditionName
	Description	Adds a condition to a specific activity of the ontology.
Method: addInputRepresentationToResource		
	Return type	void
	Parameters	String resourceName, String inputRepresentation
	Description	Adds an input representation to a specific resource.
Method: addNextActivityToActivity		
	Return type	void
	Parameters	String activityName, String nextActivityName
	Description	Adds a forthcoming activity to a specific activity of the ontology.
Method: addOutputRepresentationToResource		
	Return type	void
	Parameters	String resourceName, String outputRepresentation
	Description	Adds an output representation to a specific resource.
Method: addPropertyToResource		
	Return type	void
	Parameters	String resourceName, String propertyName
	Description	Adds a property to a specific resource of the ontology.

Method: addRequirement		
	Return type	void
	Parameters	String requirementName
	Description	Adds a requirement in the ontology and connects it to the project.
Method: addResource		
	Return type	void
	Parameters	String resourceName
	Description	Adds a resource to the ontology.
Method: connectActivityDiagramToElement		
	Return type	void
	Parameters	String activityDiagramName, String elementName
	Description	Connects an activity diagram to an element of the ontology.
Method: connectRequirementToElement		
	Return type	void
	Parameters	String requirementName, String elementName
	Description	Connects a requirement to an element of the ontology.

The respective output functions are shown in Table 5.8. Note that the API, and especially the output part of it, is obviously prone to changes according to the specifications of the S-CASE components. Thus, one may notice that this API is not exhaustive, i.e. not all possible functions are covered. However, its main structure is very close to the required functionality for creating a CIM for a software project.

Since the main element of a CIM is a resource, the API relies on iterating over resources using the function *getResources*. After that, given a specific resource, one can easily access its activities (*getActivitiesOfResource*), its properties (*getPropertiesOfResource*), and its representations (*getInputRepresentationOfResource*/ *getOutputRepresentationOfResource*). Given an activity, we can also proceed in finding its action (*getActionOfActivity*), its type (*getActivityTypeOfActivity*), and any forthcoming activities (getNextActivitiesOfActivity). Additionally, since activity is also a central element of the ontology, its connection to resource is bidirectional, thus the corresponding resource can be found using the function *getResourceOfActivity*. Finally, note that since these functions return *String* representations, object representation is comprehensive, while any implementation details are hidden.

Table 5.8 Output Functions of the API for the Aggregated Ontology

Method: getActionOfActivity		
	Return type	String
	Parameters	String activityName
	Description	Returns the action of a specific activity.
Method: getActivitiesOfResource		
	Return type	ArrayList<String>
	Parameters	String resourceName
	Description	Returns the activities of a specific resource.
Method: getActivityTypeOfActivity		
	Return type	String
	Parameters	String activityName
	Description	Returns the activity type of a specific activity.
Method: getInputRepresentationOfResource		
	Return type	String
	Parameters	String resourceName
	Description	Returns the input representation of a specific resource.
Method: getNextActivitiesOfActivity		
	Return type	ArrayList<String>
	Parameters	String activityName
	Description	Returns the forthcoming activities of a specific activity.
Method: getOutputRepresentationOfResource		
	Return type	String
	Parameters	String resourceName
	Description	Returns the output representation of a specific resource.

Method: getPropertiesOfResource		
	Return type	ArrayList<String>
	Parameters	String resourceName
	Description	Returns the properties of a specific resource.
Method: getResourceOfActivity		
	Return type	String
	Parameters	String activityName
	Description	Returns the resource of a specific activity.
Method: getResources		
	Return type	ArrayList<String>
	Parameters	-
	Description	Returns the resources of the ontology for the current project.

5.4 Ontology Instantiation using the RESTful API Modeling Language

The aggregated ontology defined in this Section provides a unified view of software projects. As such, the instantiation of the ontology can be performed using various sources. In specific, given the ontology API defined in the previous subsection, one can develop tools for instantiating the ontology using any source he/she desires. In this subsection, we provide an example of instantiating the ontology using a RESTful API Modeling Language (RAML) representation.

RAML [18] is a language used to describe RESTful APIs, based on YAML and supporting also providing schemas in the form of JSON. Since this representation is the current state-of-the-art in defining RESTful APIs, we decided to provide a parser for loading a project in this form in the ontology. Note, however, that RAML is a language used to define a fully-determined API, thus the information included in this representation may be extended beyond the more high-level structure of the ontology. For example, response codes or schemata are not supported by the ontology which is expected since they are not defined in the level of requirements. Therefore, the main scenario handled in this subsection involves a developer that has devised a draft RAML representation and wishes to use this instead of the data from the two ontologies, or even use it along with the data of the static and/or the dynamic ontology. In other words, S-CASE allows also using this representation along with the other ones used in WP3 (functional requirements, UML diagrams, etc.), in order to account for a more detailed view of the system.

An example RAML representation for project Restmarks is shown in Figure 5.5.

```
#%RAML 0.8

title: Restmarks API
baseUri: http://www.scasefp7.eu/restmarks/
/account:
  get:
    description: get the list of all accounts
  post:
    description: create a new account
    queryParameters:
      accountId:
        description: the id of the account to be created
        type: integer
        required: true
  /{accountId}:
    get:
      description: get specific account
      responses:
        200:
          body:
            application/json:
              schema: |
                {
                  "accountId": { "type": "integer" },
                  "accountName": { "type": "string" }
                }
    put:
      description: update specific account
      queryParameters:
        accountName:
          description: the name of the account to be updated
          type: string
          required: true
      responses:
        200:
          body:
            application/json:
              schema: |
                {
                  "accountId": { "type": "integer" },
                  "accountName": { "type": "string" }
                }
    delete:
      description: Delete an account
    /bookmark:
      get:
        description: get the list of all bookmarks
      post:
        description: create a new bookmark
        queryParameters:
          bookmarkId:
            description: the id of the bookmark to be created
            type: integer
            required: true
          bookmarkName:
            description: the name of the bookmark to be created
            type: string
            required: true
      /{bookmarkId}:
        get:
          description: get specific bookmark
          responses:
            200:
              body:
                application/json:
                  schema: |
                    {
                      "bookmarkId": { "type": "integer" },
                      "bookmarkName": { "type": "string" }
                  }
        put:
          description: update specific bookmark
          queryParameters:
            bookmarkName:
              description: the name of the bookmark to be updated
              type: string
              required: true
          responses:
            200:
              body:
                application/json:
                  schema: |
                    {
                      "bookmarkId": { "type": "integer" },
                      "bookmarkName": { "type": "string" }
                    }
        delete:
          description: Delete a bookmark
        /tag:
          get:
            description: get the list of all tags for this bookmark
          post:
            description: create a new tag for this bookmark
            queryParameters:
              tagId:
                description: the id of the tag to be created
                type: integer
                required: true
              tagName:
                description: the name of the tag to be created
                type: string
                required: true
          /{tagId}:
            get:
              description: get specific tag
              responses:
                200:
                  body:
                    application/json:
                      schema: |
                        {
                          "tagId": { "type": "integer" },
                          "tagName": { "type": "string" }
                        }
            put:
              description: update specific tag
              queryParameters:
                tagName:
                  description: the name of the tag to be updated
                  type: string
                  required: true
              responses:
                200:
                  body:
                    application/json:
                      schema: |
                        {
                          "tagId": { "type": "integer" },
                          "tagName": { "type": "string" }
                        }
            delete:
              description: Delete a tag
/tagsearch:
  get:
    description: search for a bookmark given a specific tag
    queryParameters:
      tagName:
        description: the name of the tag to search for
        type: string
        required: true
      systemWide:
        description: boolean denoting whether the search must be system wide or
user wide
        type: boolean
        required: true
    responses:
      200:
        body:
          application/json:
            schema: |
              {
                "bookmarkIds": { "type": "list" }
              }
```

Figure 5.5 Example RAML representation of project Restmarks

Resources in RAML are defined using the `/` symbol in front of each resource object. As shown in Figure 5.5, the RAML representation of Restmarks involves the resources `account`, `{accountId}`, `bookmark`, `{bookmarkId}`, `tag`, `{tagId}`, and `tagsearch`. The hierarchy of these resources is defined by using the indentation supported by the YAML notation. Thus, `{accountId}` is a subresource of `account`, `bookmark` is a subresource of `{accountId}`, etc. Since `{accountId}` actually defined the action of accessing a specific `account` and the same holds for any resources surrounded by brackets (`{bookmarkId}`, `{tagId}`), we can instantiate the `Resource` class of the ontology with the resources `account`, `bookmark`, `tag`, and `tagsearch`.

Concerning the actions on resources, these are defined in the next level of indentation, i.e. inside of each resource. So, for example, we can see that resource `account` has the actions `get` and `post`. Note that actions are defined in implementation-level, thus they are HTTP verbs. In our case, they are translated in the ontology in the same verbs in the instance `Action`, while the instances of the `Activity` class are also defined using the format `Activity_Resource`, e.g. for the `Resource` `bookmark` and the `Action` `post`, we also define the `Activity` `post_bookmark`. Given, however, that resources may have also bracketed counterparts referring to an individual resource (e.g. `bookmark` has `{bookmarkId}`), there are cases where a resource may have more than one `get` verbs. In this case, we conventionally define the super-resource `get` as `list`. E.g., for `bookmark`, we have an action `list` that lists the user’s bookmarks and an action `get` that retrieves a specific bookmark.

The ontology class `Property` is instantiated using the `queryParameters` RAML element. For example, resource `tag` has the parameters `tagId` and `tagName`, as determined by the `queryParameters` of the verbs `post` and (`{tagId}/`)`put`.

Finally, note that several elements of the RAML file are not included in the ontology. For instance, URIs, schemata, variable types, etc. are not supported by the ontology classes. However, the mapping described in the previous paragraphs allows instantiating the ontology using all the main elements of the architecture of a RESTful service, including resources, actions, and properties. The parser can be extended if required to involve more information in the form of OWL comments, however the main functionality already supports our scope for instantiating the ontology.

Given the RAML of project Restmarks, the corresponding ontology instantiation of our RAML parser is shown in Figure 5.6.

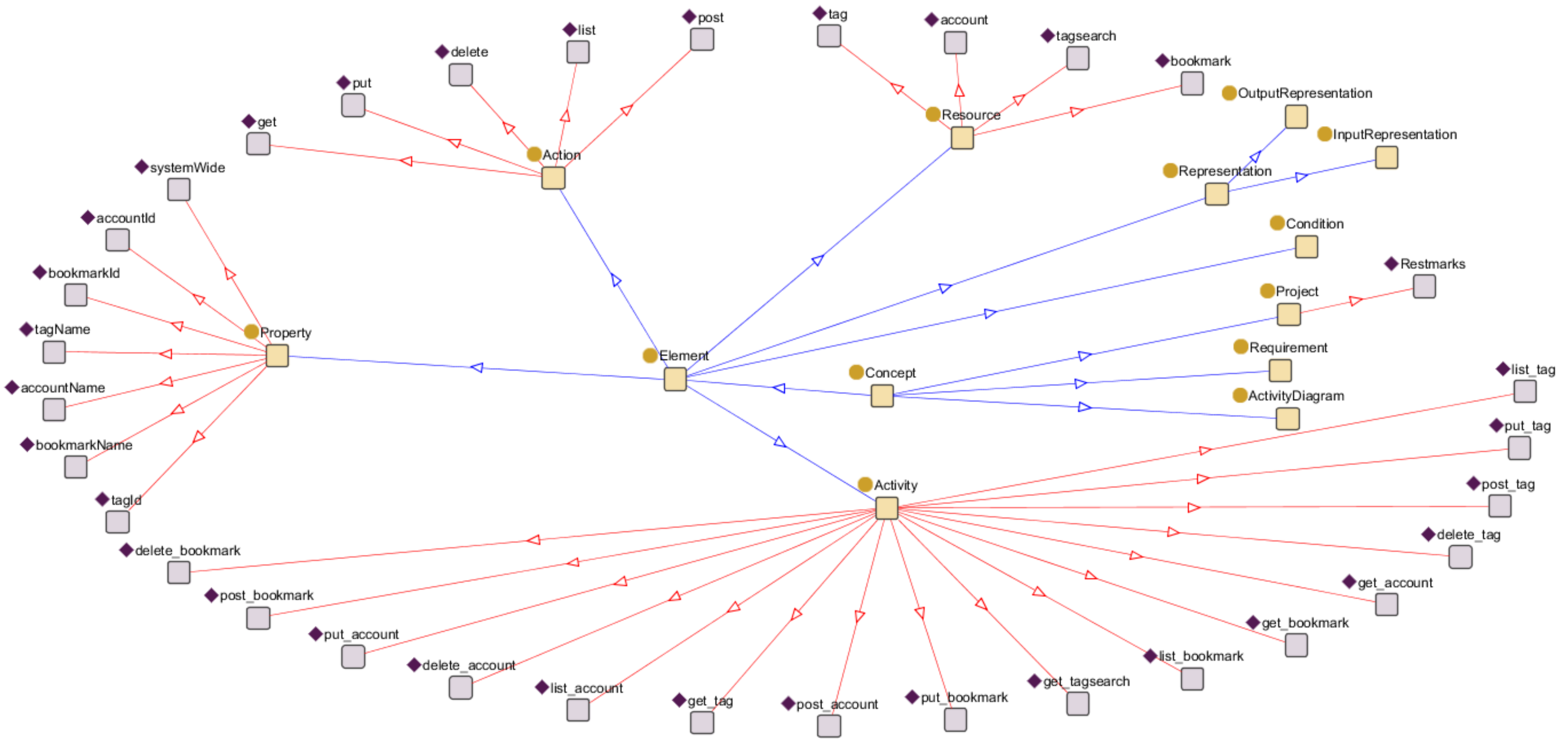


Figure 5.6 Example Instantiation of the Aggregated Ontology for the RAML representation of project Restmarks

As shown in Figure 5.6, the ontology indeed includes the main elements of the RAML representation of Restmarks. It is interesting to note that the activities `list_account`, `list_bookmark` and `list_tag` are generated by the corresponding `get` operations of the RAML. Additionally, note that `tagsearch` is recognized as another resource, which is actually quite convenient for designing the service. By contrast, in the ontology instantiation of Figure 5.4, the corresponding resource is shown as an action `search` that is performed on the resource `tag`. Since when creating the CIM we may have to make this translation, the ontology that is instantiated from the RAML is actually more convenient in this case.

Finally, the properties of the instances of the ontology are also correctly instantiated. Some of the related instances for resource "bookmark" are shown in Table 5.5.

Table 5.9 Related Instances for the resource "bookmark" of the RAML representation of Restmarks

OWL Class	**OWL Property**
`Project`	`Restmarks`
`Property`	`bookmarkId, bookmarkName`
`Activity`	`delete_bookmark, get_bookmark, list_bookmark, post_bookmark, put_bookmark`

6 Conclusions

The work discussed in this deliverable summarizes our progress on Task 3.2 of WP3 of S-CASE. In the context of the S-CASE architecture, this deliverable provides an initial viewpoint for the multimodal information processing purpose of WP3. In specific, we introduce an ontology that stores the dynamic view of software projects and also design an aggregated ontology that provides a unified view of software projects as collections of artefacts.

Similar to the static ontology designed in Task 3.1, this deliverable describes the design of an ontology capable of storing dynamic artefacts of projects. The input for this ontology is given in the form of storyboards and activity diagrams, although it may easily be extended to other types of dynamic flow representations. The ontology describes effectively the main elements as well as the flow of data in a system.

As for the diagrams of the dynamic view of the system, we have concluded that activity diagrams are sometimes verbose and generally do not fit perfectly the RESTful paradigm. Consequently, we have designed storyboards as a new type of diagram that is more effective in describing resources, RESTful actions, and dynamic flows of systems. Furthermore, we have designed and implemented a diagram editor for these types of storyboards as a plugin of the Eclipse IDE.

Finally, a unified view of the static and dynamic concepts of a software system was defined using data from the ontologies for the static and dynamic views of the system. The aggregated ontology functions as a REST-oriented representation while also ensuring that the traceability with respect to the two ontologies is achieved. The API of the aggregated ontology can be used for instantiating it using several representations, such as the RAML one shown in this deliverable. Additionally, the API shall prove useful for creating the first version of the CIM for a software project.

References

[1] Object Management Group: OMG Unified Modeling Language (OMG UML), Infrastructure. Version 2.5 – Beta 2. OMG Document Number ptc/2013-09-05, 2013, available online: http://www.omg.org/spec/UML/2.5/Beta2/

[2] Project Restmarks, RESTAPPS, S-CASE Consortium, 2014.

[3] Cucumber: behaviour driven development with elegance and joy, 2014, available online: http://cukes.info/

[4] List of Unified Modeling Language tools, Wikipedia, 2014, available online: http://en.wikipedia.org/wiki/List_of_Unified_Modeling_Language_tools

[5] ArgoUML, 2014, available online: http://argouml.tigris.org/

[6] StarUML 2, A sophisticated software modeller, 2014, available online: http://staruml.io/

[7] Papyrus, Eclipse UML tool, 2014, available online: http://eclipse.org/papyrus/

[8] Modelio open source modeling environment, 2014, available online: http://www.modelio.org

[9] UML Tools, S-CASE wiki, 2014, available online: http://wiki.scasefp7.com/index.php/UML_tools

[10] Graphiti - a Graphical Tooling Infrastructure, 2014, available online: http://eclipse.org/graphiti/

[11] EuGENia, 2014, available online: http://eclipse.org/epsilon/doc/eugenia/

[12] Eclipse Modeling Framework Project, available online: http://www.eclipse.org/modeling/emf/

[13] Graphical Editing Framework, available online: http://www.eclipse.org/gef/

[14] Eclipse Graphical Modeling Framework / Tutorial / Part 1, available online: http://wiki.eclipse.org/Graphical_Modeling_Framework/Tutorial/Part_1

[15] Protégé, Stanford Center for Biomedical Informatics Research (BMIR), Stanford University School of Medicine, 2014, available online: http://protege.stanford.edu

[16] Storyboard Creator Technical Manual, S-CASE Consortium, 2014

[17] Storyboard Creator User Manual, S-CASE Consortium, 2014

[18] RAML - RESTful API Modeling Language, available online: http://raml.org/