MASTER THESIS

AUTOMATIC GENERATION OF GRAPHICAL DOMAIN ONTOLOGY EDITORS

C.F. Nijenhuis

FACULTY OF ELECTRICAL ENGINEERING, MATHEMATICS AND COMPUTER SCIENCE
SOFTWARE ENGINEERING

EXAMINATION COMMITTEE
Luís Ferreira Pires
Luiz Olavo Bonino da Silva Santos
Ivan Kurtev

DOCUMENT NUMBER
EWI/SE – 2011-002

UNIVERSITY OF TWENTE.

MARCH 2011

UNIVERSITY OF TWENTE.

Automatic generation of graphical domain ontology editors

Christiaan Frank Nijenhuis

Enschede, The Netherlands

March 2011

ABSTRACT

The field of Service-Oriented Computing has the vision that services represent distributed pieces of functionality. Combining services may result in new and more complex functionality. Platforms to help users find, select and invoke services are being built. These platforms provide users with tools to help them with these tasks. One of these platforms is the Context-Aware Service Platform.

This thesis proposes an architecture for automatic generation of tool support for domain specialists performing modeling tasks. The research has been done in the scope of the Context-Aware Service Platform. The proposed architecture provides an automatic way to generate domain ontology editors, based on a language described by an upper level ontology. The process involves translating upper level ontologies into metamodels, automatically generating editors from metamodels and keeping traces between the stages of the process. These are traces between the upper level ontologies and the metamodels resulting from the translation, and traces between newly generated languages and existing languages used by the generated domain ontology editors. A prototype tool has been developed and an evaluation of this prototype has been performed within this research.

The resulting domain ontology editors can be used by domain specialists of the Context-Aware Service Platform, providing them with the means to specify knowledge about domains. Service providers can annotate their services with domain-specific knowledge. This knowledge can be used by the platform to help service clients to find and invoke services.

TABLE OF CONTENTS

PREFACE

This thesis is the result of my Master of Science assignment, which I performed at the Software Engineering Group at the University of Twente. This assignment concludes the Software Engineering track in the Computer Science program.

I would like to thank my supervisors Luís Ferreira Pires, Luiz Olavo Bonino da Silva Santos and Ivan Kurtev for their help and guidance during this project. Also, I want to thank my fellow Master of Science students, who shared an office with me and who were always good for some laughs when needed.

Furthermore, I want to thank my parents, brother and sister for their stimulating support. They are always there for me when I need them and they have always believed in me. My friends also deserve a word of thanks for providing so much fun and pleasant distractions during the 7,5 great years that I spent in Enschede.

Especially and above all, I would like to express my great gratitude to my wonderful girlfriend and soon to be wife Lidia Ferrari, who always stands by me and encouraged me to complete this work.

Cheers,

Frank Nijenhuis

Enschede, The Netherlands
March 2011

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

API	Application Programming Interface
ATL	Atlas Transformation Language
CASP	Context-Aware Service Platform
DSL	Domain Specific Language
EMF	Eclipse Modeling Framework
EMF4SW	Eclipse Modeling for Semantic Web
GDSL	Goal-Based Domain Specification Language
GEF	Graphical Editing Framework
GMF	Graphical Modeling Framework
GPS	Global Positioning System
GSF	Goal-Based Service Framework
GSO	Goal-Based Service Ontology
HTML	HyperText Markup Language
OCL	Object Constraint Language
OMG	Object Management Group
MOF	Meta Object Facility
RDF	Resource Description Framework
SOC	Service-Oriented Computing
SWRL	Semantic Web Rule Language
UFO	Unified Foundational Ontology
ULO	Upper Level Ontology
UML	Unified Modeling Language
OWL	Web Ontology Language
XML	Extensible Markup Language

1 INTRODUCTION

This chapter presents the motivation, the objective, the approach and the structure of this thesis. The motivation is discussed in section 1.1. This is followed by the objective of our research, which is presented in section 1.2. Our approach to achieving our objective is explained in section 1.3 identifying the steps that were taken in this research project. This chapter ends by elaborating on the structure of this document in section 1.4.

1.1 MOTIVATION

The majority of the pages in the Web is in human readable format only. Software agents are not capable of understanding or processing this information [1]. In order for distributed applications on the Internet to have automatic data processing between software agents, semantics is needed. With semantics, agents can reason about data and process data without human interference.

The Web was originally formed around HTML. XML was introduced to define arbitrary domain and task specific extensions. After XML, RDF was introduced to represent machine-processable semantics of data by using simple data models [2]. These techniques were the first steps towards the Semantic Web.

The Semantic Web is a vision about a new form of web content that is meaningful to computers. It is not a separate Web, but builds on the Internet we know today. The Semantic Web will bring more structure to the data that is present in the Internet, giving content a well-defined meaning, enabling machines to process data without interference of people and improving cooperation between computers and people. The Semantic Web can only function when computers can perform automated reasoning. To achieve this, computer-understandable and structured collections of information and sets of inference rules are needed. Providing a language that can express both data and rules for reasoning about the data is a challenge of the Semantic Web. The next step in achieving this is to add logic to the Web, allowing to use rules to make inferences and answer questions [3].

An aspect that benefits from adding logic to the Web is context-awareness in software agents. When an agent is aware of the context of the provided

information, it can make (better) choices in reasoning. Dey and Abowd [4] discuss context-awareness. They define that a system is context-aware if it uses context to provide relevant information and/or services to the user, where relevancy depends on the user's tasks. For a system to achieve context-awareness it is thus important that it knows which information relates to context and whether the information is relevant or not. Therefore it needs to know which things are in the domain of the application and what the relationships are between these things. For example, a system controlling the conditions inside a greenhouse works on a totally different domain than a system looking for the nearest bus stop. In the first system, the temperature inside the greenhouse (measured by a thermometer) is important, whereas the current location of the any person (provided by Global Positioning System (GPS) sensors) is useless. In the second system, this would be the other way around. However, even if the systems know which domain is relevant, they would still not be able to reason about the information in the domain, since no semantics is defined.

Ontologies play a key role in expressing domain knowledge and semantics, which are needed for automated reasoning in software agents. The term Ontology comes from philosophy, in which it denotes the study of the kinds of things that exist. Aristotle described Ontology as "the science of being qua being". In computer science, the term ontology was used for the first time by Mealy, referring to the question of what exists [5]. In this area, ontologies are content theories about what sort of objects, relations between objects and properties of objects exist in a universe of discourse. However, an ontology is not just a representation vocabulary for an arbitrary domain. The terms in the vocabulary try to capture a conceptualization of real world objects, properties and relations. Translating the terms in the ontology to another language, will not change the conceptualization, and thus will not change the ontology [6].

Using a conceptualization of a domain, a software agent can reason about objects, relations and properties in this domain, since it knows how they are related. Ontologies also enable knowledge sharing. Software agents sharing the same vocabulary (and the underlying conceptualization) can communicate with each other about objects in the domain. It forms the basis for domain-specific knowledge representation languages [6].

The above described concepts play a major role in the field of Service-Oriented Computing (SOC). It is the vision of SOC that services represent

distributed pieces of functionality and that combining these pieces will result in new and more complex functionality. Ideally, these services are combined without human interference. This vision is, however, not yet reality, although some work towards its realization has been done. Platforms for supporting service provisioning have been built, e.g. the Context-Aware Service Platform (CASP) [7]. The main benefit of this platform is that it allows users to specify their service request in concepts that are close to the user's perception, instead of in technical terms. Figure 1.1 depicts the architectural design of the platform.

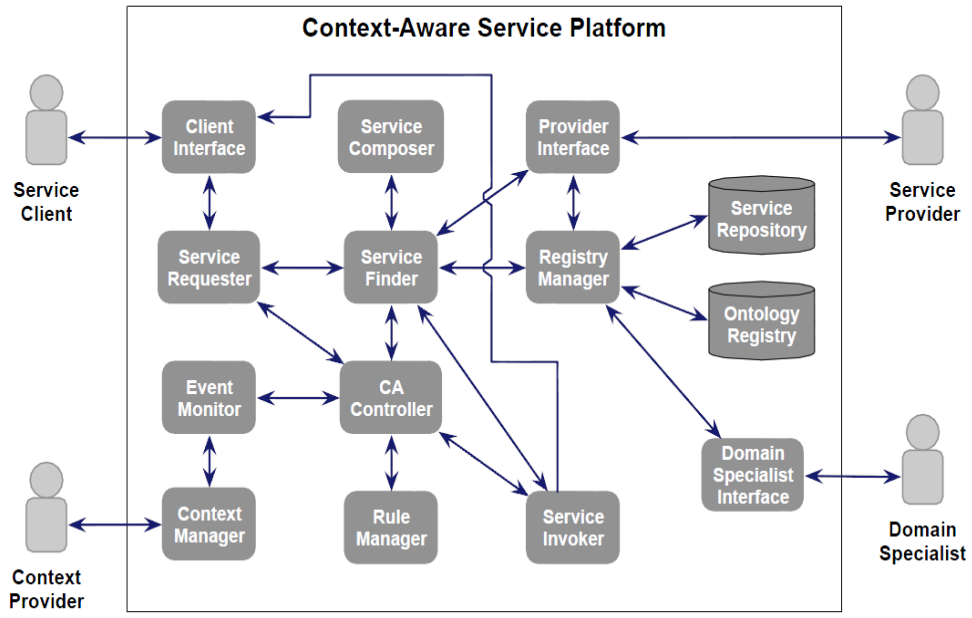

FIGURE 1.1 ARCHITECTURAL DESIGN OF THE CONTEXT-AWARE SERVICE PLATFORM

In order to support service provisioning to users, the platform needs to be able to reason about the information received from users, services and contextual services. Therefore the platform should be aware of the domain a service operates on. It is the task of the context provider to supply mechanisms that allow the platform to gather contextual information of users. This information is used by the platform to reduce direct user interactions with the services. In the case of the CASP, creating and maintaining domain ontologies that describe the domain of the services, is a major task of the domain specialist. These domain ontologies allow the platform to gather and combine contextual information and use this

information in the discovery, selection and invocation of services [7]. The domain specialist should have its own interface to the platform. However, this was not available for this specific platform at the beginning of this project.

1.2 OBJECTIVE

The main objective for this research is twofold: (i) provide an architecture for automatic generation of tool support for domain specialists performing modeling tasks and (ii) evaluate this architecture by means of a prototype tool.

1.3 APPROACH

To achieve the objective the following steps were taken:

- Researching existing techniques.
  Techniques to translate ontologies into metamodels already exist. There also are a number of ontology languages available today. These techniques and languages have been researched to identify usable techniques and functionality to support domain specialists.
- Performing a requirements analysis.
  We identified who the stakeholders of this research are and we set up requirements for the project. These requirements are the basis for the design of the architecture. This requirements analysis is done in the scope of a specific framework: the CASP. This framework contemplates the existence of domain models.
- Selecting a tool environment.
  To develop a prototype tool to support the domain specialist, a tool environment is needed. We determined which environment is the most suitable option for this project.
- Designing the architecture.
  We determined how the generation of tool support can be automated, then we designed the architecture based on this automation and on the requirements.
- Developing the prototype.
  We selected the functionality of the designed architecture that should be implemented in the prototype tool. We developed the prototype tool by implementing the chosen parts of the design.

- Evaluating the prototype.
  Finally, we evaluated the prototype tool by applying it in a use case scenario and checking the fulfillment of the requirements.

1.4 STRUCTURE

The order of the chapters of this thesis corresponds to the order in which issues have been dealt with within this project. Chapter 2 presents the background information that forms the basis for this project. It also discusses the related work. Chapter 3 discusses the requirements analysis, consisting of a stakeholder analysis, a use case scenario, the requirements specification and a section on traceability. Chapter 4 describes the development of the tool, including the design and a description of the prototype. Chapter 5 elaborates on the evaluation of the prototype tool. First the criteria against which the prototype is evaluated are presented. Then the evaluation itself is performed and described, and finally the results of the evaluation are discussed. Chapter 6 elaborates on related work, presents the conclusions of the project and gives suggestions for future work.

2 BACKGROUND

This chapter describes the background information needed to understand this thesis. It starts with an explanation of metamodeling in section 2.1. Section 2.2 shortly presents the field of Service-Oriented Computing. Section 2.3 elaborates on the Unified Foundational Ontology. Section 2.4 provides information on the Context-Aware Service Platform. Finally, section 2.5 elaborates on some platforms and techniques.

2.1 METAMODELING

In the context of software development, a model is "an abstraction of a system allowing predictions or inferences to be made" [8]. The word "meta" originates from the Greek language meaning (among others) "about" or "beyond". Hence, a metamodel is a "model of models" [9], i.e. a metamodel is an abstraction of a model. The metamodel describes valid concepts, relations and properties of a model. A model is formulated in terms of the metamodel, i.e. the metamodel describes the language in which models can be described.

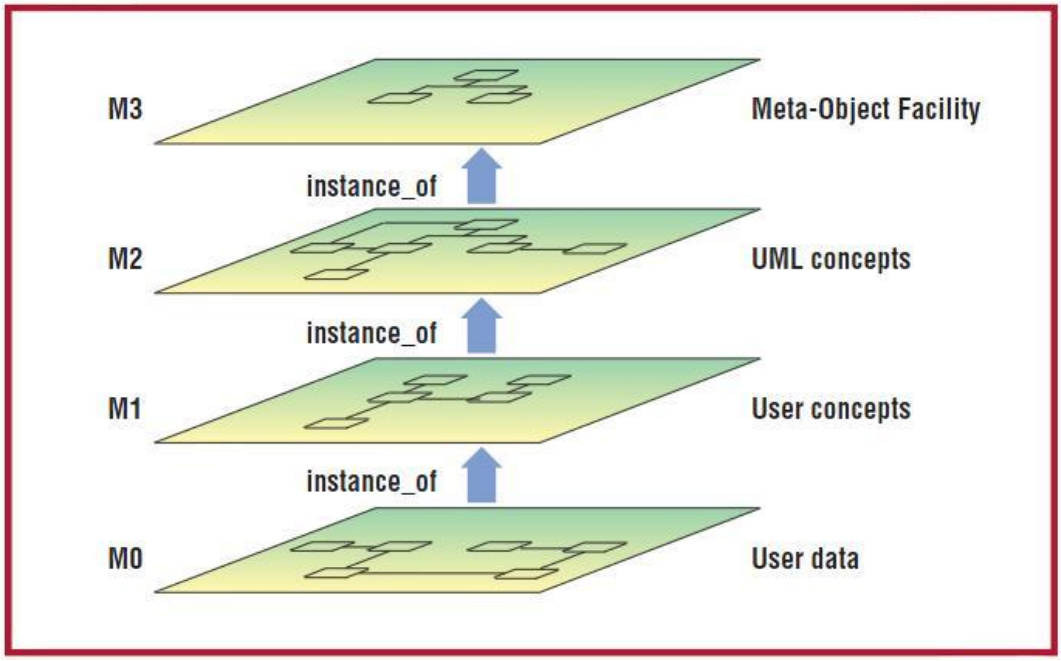

FIGURE 2.1 TRADITIONAL OBJECT MANAGEMENT GROUP MODELING INFRASTRUCTURE

By creating a model of a metamodel, we can add another abstraction level. A metamodel is described by a metametamodel. Since a metamodel describes a

language, we can refer to a metametamodel as a model of a metalanguage, i.e. a language that can be used to describe languages. These layers of abstraction form the traditional Object Management Group (OMG) modeling infrastructure (Figure 2.1). Each level (except the top level) in this infrastructure is characterized as an instance of the level above [10]. M0, the bottom level, is where the real world objects are. The next level, M1, is the level of models, an abstraction of the real world objects. Level M2 contains the metamodels, describing the languages used to represent the models at level M1, e.g. the UML language [11]. The top level, M3, contains the metalanguages. Examples of metalanguages are Meta Object Facility (MOF) [12] and Ecore.

Bezivin [13] has a slightly different view on the OMG modeling infrastructure. He claims that this infrastructure should more precisely be named a 3+1 architecture (Figure 2.2). The bottom level still contains the real system, which is represented by a model in the M1 level. The model conforms to its metamodel in the M2 level. The metamodel itself conforms to the metametamodel, the metalanguage, in the M3 level. A metametamodel conforms to itself.

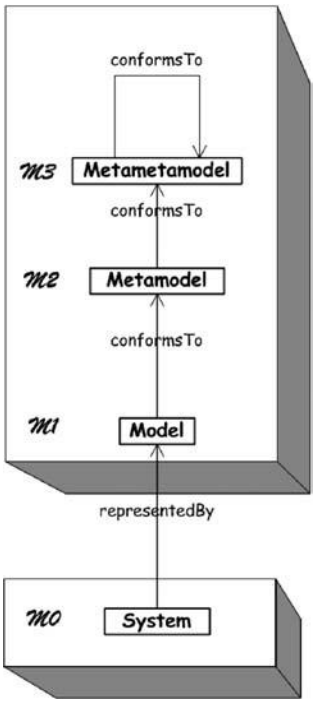

FIGURE 2.2 3+1 ARCHITECTURE

2.2 SERVICE-ORIENTED COMPUTING

Service-Oriented Computing (SOC) is a computing paradigm with the vision that services represent distributed pieces of functionality. Combining these pieces can result in additional and more complex functionality. In the vision of SOC, services are the constructs that can be used to facilitate the development of distributed applications with low costs. Services are autonomous, platform-independent computational entities that can be easily composed to develop a range of distributed systems, independent of a specific platform. The ultimate goal of SOC is that a service can be requested by an end-user by just expressing requirements, leaving the software infrastructure responsible for the discovery, selection, composition and invocation of the services, without any human interference [14].

2.3 UNIFIED FOUNDATIONAL ONTOLOGY

A language to represent an ontology should be grounded on a foundational ontology that defines a set of domain-independent real-world concepts, which can be used to talk about reality. According to Guizzardi [15] an ontology representation language *"should commit to a domain-independent theory of real-world categories that account for the ontological distinctions underlying language and cognition."* A foundational ontology can also be called a meta-ontology or an upper level ontology (ULO). A unified foundational ontology is a combination of some foundational ontologies.

The Unified Foundational Ontology (UFO), developed by Guizzardi and Wagner [16], is a combination of the foundational ontologies GFO/GOL [17] and OntoClean/DOLCE [18]. The design of UFO is split into three incremental sets. UFO-A defines the core of UFO, i.e. terms like *Thing*, *Entity*, *Entity Type* and *Individual*. UFO-B increments UFO-A by adding terms related to perdurants. A perdurant, as opposed to an endurant, is a kind of Individual that does not have to be wholly present whenever it is present. A perdurant is composed of temporal parts. If a perdurant is present, it might not be the case that all its temporal parts are present. An endurant, which is defined in UFO-A, is always wholly present whenever it is present. Examples of endurants are tangible things, like a table or a tree. Examples of perdurants are events, like a conversation, or the middle ages. UFO-C increments UFO-B by adding terms related to the areas of intentional, social and linguistic issues. Examples are the enrichment of the notion of event to

be an action or a non-action, and the notion of communication between endurants.

2.4 CONTEXT-AWARE SERVICE PLATFORM

The Context-Aware Service Platform (CASP) [7] is a platform, developed at the University of Twente, aimed at supporting service provisioning to non-technical users, developed at Twente University. This platform allows users to use concepts close to their natural perception to expressing their service requests. It also reduces the need of direct user interactions with the services. The platform should deal with finding the most optimal service, selecting the service, possibly negotiating with the service, invoking the service and handling the results of the service. The user just has to specify the service request and possible restrictions. These service requests can be specified in an intuitive way, to enable also non-technical users to use the platform.

Four stakeholders were identified for this platform. Their roles are explained below:

- Service client
  The service client is the one who requests the service provisioning. The service client also deals with possible negotiations on the terms of service provisioning, e.g. discounts on bulk purchases. There is a distinction between service client and service beneficiary, the first being the one who requests the service provisioning and the latter being the one who actually benefits from it. Often these are the same, however, it is possible that the service client and the service beneficiary are different persons, e.g. a parent contracting the education services of a school for his child, the parent being the service client and the child being the service beneficiary. For simplicity in the description of the platform, the service client is assumed to be also the service beneficiary.
- Service provider
  The service provider is responsible for the service provisioning of its offered services. The service provider is also responsible for providing the service descriptions of its offered services and semantically annotating the terms in these descriptions. A distinction can be made between a service provider and a service executor, similarly to the distinction between a service client and a service beneficiary. The

service provider is responsible for the service and the service executor actually performs the activities related to the service. For simplicity, the service provider and the service executor are also assumed to be the same entity.

- Context provider
  The context provider is responsible for supplying mechanisms that allow the platform to gather contextual information about service clients. These mechanisms should gather contextual information from the service client's software-based data and from sensor devices. The gathered information is used to reduce the amount of user interactions with the platform.
- Domain specialist
  The domain specialist is responsible for gathering relevant knowledge of a particular domain and representing this knowledge in terms of a domain ontology. Domain ontologies are semantic descriptions of the concepts in a particular domain, therefore they can be used to semantically annotate the terms in service descriptions.

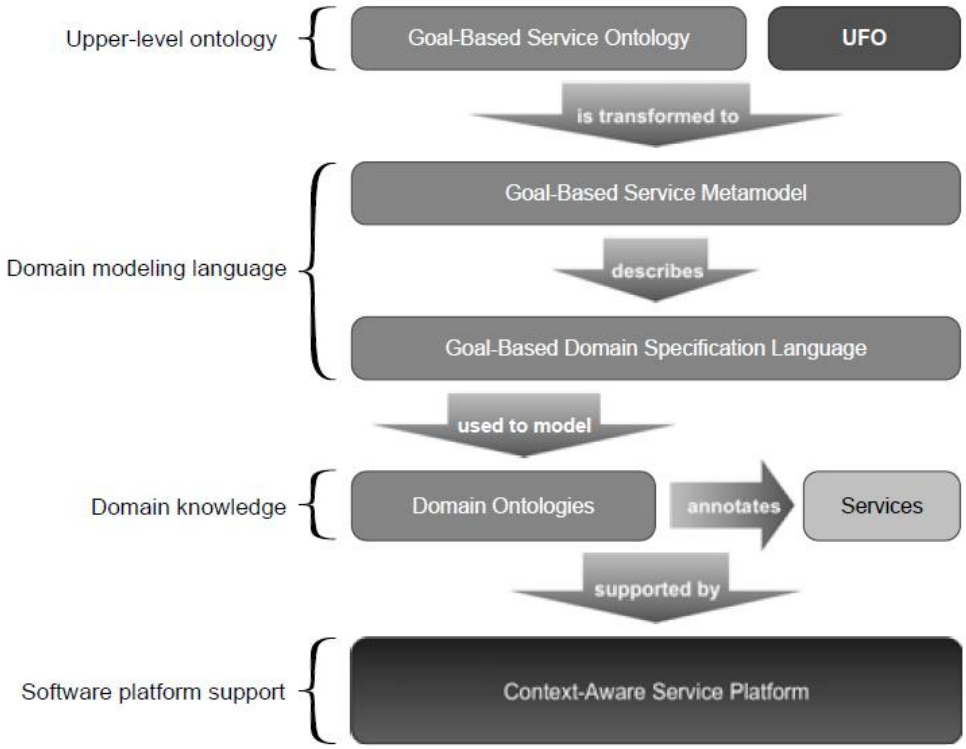

FIGURE 2.3 GOAL-BASED SERVICE FRAMEWORK

The CASP is embedded in the Goal-Based Service Framework (GSF), which is shown in Figure 2.3. At the top is the Goal-Based Service Ontology (GSO), which extends the UFO by adding SOC related concepts, goals, tasks and services to it. The GSO defines domain-independent concepts, which can be used in domain ontologies. In the framework, below the GSO is the Goal-Based Service Metamodel (GSM), into which the GSO should be transformed. A metamodel can describe a Domain Specific Language (DSL). A DSL provides a notation specific for an application domain. A DSL is based on the relevant features and concepts of that domain [19]. The GSM describes the Goal-Based Domain Specification Language (GDSL), in which domain ontologies are to be modeled. The application domain, which is described by the GDSL, is thus a broad domain, namely the domain described by the GSO, which defines domain-independent real-world concepts. Domain ontologies describe a domain by defining concepts, goals and tasks specific to that domain, and the relations among them. The domain ontologies are then used to annotate services supported by the CASP. The CASP facilitates interactions between the service providers and the service clients, and supports these interactions by providing mechanisms for the publication of services to the service providers, and mechanisms for the discovery, selection and invocation of services to the service clients.

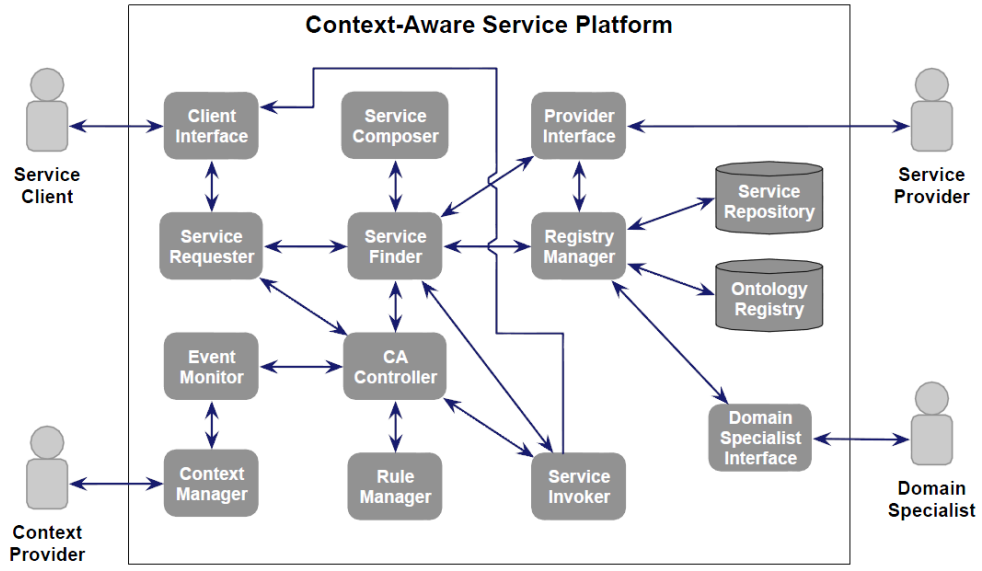

FIGURE 2.4 ARCHITECTURAL DESIGN OF THE CONTEXT-AWARE SERVICE PLATFORM

Figure 2.4 shows the architectural design of the CASP. The CASP components are divided in three areas: Stakeholders' Interface Components, Service Provisioning Components and Context-Aware Components [7] [19].

Stakeholders' Interface Components provide the stakeholders with interfaces to the platform. They allow applications, operated by the stakeholder, to interact with the platform. The API's of these interfaces provide methods for interactivity with the platform, e.g. submitting service requests by service clients, retrieving domain ontologies for annotating service descriptions by service providers, managing registration of contextual information by context providers and managing domain ontologies by domain specialists.

Service Provisioning Components handle the process of discovering, selecting and invoking services. They use the goals of the service client and its contextual information for this process. The service clients' goal is represented by a specification of a state of affairs that satisfies the goal. The Service Provisioning Components generate a service request, discover candidate services, compose services if needed, invoke the selected services and provide the Client Interface with the outputs to inform the service client.

The Context-Aware Components gather contextual information and make this information accessible to the other components. These components provide the contextual information that is necessary for the Service Provisioning Components to discover, select and invoke the correct services. They also provide the Stakeholders' Interface Components with the contextual information the users need to operate the platform.

2.5 PLATFORMS AND TECHNIQUES

This section describes the platforms and techniques used in our research. It gives a description of the platforms Protégé and Eclipse and their possible uses in our research. It also discusses the EMF and GMF technology, and the EMF4SW tool, which is a plug-in for the Eclipse platform.

2.5.1 PROTÉGÉ

Protégé [20] [21] is an open-source platform that provides users with a set of tools to create domain models and knowledge-based applications with ontologies. It is developed at Stanford Medical Informatics. Protégé provides users with knowledge modeling structures and actions to create, visualize

and manipulate ontologies. This can be done in various formats. Protégé can be extended by defining new plug-ins. The system is domain-independent and has been successfully used in many application areas. The platform is separated into two parts: (i) a model and (ii) a view. The model is based on a flexible metamodel [22] that can represent ontologies. The model is the internal representation mechanism for ontologies and knowledge bases. One of the strengths of Protégé is that the Protégé metamodel itself is a Protégé ontology, facilitating extension and adaption to other representations. The view components provide a user interface that displays the underlying model. With the views of the user interface it is possible to create and maintain ontologies. Protégé is able to automatically generate user interfaces that support the creation of individuals for these ontologies. These interfaces can be further customized by the user with the Protégé's form editor.

Two main ways of modeling ontologies are supported by Protégé: Protégé-Frames and Protégé-OWL. Protégé-Frames enables users to build frame-based ontologies. Protégé-OWL is an extension of Protégé that enables users to build ontologies for the Semantic Web. Protégé-OWL is interesting to our research, mainly to develop and maintain ontologies that can be used as input for the transformation tool. Protégé-OWL is a complex Protégé extension that can be used for much more, like editing databases, however, since that is not part of our research we will not discuss this here.

2.5.2 ECLIPSE

Eclipse [23] is an open source community that carries out projects to create an extensible development platform, runtimes and application frameworks. These are intended for building, developing and managing software. The Eclipse platform is a universal platform for integrating development tools. Eclipse allows the development of new plug-ins. Almost everything in Eclipse is a plug-in. These plug-ins can add functionality to the Eclipse platform by providing code, but they can also only provide documentation, resource bundles or other data to be used by other plug-ins. A plug-in exists of at least the plug-in manifest file (*plugin.xml*). This file describes how the plug-in extends the platform, what extensions it publishes and how its functionality is implemented. One of the fundamental features of the Eclipse platform is that applications built on top of it, look and feel like native Eclipse applications. The Eclipse platform is interesting to our research to be used to develop the (prototype) tool and deploy the (prototype) tool as a plug-in.

2.5.3 EMF/GMF

The core of a DSL is its abstract syntax, which is used in the development of almost every artifact that follows in the development of a DSL. Eclipse Modeling Framework (EMF) provides the means for the development of the abstract syntax. In its project description, EMF is described as "a modeling framework and code generation facility for building tools and other applications based on a structured data model." EMF consists of several components, which provide functionality to create, edit, validate, query, search and compare models. EMF has an Ecore model, which is the metamodel for defining a DSL. The semantics and structure of the DSL can be refined further by defining Object Constraint Language (OCL) constraints.

To expose the abstract syntax for use by humans, one or more concrete syntaxes have to be created. The Graphical Modeling Framework (GMF) can be used to develop a concrete syntax for a DSL and to map the concrete syntax to the abstract syntax. These models can be used to generate a diagram editor. GMF consists of two components: a runtime and a tooling framework. The runtime bridges the gap between EMF and GEF (Graphical Editing Framework, a framework to develop graphical editors). The tooling component allows one to define graphical elements, diagram tooling and mappings to a domain model in a model-driven way [24].

2.5.4 EMF4SW

Eclipse Modeling for Semantic Web (EMF4SW) [25] is a set of Eclipse plug-ins that bridges the gap between EMF and some Semantic Web modeling languages, like OWL and RDF, by providing metamodels for these languages. It also provides model transformations that allow a user to convert models from one language into another, e.g. Ecore to OWL or the other way around. EMF4SW includes a Java API to access these transformations, but they can also be accessed via an Eclipse menu.

3 REQUIREMENTS ANALYSIS

We are aiming to generate tool support for the domain specialist. This tool support can use the Domain Specialist Interface to communicate with the platform. To investigate this support we started with a stakeholder analysis to identify the stakeholders of this tool, which is described in section 3.1. After that we present a use case scenario in section 3.2. Section 3.3 gives the requirements for the prototype tool. Finally, section 3.4 elaborates on the importance of traceability.

3.1 STAKEHOLDER ANALYSIS

In order to identify the stakeholders, we first need to establish a thorough understanding of one of the existing needs of the CASP. To develop and maintain domain ontologies, an ontology editor is needed. This editor should use the domain specialist interface to communicate with the CASP. The domain specialist has to develop and maintain ontologies in a language, which is provided by a language designer. We give a schema of the system in order to visualize this need, which is shown in Figure 3.1.

To fulfill this, need we need to come up with a way to create an editor. To do this we provide two approaches. In both approaches, the language designer designs the language as an upper level ontology. A DSL is described by a metamodel, so we need to translate the ULO to a metamodel and then derive the DSL from that metamodel. The DSL can then be used to create and maintain domain ontologies. These relationships are depicted in Figure 3.2.

In the first approach, we manually translate the ULO to a metamodel. We then derive the DSL and generate an editor for this DSL with the EMF and GMF technologies. We can then tune the editor to the needs of the domain specialist. In the second approach, we translate the ULO to a metamodel automatically and then generate an editor also automatically. This approach requires more research time, since we have to find a way to do all the steps automatically. In this option we do not have the opportunity to tune the editor to the needs of the domain specialist.

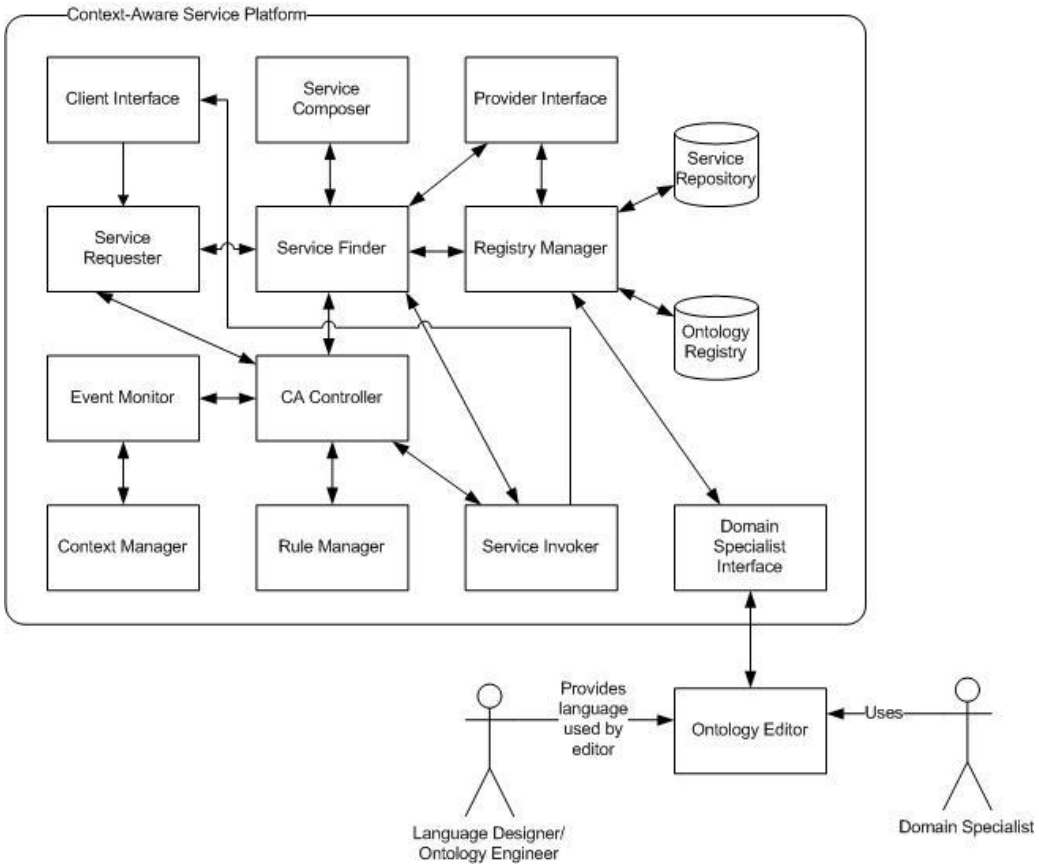

FIGURE 3.1 ONTOLOGY EDITOR FOR DEVELOPING AND MAINTAINING DOMAIN ONTOLOGIES FOR THE CONTEXT-AWARE SERVICE PLATFORM

A major benefit of the first approach is that the editor can be tuned according to the needs of the domain specialist, whereas this is not the case in the second approach. A major benefit of the second approach is that the editor is not rigid, as opposed to the editor in the first approach. If something needs to be changed in the ULO, one can simply (automatically) regenerate the editor. In the first approach, if anything changes, all steps will have to be done again by hand. In the case the ULO is changed often, the first approach will result in a massive amount of work, whereas in the second approach no extra development work at all is necessary. Another distinction between the approaches is the scientific value. The scientific value of the first approach is limited, since it does not introduce any new methods, new insights or major improvement of methods. The scientific value of the second approach is significantly higher, since it involves creating and improving methods to automatically translate an ontology into a metamodel and to automatically generate an editor.

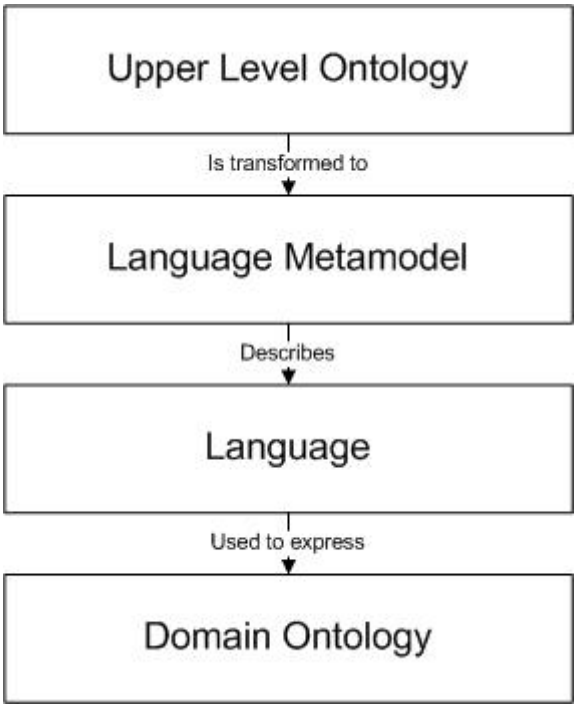

FIGURE 3.2 FROM UPPER LEVEL ONTOLOGY TO DOMAIN ONTOLOGY

In both approaches, traces between constructs have to be kept. The editor will be used to create domain ontologies. In case something changes in the ULO, the editor has to be regenerated, either by hand or automatically. Traces between constructs can then help decide whether the already existing domain ontologies are still valid and whether the language used in the new editor is indeed translated correctly from the new ULO. Keeping traces in the second approach is less error prone than in the first approach, since it can also be done automatically instead of by hand.

Based on the aforementioned arguments we decided to apply the second approach in this research project. An overview of this approach is depicted in Figure 3.3. In this research project we developed a transformation tool that takes an ULO as input and generates an editor for this ULO.

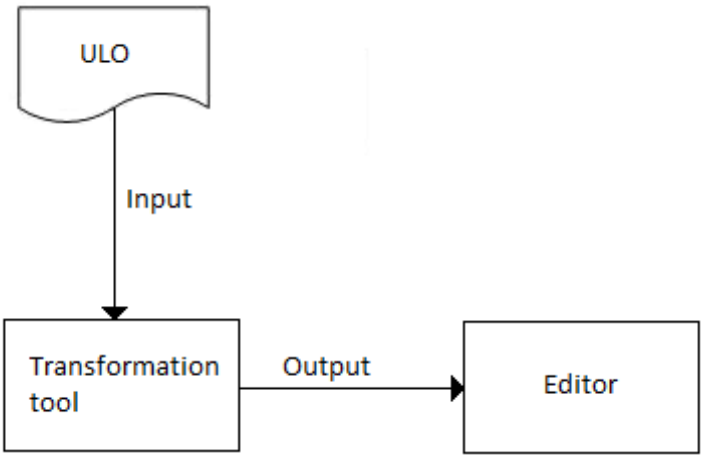

FIGURE 3.3 OVERVIEW OF THE TRANSFORMATION TOOL

Figure 3.4 shows the environment in which the tool operates. In this environment we identified two stakeholders: (1) the *language designer*, who feeds the transformation tool with the ULO. (2) The *domain specialist*, who develops and maintains domain ontologies, using the resulting editor.

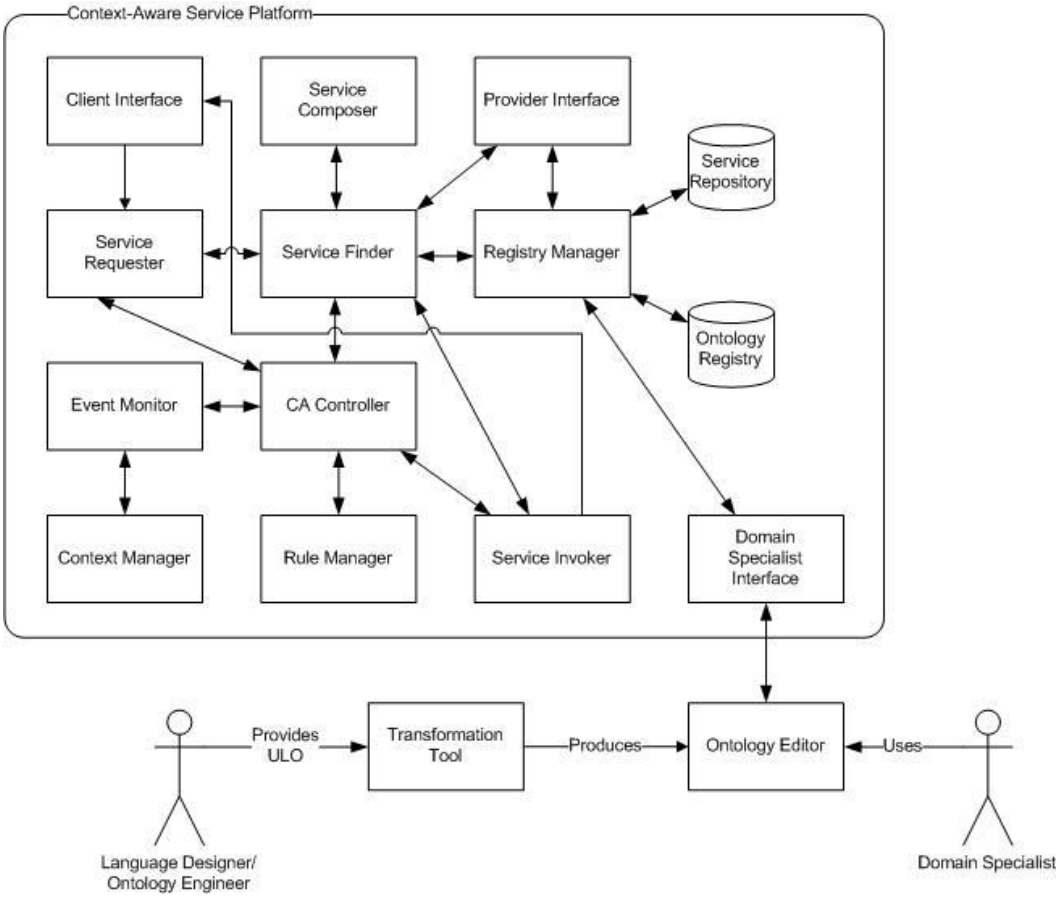

FIGURE 3.4 ENVIRONMENT OF THE TRANSFORMATION TOOL

3.2 USE CASE SCENARIO

This section presents a use case scenario aimed at identifying usage patterns for the transformation tool. For our use case scenario we use the notion of a mind map [26]. We define here the notion of a mind map. Afterwards we describe how we use this notion for the transformation tool.

A mind map is a diagram used to represent topics that are arranged around and linked to a central topic. A topic can be a word, an idea, a task or anything else. Mind maps are used to achieve various goals, e.g. to help generate and visualize ideas, to organize and study information, to recall memories or to solve problems. The elements of a mind map are arranged intuitively according to the importance of the concepts. A mind map is usually a drawing in which the central topic is in the middle of the page. The other concepts are arranged around the central topic and are classified into branches or groupings, aiming to represent semantics or other connections between pieces of information. This way of drawing a mind map enables brainstorming. The branches of a mind map represent a hierarchical structure, but their arrangement disrupts the prioritization of concepts that usually comes with a hierarchical structure. This encourages users to connect concepts to each other without using a particular conceptual framework.

Colors and images are used when drawing a mind map. Since it is a graphical way of brainstorming, visual effects are important. Colors are used for visual stimulation and to group concepts. Importance can also be made visible with visual effects, like thick lines between concepts. A big difference between mind mapping and other ways of modeling (like UML) is that there is no explicit related abstract syntax with mind mapping. Mind maps serve the purpose of supporting memory and organization. One can develop his own mind mapping style. An example of a mind map is shown in Figure 3.5.

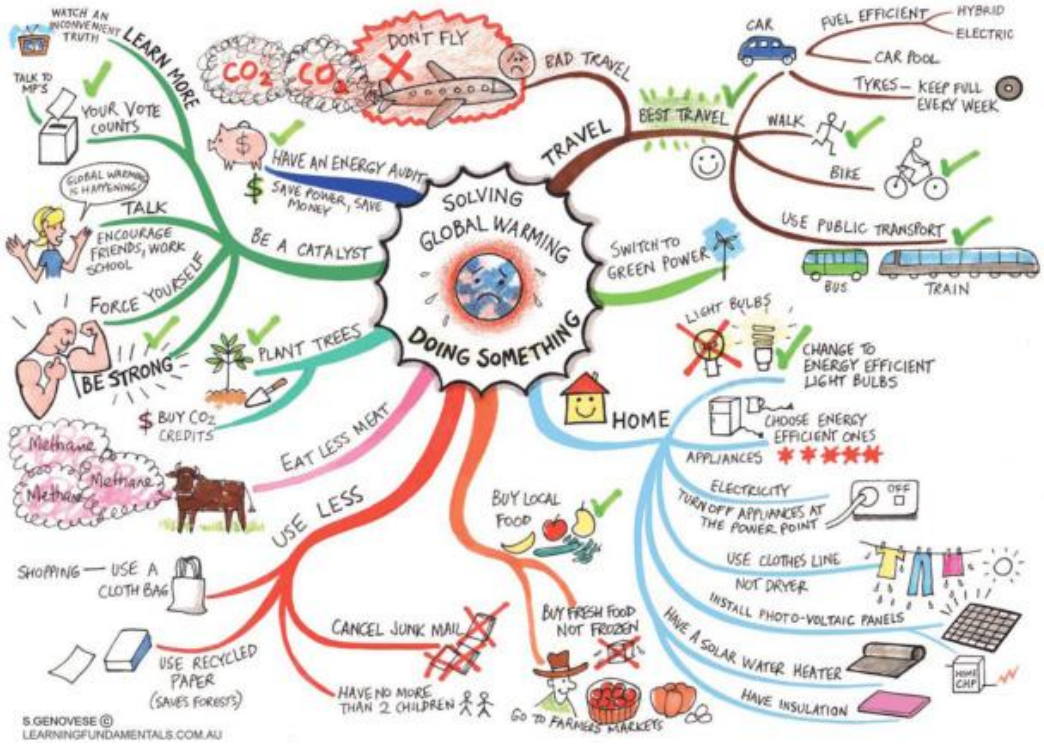

FIGURE 3.5 AN EXAMPLE MIND MAP

To represent mind maps on a computer, we can model the concepts, creating an ULO for a mind map language. This ULO describes concepts that can be used to create mind maps. Since an ULO represents domain-independent concepts that exist in the world, this is a quite simplified view on the world. This means that in our view the world consists of mind maps. However, for this use case scenario, which is used to evaluate our prototype tool, this simplified view has done just fine. A mind map can be about anything, which makes the described concepts domain-independent. The language we generate from this ULO can be used to describe mind maps, which in this sense are domain-dependent instantiations of the domain-independent concepts described by the ULO. We realize that we stretch the definition of an ULO to the limit, but for this use case scenario the mind map ULO is enough. A mind map created with these concepts can model anything that is of importance to the user. In this respect a mind map is a domain ontology.

The mind map ULO we used in our work was written in OWL. It defines 6 classes: Type, Priority, Map, MapElement, Topic and Relationship. Topic and Relationship are subclasses of MapElement. These subclasses are disjoint. We put a covering axiom on the subclasses of MapElement, meaning that an individual that is in MapElement, always must also be in either Topic or

Relationship. There cannot be an individual that is only a MapElement. The classes Type and Priority are enumerated classes. The class Type enumerates three individuals: DEPENDENCY, EXTEND and INCLUDE. The class Priority also enumerates three individuals: HIGH, MEDIUM and LOW. We modelled this by adding an equivalent class to both classes, listing their individuals between curly brackets. For the class Type the equivalent class is {DEPENDENDY, EXTEND, INCLUDE} and for the class Priority it is {HIGH, MEDIUM, LOW}.

We also defined object type properties: elements, rootTopics, parent, subtopics, hasPriority, hasType, source and target. The object type property elements has domain Map and range MapElements. This is where we encountered a limitation of OWL. We intended this property to be a containment. However, containments do not exist in OWL. We chose to just use an object type property and adapt the metamodel after translation. The object type property rootTopics, pointing to the central topic(s), has domain Map and range Topic. The properties parent and subtopics are inverse properties of each other, both with domain and range Topic. The property parent is functional, meaning that, for a given individual, there can be at most 1 individual that is related to the individual through this property. Since property subtopics is the inverse property of parent, subtopics is inverse functional, meaning that the inverse property is of this property is functional. The object type property hasPriority has domain Topic and range Priority. The class Relationship is the domain of the object type properties hasType, source and target. The range of hasType is the class Type and the range of source and target is the class Topic. The properties hasPriority, hasType, source and target are all functional. For all of these 4 properties a restriction is formulated that relates individuals from the domain classes of these properties to exactly one individual, instead of to at most one individual of the range classes.

Finally, we also defined the data type properties created, title, name, description, start and end. The data type property created has domain Map and range date and the property title has domain Map and range string. The property name has domain MapElement and range String. A description is a string data type property for an individual in the class Topic. Both the start and end property have domain Topic and range date. For all data type properties discussed here, a restriction is added that the concerning individuals have exactly one of these data type properties. Figure 3.6 shows

the class hierarchy of the mind map ULO, in which Thing is the superclass of everything.

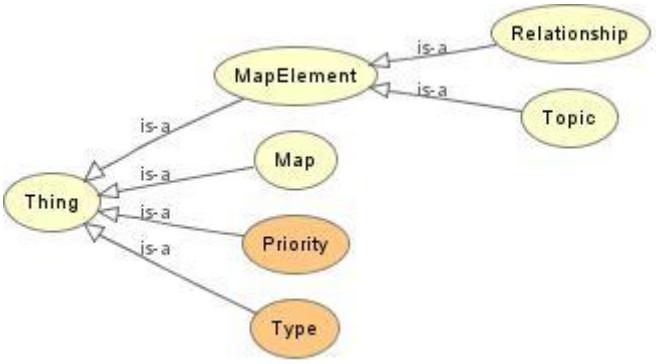

FIGURE 3.6 CLASS HIERARCHY OF THE MIND MAP ULO

The ULO we used does not include concepts like 'color' or 'image'. However, since a Topic has a description, we can describe these aspects for each Topic. To make the ULO more powerful and complete, these concepts could be added to the ULO. For our evaluation, however, we did not find it necessary, because it does not influence the behavior of the transformation tool.

Our mind map ULO is the input to the transformation tool. The tool generates a DSL from the mind map ULO, which allows users to model mind maps. From this DSL the transformation tool generates a graphical editor, which uses the language. The resulting editor can be used to graphically create mind maps, enabling users to create mind maps in a similar way as drawing on paper and at the same time providing them with the possibility of computer support.

3.3 REQUIREMENTS

The requirements are formulated for the transformation tool, based on the use case scenario and the stakeholder analysis. We kept them very general, since we intend the transformation tool to be very general, i.e. the transformation tool should work with an ULO specified in any ontology language.

For the transformation tool the following requirements were formulated.

1. Data requirements:

R1. *The transformation tool should accept an ULO as input.*
The input of the transformation tool is the ULO provided by the language designer. The ULO should be represented in an ontology language.

R2. *The transformation tool should generate as output an editor to be used by the domain specialist.*
After various transformations the editor should be the output. This editor will either be a plug-in for Eclipse or a standalone editor.

2. Functional requirements:

R3. *The transformation tool should generate a DSL from the ULO.*
The ULO is provided by the language designer, defined in an ontology language. This ontology should be converted into a DSL, which is to be used by the resulting editor.

R4. *The generated DSL should allow domain ontologies to be described.*
The DSL is the language in which domain ontologies have to be described. The domain specialist uses the DSL accordingly.

R5. *The transformation tool should allow the ULO to be specified in any ontology language.*
The tool should be very general. By allowing the ULO to be specified in an arbitrary ontology language we do not bound the tool to one or more specific languages.

R6. *The editor should contain functions to add, load and save a domain ontology.*
At least the most basic manipulation functions should be supported by the editor.

R7. *The editor should be extendable.*
It should be possible to extend the editor with more functions. This can be done by either altering the transformation tool or the editor itself.

3. Quality requirements:

Traceability

R8. *The transformation tool should provide traceability from the changes in the ULO to domain ontologies.*

When the ULO is changed, the editor should be regenerated. The transformation tool should keep track of the ULO changes and provide the user with information about which constructs in which ontologies will have to be changed due to the changed ULO. This form of traceability is interesting to the domain specialist.

R9. The transformation tool should provide traceability from the ULO to the DSL.

Due to technology constraints it might not be possible to generate the DSL from the ULO exactly as it was intended by the language designer (e.g. it might be impossible to map a construct in the ULO directly to a construct in the DSL). This might result in language concepts that do not match the ULO concepts. The transformation tool should notify the language designer of the differences between the ULO and the DSL, providing the language designer with the option to either accept the differences or change the ULO. This form of traceability is interesting to the language designer.

Compliance

R10. The generated DSL should comply as much as possible with the ULO given as input.

Due to technology constraints it might not be possible to have full compliance between the ULO and the DSL. The intention is to have as much compliance as possible.

3.4 TRACEABILITY

Two requirements are concerned with the traceability provided by the tool. If the ULO is changed and the editor is generated again, the constructs defined in already existing domain ontologies might be incorrect or the meaning of the constructs might have been changed. Traceability in these constructs indicates which concepts and properties of the domain ontologies correspond to which concepts and properties of the ULO. The transformation tool should provide users with a list of constructs affected by the change. To be able to do this, the previous metamodel (and thus the previous DSL) should be stored. When the previous and the new metamodel are compared, the constructs that have been changed (or even removed) can be derived. When the constructs are known, the tool should search the ontology registry for their affected use and then notify the users by providing a list of affected ontologies. It is then

up to the user to decide whether the ontologies are still valid or they need to be changed. A tool can be built to help the user with these decisions.

The other kind of traceability described in the requirements specification, is about keeping the traces between the constructs of the provided ULO and the constructs of the resulting DSL. Traceability in these constructs indicates which concepts and properties of the ULO result in the concepts and properties of the DSL. These traces should be provided to the language designer, in the form of a diagnostics file, to provide him with the information he needs to verify the correctness of the transformation. For analysis of this diagnostics file a tool can be built to help the language designer interpret the traces.

4 DEVELOPMENT

This chapter describes the design of the transformation tool and also elaborates on the prototype tool itself. The design has been made to meet the requirements as closely as possible. Due to time limitations we had to make a selection of the parts of the design we have implemented in the prototype tool. Section 4.1 describes the architectural design of the transformation tool, presenting the components of the tool and the flow of artifacts between them. It also presents the architecture of the components. Section 4.2 elaborates on the tool chain, presenting the sequence in which actions have to be taken and tasks have to be executed. Section 4.3 describes translation rules, which have to be added to the translation rules of the EMF4SW tool. Finally, the prototype is described in section 4.4.

4.1 ARCHITECTURAL DESIGN

The design of the prototype tool starts with the architectural design of the tool. The architectural design shows the components of the tool and the way they interact with each other. This is depicted in Figure 4.1. The input to the transformation tool is an ULO, defined in some ontology language, e.g. the Web Ontology Language (OWL). The first component of the tool is the translator. Its task is to translate the ULO into a metamodel, defined in some metalanguage, e.g. Ecore. To perform the translation, the translator needs a set of translations rules. To make the tool general we designed the translator to use a repository with translation rules for the used languages. If the ULO is defined in OWL and the resulting metamodel is requested to be defined in Ecore, the translator takes the OWL-to-Ecore translation rules from the repository and uses them to translate the ULO. Besides the metamodel, the translator produces a log file, which contains the performed mappings. This log file can be used (possibly with the help of an analysis tool) to check whether the translation has been performed as intended or not.

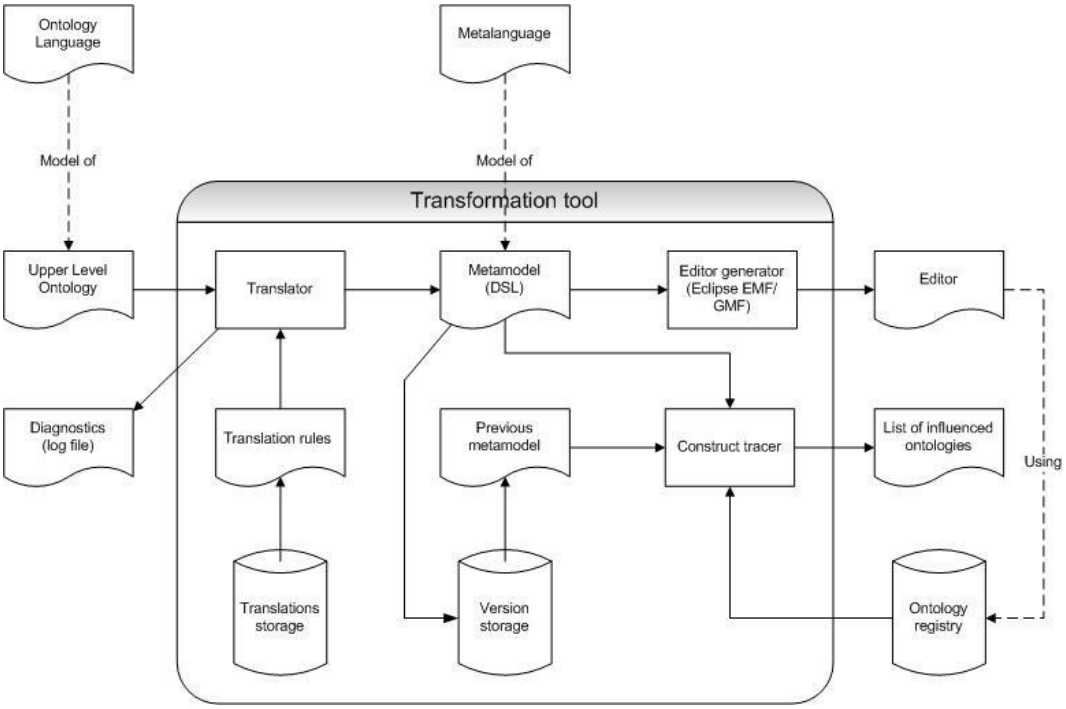

FIGURE 4.1 ARCHITECTURAL DESIGN OF THE TRANSFORMATION TOOL

When the metamodel is produced, the tool checks if there has been an earlier version of this metamodel. If this is the case, the DSL defined by the metamodel was already in use. The tool retrieves the previous metamodel from the version storage, and invokes the construct tracer with the new metamodel and the previous metamodel as input. The construct tracer analyzes the metamodels and determines the differences. It then takes the existing domain ontologies from the ontology registry and determines whether these ontologies have been affected by the change of the metamodel. The construct tracer produces a list with the influenced ontologies as output. The domain specialist should then check and possibly update the ontologies on the list. The construct tracer is depicted in more detail in Figure 4.2. It shows that the construct tracer consists of a comparator, which compares the metamodels and determines the affected constructs, and a construct locator, that searches for the provided constructs in the existing domain ontologies and produces a list with influenced ontologies.

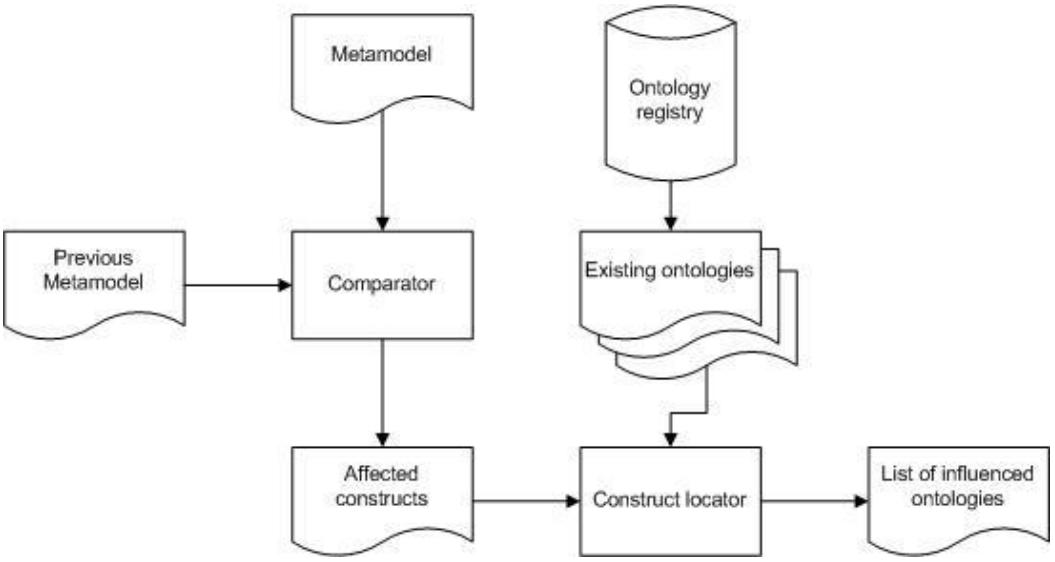

FIGURE 4.2 DETAILED OVERVIEW OF THE CONSTRUCT TRACER

The last component of the Transformation tool is the editor generator. The editor generator takes the produced metamodel as input and automatically generates a graphical editor. The editor can then be used by the domain specialist to create and maintain domain ontologies. The editor generator uses EMF and GMF technology to create the graphical editor. Using this technology introduces a requirement on the used metalanguage, since EMF and GMF require the metamodel to be represented as an Ecore file. That means that the ULO should always be translated to Ecore. The ULO can still be specified in any ontology language, provided that the correct set of translation rules for this translation is added to the repository. The editor generator is depicted in more detail in Figure 4.3, which shows how the input (the metamodel) is used to generate the various artifacts and eventually the graphical editor. These artifacts are needed to generate an editor using EMF and GMF technology. EMF and GMF provide functionality for the Eclipse platform to generate these artifacts by hand. However, since we intend to generate the editor automatically, we have to generate the artifacts also automatically. First the metamodel is used to generate the domain generator model. This model is then used to generate the domain code, which provides the modeled domain and a tree-based editor. After that, the domain model (the metamodel) is used again to generate the graphical definition model and the tooling definition model. The graphical definition model defines the graphical elements that can be used on a diagramming surface. The tooling definition model specifies which tools can be used in the resulting graphical editor. The combination of the graphical definition model, the tooling

definition model and the domain model results in a mapping model. The mapping model maps the elements of the graphical definition model to the domain model and the tooling elements. Then the mapping model can be transformed into a diagram editor generator model. Finally, this model is used to generate the graphical editor code, which together with the domain code forms the graphical editor.

The step from domain model, graphical definition model and tooling definition model to mapping model involves a lot of decisions, e.g. decisions on which constructs in the metamodel should become links and which should become nodes in the resulting graphical editor. We can use automatic recognition of these links and nodes based on names or languages constructs, however, since we want the tool to be general and to be used for multiple ontology languages and the formalisms behind them, we decided to ask input from the user at this point. This means the tool does not generate a graphical editor automatically, but semi-automatically.

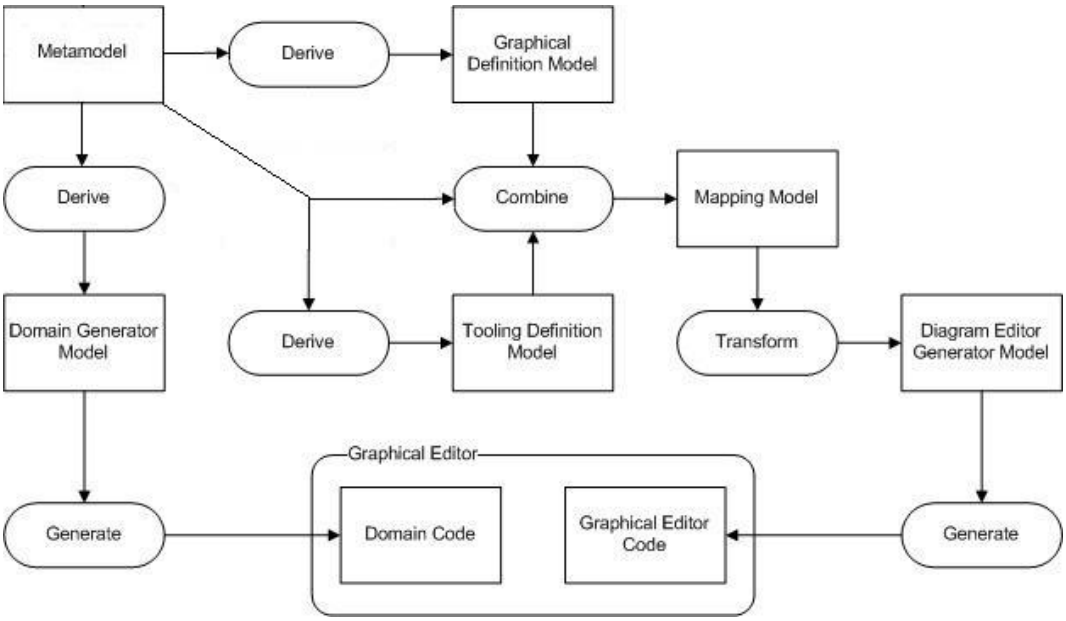

FIGURE 4.3 DETAILED OVERVIEW OF THE EDITOR GENERATOR

4.2 TOOL CHAIN

Figure 4.4 depicts the originally intended sequence of events. The sequence starts with the user (language designer) invoking the Transformation tool and providing the ULO. The tool then invokes the translator, providing the ULO to the translator, and waits for the result. The translator outputs the

Ecore metamodel and the translator log file. The tool sends the log file to the user, allowing the user to validate the translation. When the user validates the translation, the tool continues by invoking the editor generator, which produces the editor. After generating the editor, the tool retrieves the previous metamodel and passes it with the new metamodel to the construct tracer. The construct tracer performs its job and passes back the construct tracer log file, containing the influenced ontologies. The tool passes this log file on to the user. Subsequently it saves the new metamodel and returns the final result (the editor) to the user.

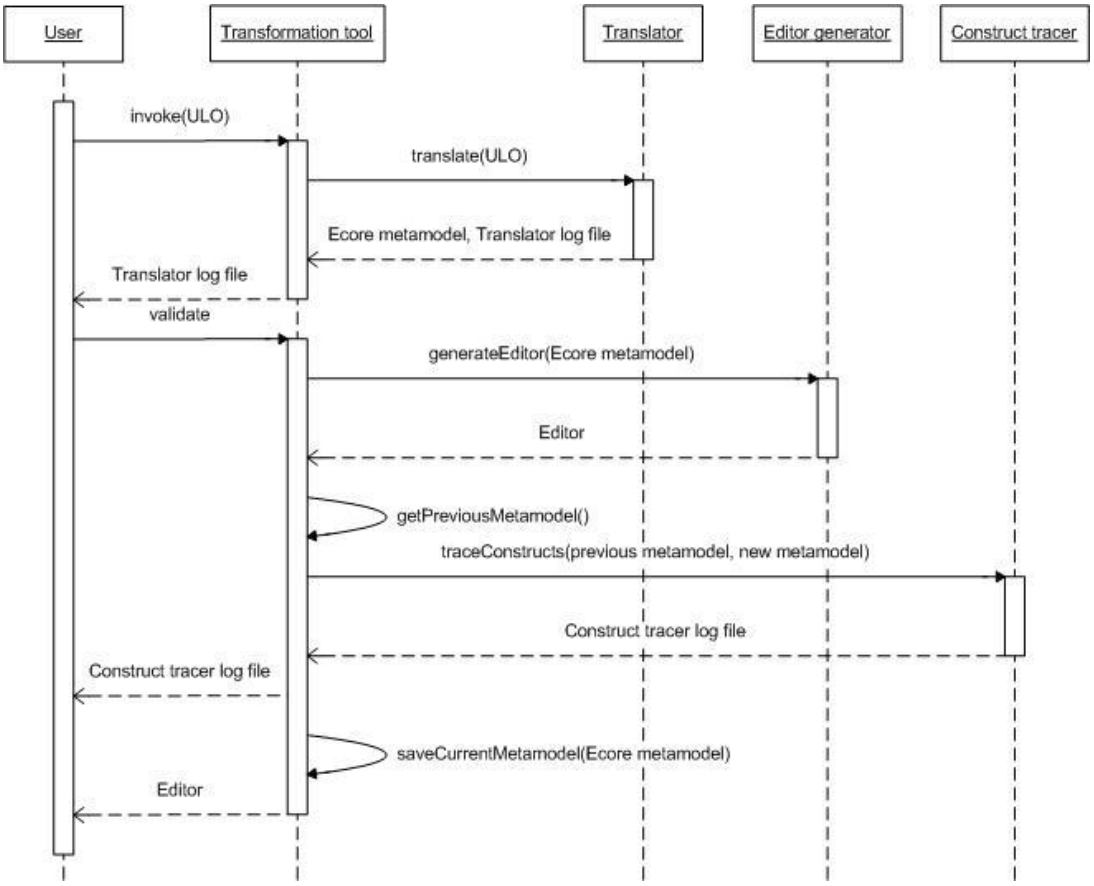

FIGURE 4.4 THE ORIGINALLY INTENDED SEQUENCE OF EVENTS

In order to develop the tool according to this sequence of events we have to place a limitation on the tool. As specified in section 4.1, a mapping model has to be created in the editor generation. This step includes quite a lot of decisions that influence the resulting editor. It is possible to automatically make these decisions, e.g. by using automatic recognition of links and nodes, based on construct types or naming of constructs. However, this would bound the tool to one or more specific ontology languages, making the tool not

suitable for other ontology languages and the formalisms behind them and thus making the tool less general. Since we want the tool to be general, we chose to avoid this limitation, and we introduced a step where the user has to make some decisions. Consequently, this also introduces a compromise on the level of automation of the process. This is a compromise with less negative impact than a compromise on the level of generality of the tool. The sequence of events according to this situation is depicted in Figure 4.5.

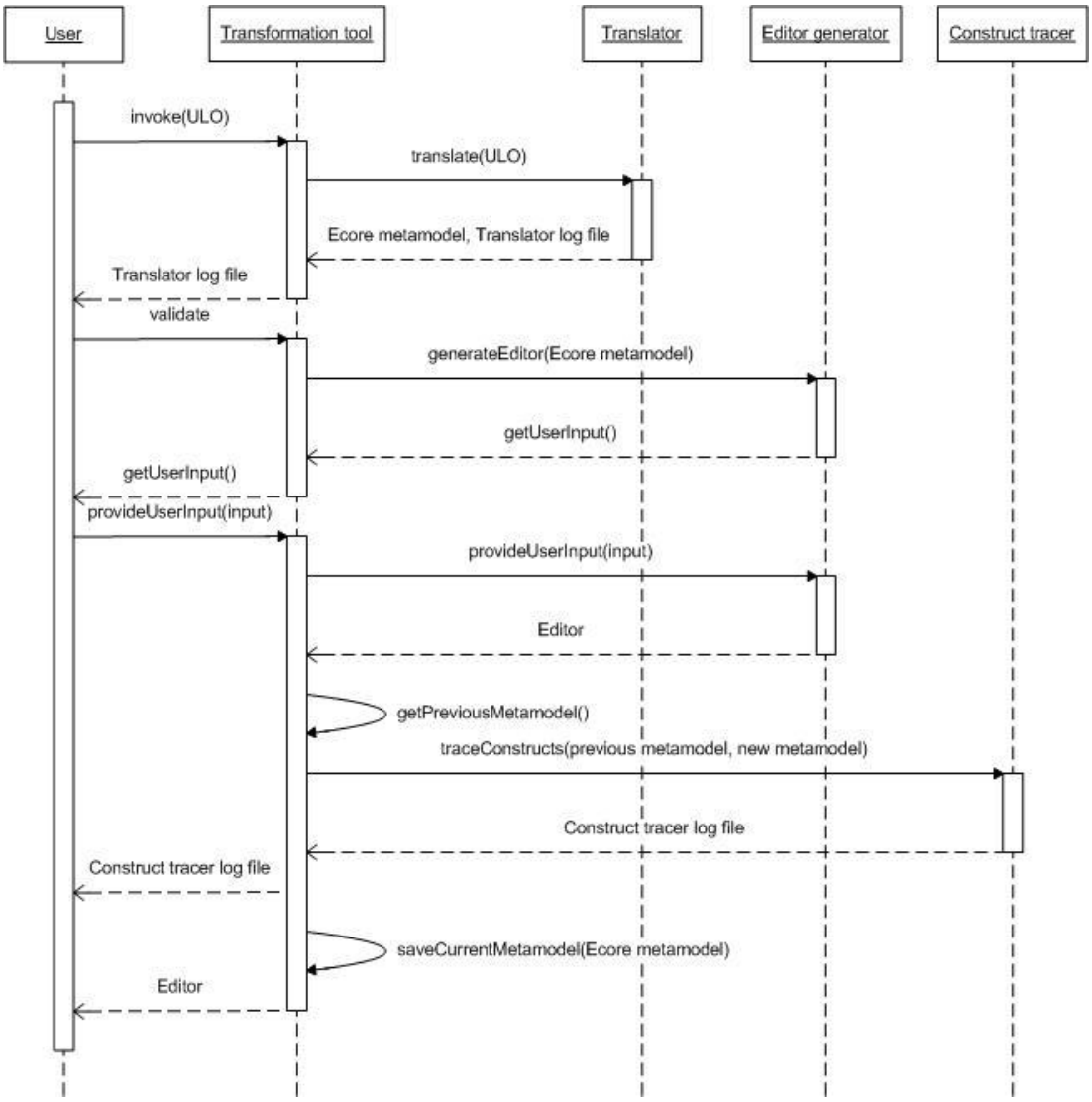

FIGURE 4.5 COMPROMISED SEQUENCE OF EVENTS

The sequence is almost the same as in the originally intended sequence, except that now the tool asks for user input during the generation of the editor. When the user has provided this input, the tool continues in the same way as it would in the originally sequence.

4.3 LANGUAGE MAPPINGS

The translator (Figure 4.1) translates the input model (the ULO) from the ontology language to the metalanguage (Ecore). To aid the translator in its task, the translations storage provides the translator with the intended set of translation rules. The translation rules define which concepts of the input model are translated into which concepts of the output model.

We have used the EMF4SW Eclipse plug-in [25] as a starting point for this mapping. This plug-in can currently translate from OWL to Ecore and vice versa, from OWL to UML and vice versa and from EMF Models to RDF and vice versa. Adding more translations to this plug-in will contribute to the generality of our transformation tool. Any ontology language can be supported in this way, as long as the translation rules are provided. However, some translations might be lossy, since a language might provide constructs that cannot be translated to Ecore. The EMF4SW plug-in uses the Atlas Transformation Language (ATL) [27] to specify the translation rules for the translation from one language to another. The set of rules provided by the EMF4SW Eclipse plug-in is not extensive enough for our research. We defined an extra set of translation rules that should be used on top of the set currently provided by EMF4SW. These extra rules are described below.

4.3.1 NON-DISJOINT SUBCLASSES

In a metamodel, all subclasses of a class are disjoint by definition. In an ontology, however, subclasses of a class are only disjoint if this is explicitly stated, otherwise they are not disjoint. This means that in an ontology an individual in a subclass might also be in another subclass. Therefore, translating a class with subclasses from an ontology language to a metalanguage is not straightforward. When the ontology states that the subclasses are disjoint, the mapping can be performed one-to-one. When this is not stated, a different translation has to be chosen. We have identified three approaches to translate this concept, as discussed below.

Introduce additional subclasses

The first approach is to introduce one or more additional subclasses, one for each of the possible combinations of subclasses an individual might be in. For example, given an ontology containing a class Game with two non-disjoint subclasses Cardgame and Dicegame, the generated metamodel should

contain a class Game with the subclasses Cardgame, Dicegame and CardgameAndDicegame.

The benefit of this approach is that there is a very clear mapping, which provides the correct information (i.e. a game is a card game, a dice game or both). The drawback of this approach is that the resulting DSL (generated from the created metamodel) will not recognize an instance from CardgameAndDicegame as just a Cardgame or as just a Dicegame, since it is both. Another drawback is that the number of extra classes grows exponentially when the number of non-disjoint subclasses grows. A translation of n subclasses in the ontology results in $2^n - 1$ subclasses in the metamodel. For n = 2 this results in 3 subclasses in the metamodel, but for n = 4 it is already 15 subclasses and n = 6 results in 63 subclasses.

Addition of equivalent objects reference

The second approach is to translate the classes one-to-one, adding a reference to the superclass providing the possibility to denote equivalent objects. In the example of the games, an individual that is in both Cardgame and Dicegame would be defined by two objects (a Cardgame and a Dicegame) with a reference to each other, saying that they are equivalent (even though they are different objects). The benefit of this approach is that it correctly models the concept as it was intended in the ontology, since it is possible to have an object that is an instance of multiple classes, modeled as different objects that are related. There are, however, many drawbacks. An object might need a lot of references to equivalent objects when there are many non-disjoint subclasses. If there are n non-disjoint subclasses, an object might need n-1 references. This does not improve the surveyability of the resulting metamodel. On top of that, the metamodel provides the possibility to link objects that should not be linked, so there should also be rules about when to use the reference and when not.

Multiple inheritance

The third approach uses the concept of multiple inheritance. This concept assumes that it is possible for a class to have multiple superclasses. With this concept we can introduce a new class for each possible combination of subclasses an individual might be in. The superclasses of such an introduced class are all (non-disjoint) subclasses that are part of that particular combination. In the example of the games, the generated metamodel should contain a class Game with the subclasses Cardgame and Dicegame.

Furthermore, there should be a class CardgameAndDicegame, which is a subclass of Cardgame as well as of Dicegame (Figure 4.6).

The benefit of this approach is that it correctly models the non-disjoint subclasses in the metamodel, however, there are major drawbacks. To start with, it has the same drawback as the first approach, regarding the exponentially growing number of introduced subclasses. However, a far more important drawback is that this concept causes problems at the technical level. Although the concept of multiple inheritance is appealing, there are very few languages that actually support this concept. Java, the language used by the EMF and GMF technology, for example, does not support multiple inheritance of classes. Ecore actually does support multiple inheritance, but since the editor is generated by EMF and GMF technology, we cannot use this option.

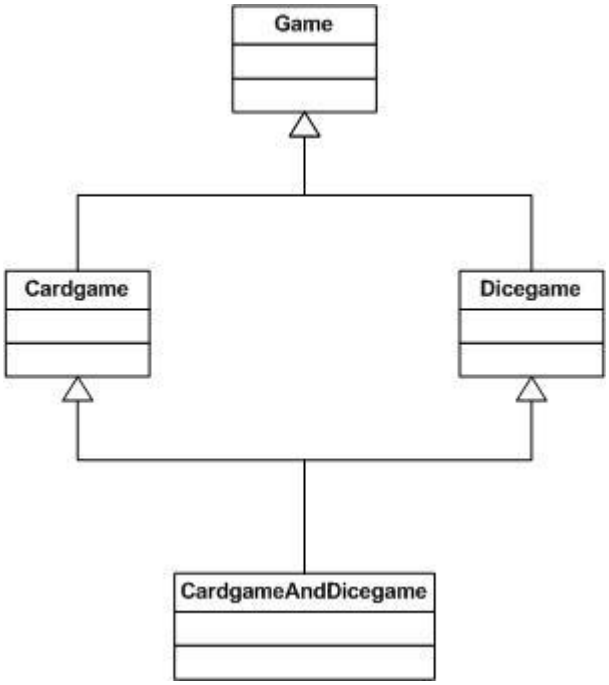

FIGURE 4.6 EXAMPLE OF MULTIPLE INHERITANCE

In our opinion the benefit of the second and third approach (correctly translating the non-disjoint subclasses) does not outweigh the drawbacks of these approaches. Therefore, our design uses the first approach. It is the responsibility of the language designer to validate the metamodel after translation.

4.3.2 CLASSES DECLARING EQUIVALENT CLASSES

In an ontology, a class with a listing of equivalent classes denotes a category. To show this, we provide an example ontology that contains 4 classes, namely the classes Male, Female and OnlyChild, which are subclasses of the class Person. The classes Male and Female are disjoint and there is a covering axiom on these classes regarding the class Person, i.e. and individual that is in the class Person must also be in either the class Male or Female. There are two object type properties: hasParent and hasSibling, both with domain and range Person. The class OnlyChild has a listing of equivalent classes that specifies that any individual that is in Person and has no hasSibling relations is an individual that is also in OnlyChild. When one translates this class to a metamodel class, errors are introduced. In the Person example, the class OnlyChild has superclass Person. The other subclasses of Person are Male and Female, which are disjoint. Translating this one-to-one would result in a metamodel in which the class Person has three subclasses: Male, Female and OnlyChild. This is not what was intended, since the class OnlyChild is only a category. A correct way to translate this class is to add a Boolean attribute to the superclass of the concerning class. In the Person example, this would result in a metamodel in which the class Person has the subclasses Male and Female and has a Boolean attribute OnlyChild.

4.3.3 CLASS COVERED BY ITS SUBCLASSES

When the subclasses of a class cover all the elements of the class, it means that all individuals that are in that class are also in (at least) one of its subclasses. Translated to a metamodel this means that there are no instances of that class that are not also an instance of one of its subclasses. In the Person example, the subclasses Male and Female are covering classes of the class Person. In the resulting metamodel it should not be possible to create an instance of the class Person, which is easily realized by making the class Person abstract. So a class that is covered by its subclasses in the ontology should be made abstract in the resulting metamodel.

4.4 PROTOTYPE

We selected parts of the design that we implemented in the prototype tool. This section discusses the our selection and development of the prototype tool.

4.4.1 FUNCTIONALITY SELECTION

Due to a limitation on the available time for this project we had to make a selection of the parts of the design to implement in the prototype. To make the tool fulfill all the requirements, the other parts will still have to be implemented.

The design describes three main parts of the transformation tool, which are the translator, the construct tracer and the editor generator. We decided not to implement the construct tracer, since the construct tracer is only useful if the other parts have been implemented. To only implement the construct tracer would not result in a prototype tool that has any functions, while being able to trace constructs is useless if there are no constructs to trace. Deciding not to implement this part means that we will not meet requirements R8 and R9 in this development iteration.

Furthermore, we decided to implement only a part of the translator, which translates the ULO to a metamodel. We decided to use the EMF4SW plug-in as a basis for our tool, however, for the prototype we decided to implement only one mapping, which is the mapping from OWL to Ecore. Our implementation of the translator uses the EMF4SW plug-in, however, the extra translation rules described in section 4.3 have not been implemented.

Since the intended output of the transformation tool is an editor, we decided that the editor generator is the most important part to implement. Therefore, we fully implemented this component. To sum up, we decided to implement the editor generator and a part of the translator. The construct tracer and the remaining part of the translator have not been implemented (see Table 4.1).

Component	**Implemented**
Translator	Partly
Construct tracer	No
Editor generator	Yes

TABLE 4.1 COMPONENTS IMPLEMENTED IN THE PROTOTYPE TOOL

4.4.2 OVERALL SOFTWARE ARCHITECTURE

We implemented the prototype tool as an Eclipse plug-in. This means that it can be installed in the Eclipse platform. Once this plug-in is installed, its functionality can be used. The plug-in contains 3 packages:

(1) nl.nijenhuiscf.editorgeneration

(2) nl.nijenhuiscf.editorgeneration.handlers

(3) nl.nijenhuiscf.editorgeneration.wizards

The first package contains the classes that take care of the functionality of the tool. These are the Translation class, the Generation class and the EmfRepository class. The Translation class is responsible for the translation of an ULO into a metamodel. The Generation class uses the EmfRepository class while executing the different steps of the generation of the graphical editor. Each step loads the necessary models in the beginning and saves the created models at the end of that step. This introduces some overhead, since a resource might be saved and closed in one step and loaded again in the next step. We accept this overhead, since this process of loading and saving resources greatly improves the extendibility of our tool. In this way, each step can be extended and adapted to the will of any developer. This decision also allows the fulfillment of requirement R7.

The second package contains the classes that handle the execution of the menu commands. The classes in this package are the EditorGenerationHandler class and the TranslationHandler class. When a menu item is selected, these classes execute the correct actions, i.e. they start the correct wizard.

The third package contains the classes that implement the wizards of this tool. It contains the classes TranslateULOToMMWizard and TranslationPage, which invoke the translation of the ULO into a metamodel. It also contains the CaptureEcoreInformationWizard class and the EcoreInformationPage class, which implement the wizard that gathers the required information for the editor generation and invokes the various steps of this process. Finally, it also contains the MyGMFMapGuideModelWizard class, which is an extension of the GMFMapGuideModelWizard class. This wizard is used to provide the user with the possibility to make some choices before generating the mapping model. The use of this wizard keeps the tool general, as is discussed in section 4.2.

Each plug-in contains a plug-in manifest file. This file is the most important file of each plug-in, since it contains all the important information about the plug-in, e.g. its dependencies and Eclipse extensions implemented in the plug-in. The plug-in manifest file of our tool contains the extensions for our

tool, which add a category, named “Editor Generation”, to the menu bar of the Eclipse platform. The category contains two menu items, named “Translate” and “Generate”, which invoke the corresponding handlers when selected. These handlers make sure that the correct actions are executed. The items “Translate” and “Generate” are also added to the toolbar and for each of them a hot key is defined.

4.4.3 TRANSLATOR

In our prototype we only use one mapping, namely from OWL to Ecore. The translator can be invoked by selecting a “Translate” item or using the hot key. The wizard (Figure 4.7) asks the user to provide the ULO file (which has to be an OWL file in the prototype). The wizard instantiates the Translation class and then runs the translation with the provided ULO file as input. The tool uses the OWL2Ecore functionality of the EMF4SW plug-in to translate the ULO (OWL file) to a metamodel (Ecore file). It starts by setting the correct options for the transformation and loading the input model, i.e. the ontology. Then it invokes the actual transformation, which results in an output model, i.e. the metamodel. This output model is saved in a new resource.

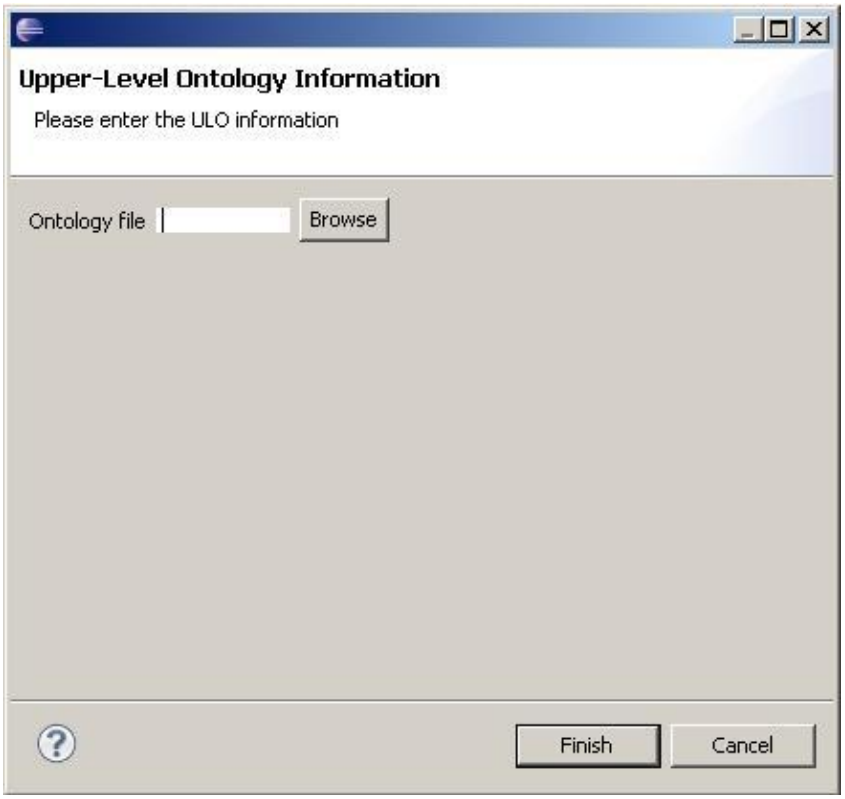

FIGURE 4.7 WIZARD FOR THE TRANSLATOR

4.4.4 EDITOR GENERATOR

We chose to implement a separate wizard for the editor generation, because of two reasons: (i) we did not implement the additional translation rules in the translator, so we cannot guarantee that the metamodel resulting from the translation from ULO to metamodel is correct; (ii) the language designer will have to check whether the translation was correctly executed or not, even if the additional translation rules have been implemented. Figure 4.5 shows that after the translation the user needs to validate the result. Therefore we implemented a separate wizard for the editor generation, so that the user can first translate the ULO into a metamodel, then manually adapt the metamodel if needed and use the next wizard to start the editor generation. Once the remaining parts of the translator are also implemented, the resulting diagnostics file can be analyzed (possibly with tool support) to validate the translation.

The wizard for the editor generation (Figure 4.8) asks the user to provide some information, prior to generation. The user has to specify the input model, which is the generated (and possibly adapted) Ecore model. Some additional information is also needed, concerning information that is not present in the model file and thus cannot be derived from this file. The user has to specify the base package, the prefix, the model plug-in ID, the model directory and the compliance level. Once the user has provided this information and hits the finish button, the wizard invokes the editor generator, which starts by generating a domain generator model. The EMF toolkit is used to do this. This toolkit contains a class GenModelFactory, which can create a domain generator model resource. We use this factory to create a resource and then we add the information from the model file to it. After initializing the domain generator model we add the information specified by the user to this model. When the domain generator model is completely finished the tool saves the resource in the file system. At this point we can see what the impact of our decision (see section 4.4.2) to save and load resources for each step is. For the next step we need to load the domain generator model resource again, so it would be easier to just keep it open. However, by closing and reopening it in each step, we enable our tool to be extended at this point. It is now possible, for instance, to skip the generation of the domain generator model or to use another method or tool for this. This can be done by extending our tool and writing a method that overrides our method.

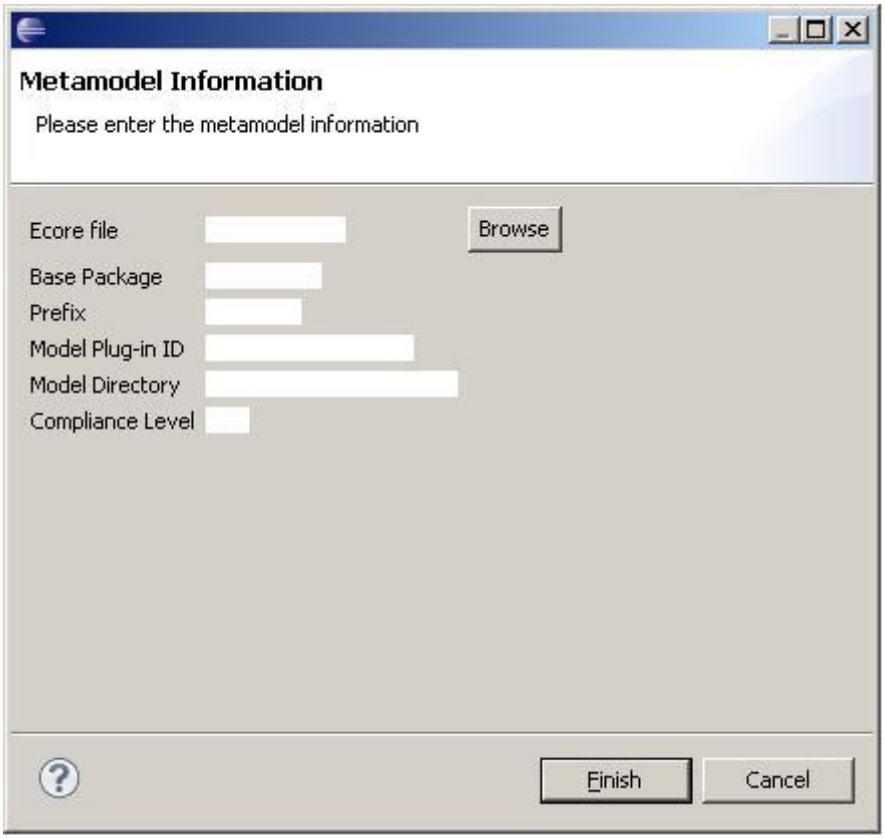

FIGURE 4.8 WIZARD FOR THE EDITOR GENERATOR

The next step executed by the tool is the generation of the domain code. The EMF toolkit contains a class Generator, which performs this task. The tool loads the domain generator model resource again and then sets this model as input for the Generator. The Generator is used to generate four kinds of domain code (model code, edit code, editor code and tests code), resulting in four new plug-ins.

The EMF part of the editor generator is now finished. The generated code can already be used as a tree-based editor, if the newly generated plug-ins are exported. Our tool continues the generation process, however, since we want to generate a graphical editor. The next artifact to be generated is the graphical definition model. This model defines the graphical elements that are to be used in the resulting editor. The GMF toolkit provides a GraphDefBuilder class, which provides the means to create a graphical definition model. We load the domain model resource again and provide the contents of this resource as input to the process method of the GraphDefBuilder class. After that we create a new resource and save the generated graphical definition model in this resource.

The creation of the tooling definition model happens in the same way as the creation of the graphical definition model. We use the class ToolDefBuilder, provided by the GMF toolkit, with the domain model as input and this results in a tooling definition model, which we save in a new resource.

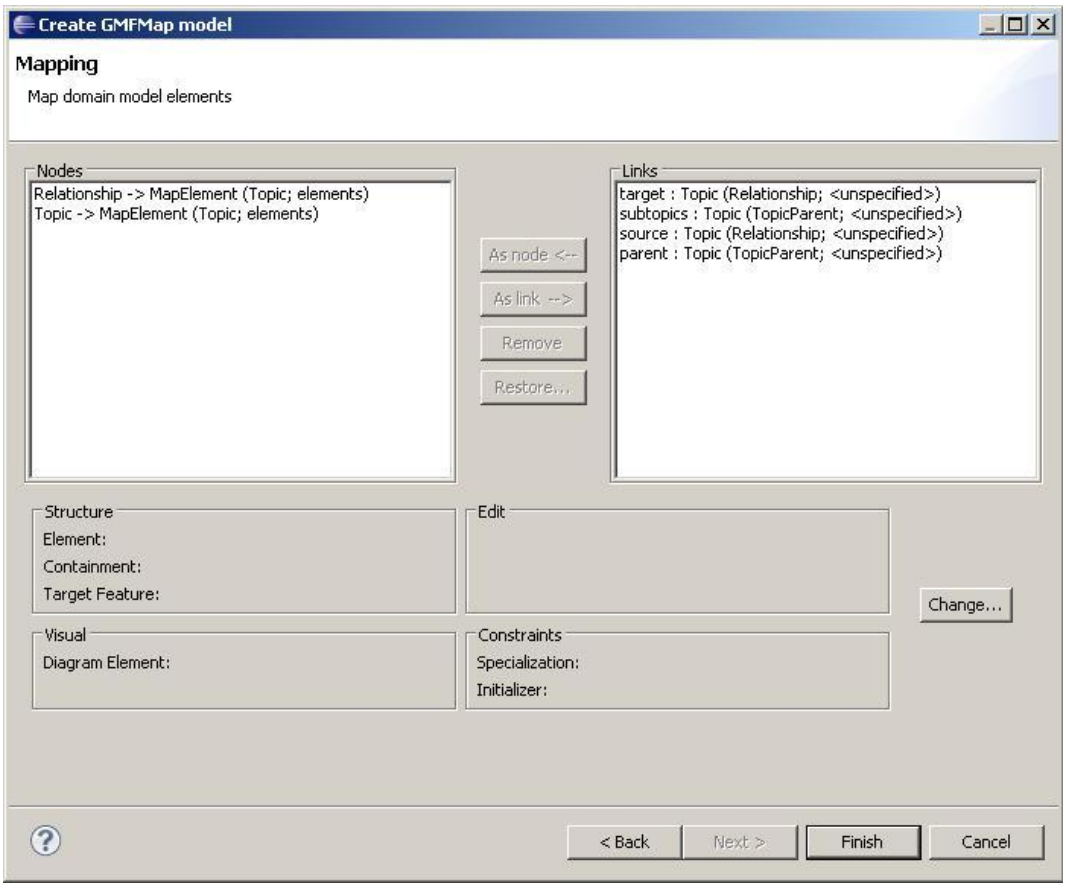

FIGURE 4.9 MYGMFMAPGUIDEWIZARD

At this point some choices have to be made by the user. The tool starts up the MyGMFMapGuideWizard (Figure 4.9), which extends the GMFMapGuideWizard, provided by the GMF toolkit. The MyGMFMapGuideWizard actually just uses the GMFMapGuideWizard, but it skips the pages that ask for information that the tool already knows, i.e. the domain model, the graphical definition model and the tooling definition model, so the wizard automatically selects these models. However, the wizard provides the user with the possibility to go back and change the automatically selected information. The wizard allows the user to make some choices and waits until the user hits the finish button, after which the mapping model is created and saved in the file system.

Once the mapping model is created, the tool continues by creating the diagram editor generator model. The input needed for this operation consists of the domain generator model and the mapping model, so the tool starts with loading these resources. To create the diagram editor generator model we use the TransformToGenModelOperation class and the TransformOptions class from the GMF toolkit. We set the correct options, load the mapping model and the domain generator model from the resources and then execute the transformation. The result is a diagram editor generator model.

Finally, we need to generate the graphical editor code. The GMF toolkit has a Generator class for this task. The tool loads the diagram editor generator model resource and uses it as input for the Generator. After running the Generator, the graphical editor code has been generated as a new plug-in.

After exporting all the plug-ins the tool has created, we can install them in Eclipse. At this point the graphical editor is ready to be used to model domain ontologies.

5 EVALUATION OF THE PROTOTYPE

This chapter evaluates the prototype tool that was developed within this project. First we provide the criteria against which we evaluated the prototype. These criteria are presented in section 5.1. After that, in section 5.2, we discuss the evaluation itself and the steps we have taken to perform this evaluation. Finally, section 5.3 discusses the results of the evaluation.

5.1 EVALUATION CRITERIA

We derived the criteria for this evaluation from the requirements we presented in section 3.3. By using the requirements as criteria for the evaluation we can on the one hand evaluate the prototype against useful criteria and on the other hand check whether the requirements of the project are met. These requirements provide criteria on data, functionality of the prototype and quality of the prototype. The requirements are presented in Table 5.1.

Number	**Requirement**
R1	The transformation tool should accept an ULO as input.
R2	The transformation tool should generate as output an editor to be used by the domain specialist.
R3	The transformation tool should generate a DSL from the ULO.
R4	The generated DSL should allow domain ontologies to be described.
R5	The transformation tool should allow the ULO to be specified in any ontology language.
R6	The editor should contain functions to add, load and save a domain ontology.
R7	The editor should be extendable.
R8	The transformation tool should provide traceability from the changes in the ULO to domain ontologies.
R9	The transformation tool should provide traceability from the ULO to the DSL.
R10	The generated DSL should comply as much as possible with the ULO given as input.

TABLE 5.1 REQUIREMENTS

To be able to use these requirements as criteria, we have to make them measurable, i.e. we have to be able to assign a value, e.g. a percentage to it. Based on that value a stakeholder can determine whether the requirement is met or not. However, most requirements are hard to quantify. We could simply assign a Boolean value (True or False) to it, but that would not reflect the level of achievement of the requirements. We decided to discuss how each requirement is met and then discuss the limitations and the level of fulfillment.

Some requirements in Table 5.1 are treated differently, though. Requirements R8 and R9 are requirements on functionality that has not been implemented in the prototype tool. It would be pointless to evaluate the prototype tool against these requirements. Requirement R5 specifies that the transformation tool should allow the ULO to be specified in any ontology language. The design of the tool allows extra translation rules to be added to the repository of the translator. This provides the option to add these for any ontology language, which means that any language can be supported, given that translation rules can be specified for it. However, to measure the requirement we can decide to judge the tool according to the number of ontology languages it accepts at the time of the evaluation. By assigning an integer value to this requirement we make this requirement measurable. Requirement R10 specifies that the generated DSL should comply with the ULO given as input. To measure this we need to specify the level in which the DSL is directly generated from the ULO. After translating the ULO into a metamodel, the metamodel may have to be adapted by the user before the prototype can continue with the editor generation. If we specify which percentage of the metamodel, resulting from the translation, has to be adapted, then we can make the decision of whether the requirement is fulfilled or not based on this percentage. We specify this percentage by counting the number of classes, attributes and references that have to be removed or added and then dividing that number by the total number of classes, attributes and references.

By making the requirements measurable we defined the criteria against which we can evaluate the prototype. These criteria and the corresponding requirements are listed in Table 5.2.

Criterion	Corresponds to	Value
C1	R1	Discussion on how the transformation tool accepts an ULO as input.
C2	R2	Discussion on how the transformation tool generates as output an editor to be used by the domain specialist.
C3	R3	Discussion on how the transformation tool generates a DSL from the ULO.
C4	R4	Discussion on how the generated DSL allows domain ontologies to be described.
C5	R5	Integer: the number of ontology languages that can be used to specify an ULO at the time of evaluation.
C6	R6	Discussion on how the editor contains functions to add, load and save a domain ontology.
C7	R7	Discussion on how extendable the editor is.
C8	R10	Percentage: the number of classes, attributes and references that need to be added or removed divided by the total number of classes, attributes and references.

TABLE 5.2 CRITERIA FOR THE EVALUATION

5.2 PROCEDURE

We performed the evaluation of the prototype tool by running the tool with an example ontology as input. The example ontology is based on the use case presented in section 3.2, which is an ULO for the mind map language. This section describes the output of the tool and all the steps we took. We evaluated the tool against the criteria defined in section 5.1. The results of the evaluation are discussed in section 5.3.

Our transformation tool is a plug-in for Eclipse. After installation of the plug-in we can use the tool. To evaluate the tool we created a new project. We copied the *Mindmap.owl* file, containing our mind map ULO, to the *src* folder of the project. Then we selected the Translate item from the menu (Figure 5.1). The tool starts the wizard for the translation part of the tool (Figure 4.7).

In this wizard we used the browse button to select the OWL file, and we hit the finish button.

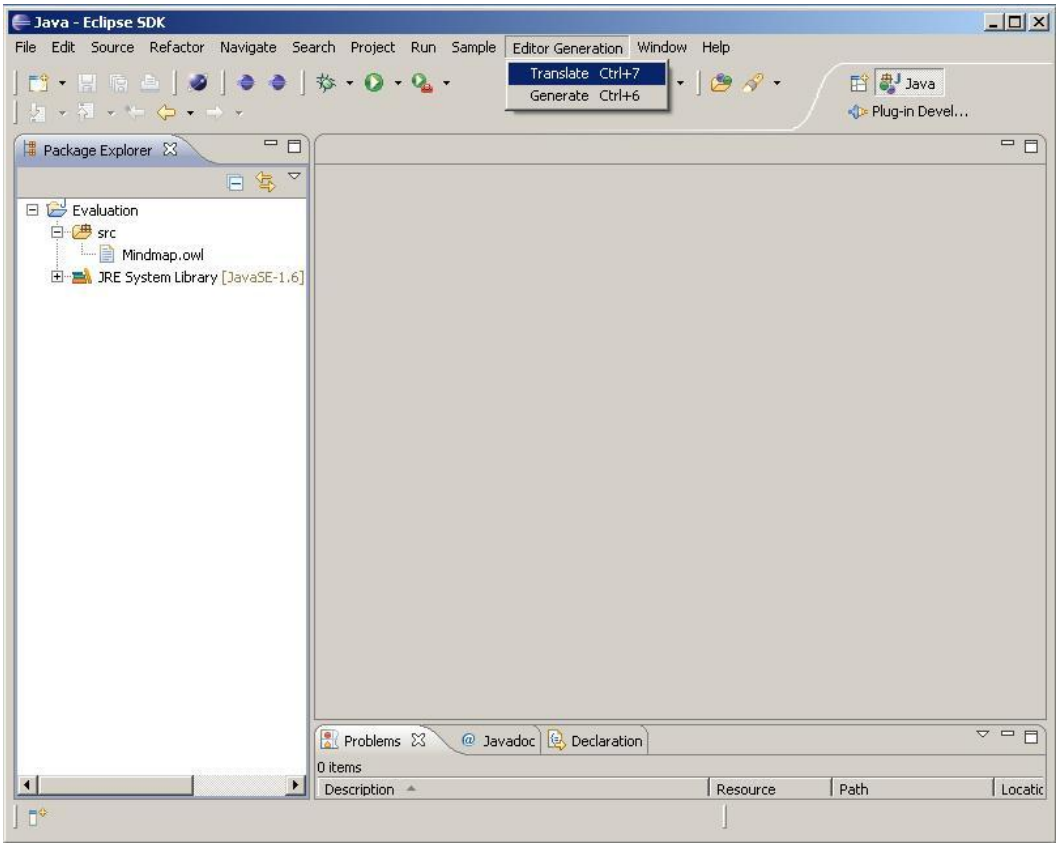

FIGURE 5.1 SELECTING THE TRANSLATE ITEM FROM THE MENU

The tool invoked the translation and when it finished it had created the file *Mindmap.ecore*. We generated an Ecore diagram for this file and then we could inspect the generated model. This diagram is depicted in Figure 5.2, in which we removed six references with the name *bottomObjectProperty*, to make it more readable.

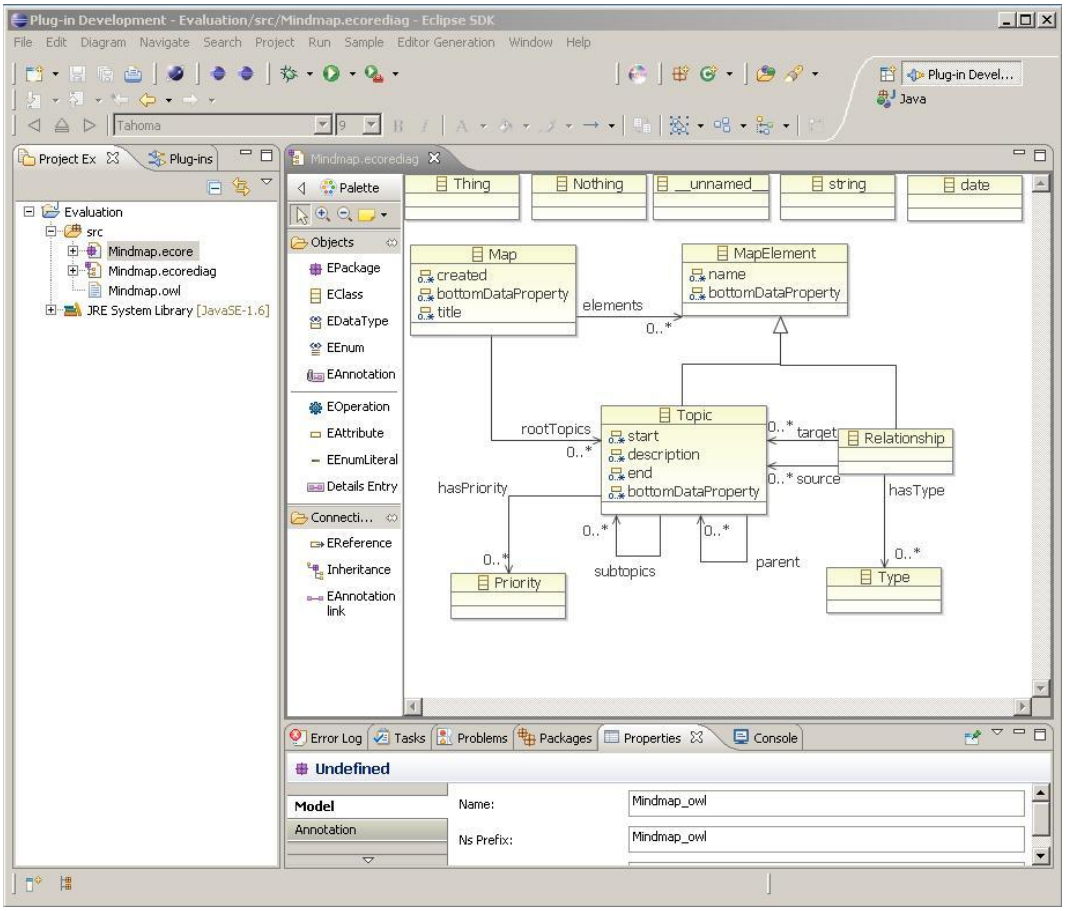

FIGURE 5.2 ECORE MODEL RESULTING FROM THE TRANSLATION

At this point we had to adapt the generated model to validate its correctness. The classes *Thing* and *Nothing* have been generated (see Figure 5.2). These are predefined class identifiers in OWL (owl:Thing and owl:Nothing), but in the Ecore model they should be removed. The classes d*ate* and s*tring* are also present in the generated model, which are the ranges of the data type properties in the *Mindmap.owl* file. These should not be defined in the Ecore model as explicit classes. Furthermore, there is a class with the name *_unnamed_* in the Ecore model, which is the result of the covering axiom that we defined on Topic and Relationship. In OWL, an extra (unnamed) class is introduced to denote that an inherited superclass of *MapElement* is the union of Topic and Relationship. In the Ecore model we can simply make the class *MapElement* abstract in order to achieve the same result. All these classes (*Thing*, *Nothing*, *date*, *string* and *_unnamed_*) have been removed from the Ecore model. The Ecore Model also contained six references with the name *bottomObjectProperty* that have been removed before, and also three attributes with the name *bottomDataProperty*, shown in Figure 5.2. These

properties are part of OWL and are used to validate an ontology. These properties are not used in the metamodel and have been removed. The classes Priority and Type were intended to be enumerations, however, they were not generated like that. We removed them and replaced them with enumerations, also adding an attribute *priority : Priority* to the class Topic and an attribute *type : Type* to the class Relationship. The reference *elements* should be a containment. We already knew that this would not be generated correctly, since a containment cannot be explicitly defined in OWL. Therefore we changed this reference to be a containment. The references parent and subtopics have been generated almost correctly. They are supposed to be EOpposites, which is not the case, so we had to adapt that. Finally, we had to change some cardinalities. All cardinalities have been generated as 0..*, which was not intended. We changed the cardinalities accordingly.

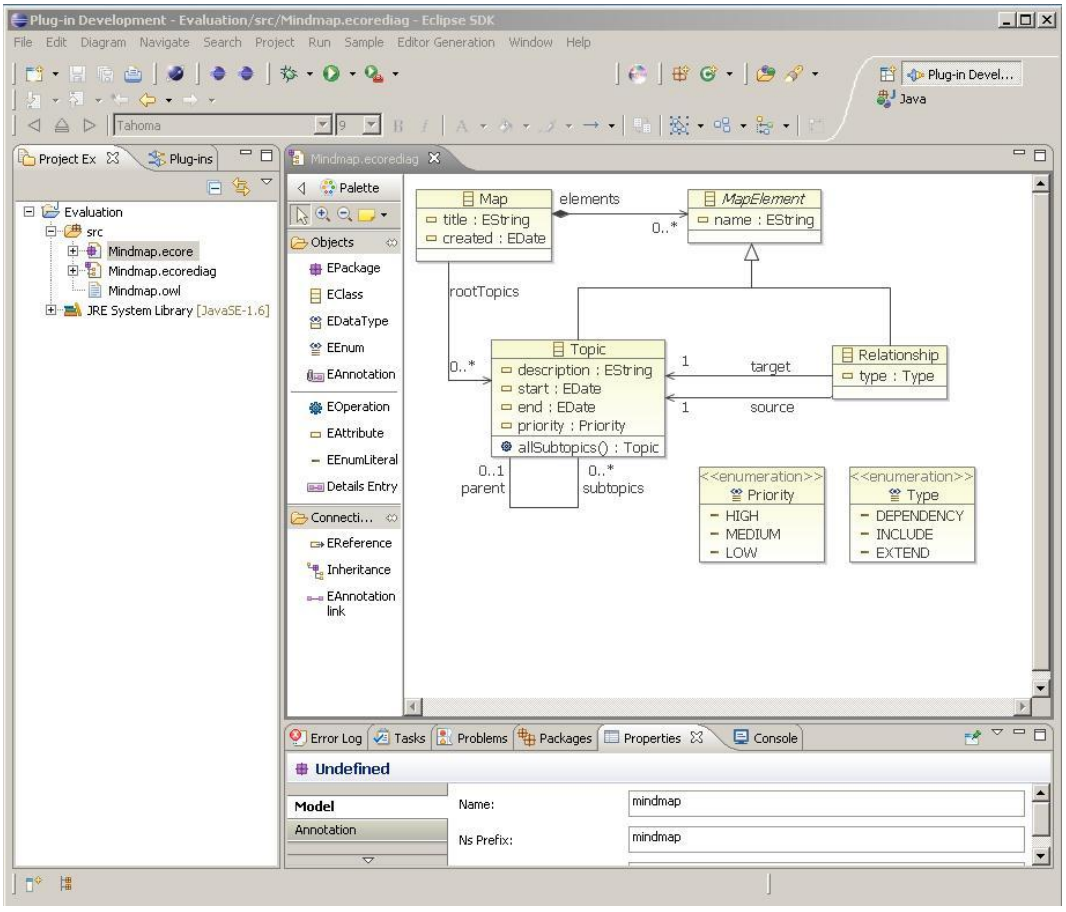

FIGURE 5.3 ECORE MODEL AFTER THE CHANGES

The resulting Ecore model, after the changes, is depicted in Figure 5.3. It is quite different from the generated model, however, a major part of the

adaptations changed predefined OWL concepts, like the classes *Thing* and *Nothing* or the properties *bottomObjectProperty* and *bottomDataProperty*. The translation rules can be changed in the future so that they can handle these concepts.

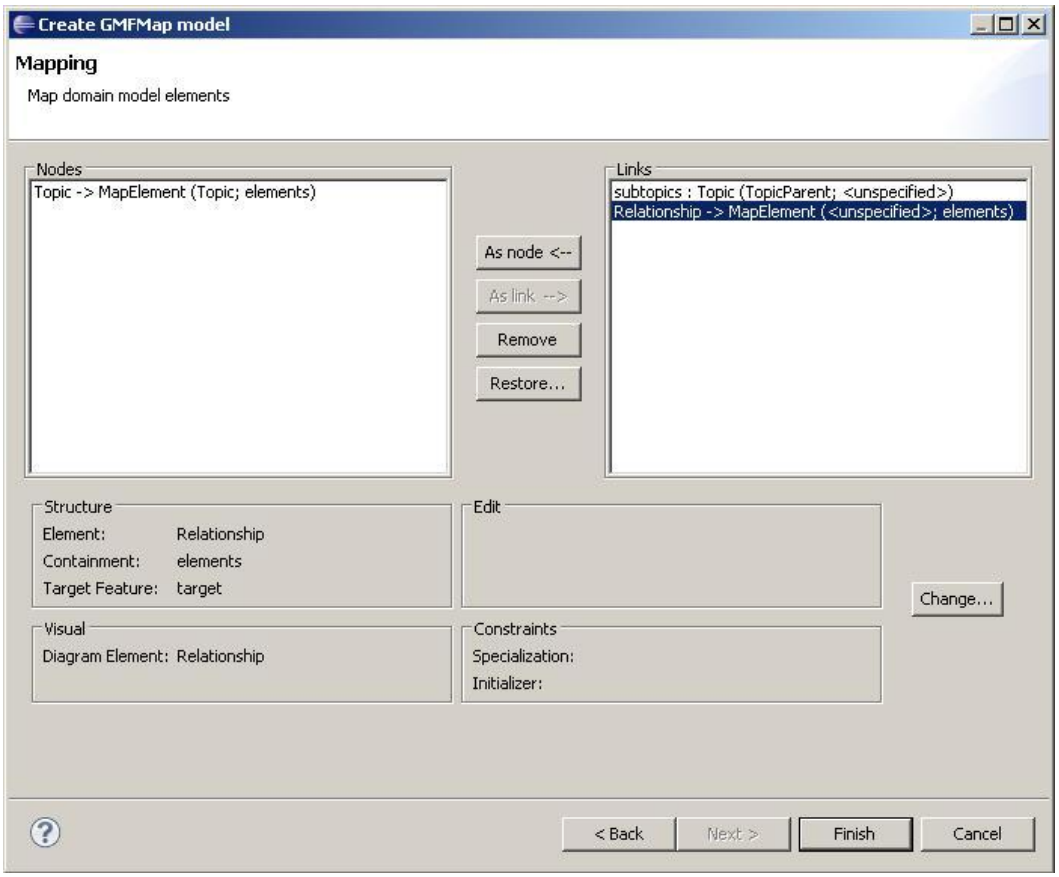

FIGURE 5.4 MYGMFMAPGUIDEWIZARD WITH ADAPTED INFORMATION

After we adapted the metamodel, we invoked the wizard of the editor generation part of the tool (see Figure 4.8), we provided the required information, and we hit the finish button. After a few seconds the tool provided another wizard (Figure 4.9). We adapted the information in the wizard like shown in Figure 5.4 and we hit the finish button again. The tool finished its work and the project explorer showed the generated files and projects (Figure 5.5). These projects provide the plug-ins for the graphical mind map editor. After installing the plug-ins we created a new mind map in the newly created editor (Figure 5.6).

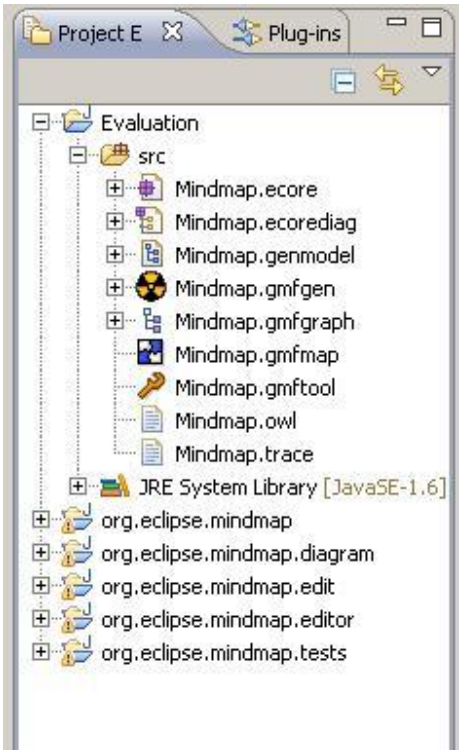

FIGURE 5.5 PROJECT EXPLORER DISPLAYING THE NEWLY GENERATED FILES AND PROJECTS

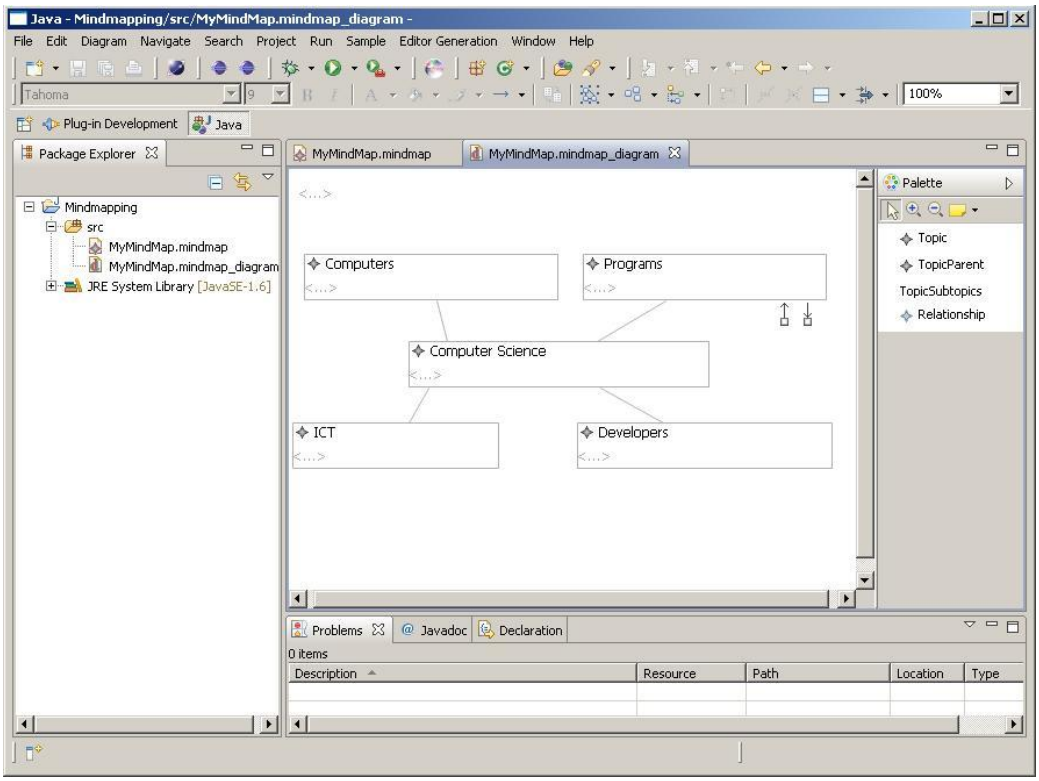

FIGURE 5.6 NEW GRAPHICAL MIND MAP EDITOR

This section discusses the results of the evaluation of the prototype tool against the criteria. We do not present a judgment of each result, however, we discuss the flaws we observed.

C1: The transformation tool should accept an ULO as input

In our evaluation we used the mind map ULO as input to our transformation tool. The tool accepted the input, which means that the criteria is met. Since our prototype only provides a translation from OWL to Ecore, there is a limitation on the language in which an ULO can be specified, since currently it can only be specified in OWL.

C2: The transformation tool should generate as output an editor to be used by the domain specialist

The editor that was generated by the tool is shown in Figure 5.6. This editor can be used by the domain specialist, so we can say that also this criterion is met. Between translation of the ULO to a metamodel and the generation of the editor, however, we had to adapt the metamodel, indicating that there is a limitation to the level of automation. This situation can be improved by improving the set of translation rules.

C3: The transformation tool should generate a DSL from the ULO

The language that is used by the editor (Figure 5.6) originates from the input file: the mind map ULO. This indicates that the criterion is met. We do have to question the level of fulfillment, however, since we had to adapt the resulting metamodel. The level in which we had to adapt the metamodel is evaluated in the discussion of criterion C8. Still the basics of the DSL are taken from the ULO, so it is fair to say that the criterion is met, although the level of fulfillment is limited.

C4: The generated DSL should allow domain ontologies to be described

The mind map models that can be created with the generated editor are the domain ontologies in our evaluation. The editor provides the possibility to create and maintain these mind map models in some language. This language is the generated DSL. This means that the generated DSL is capable of expressing domain ontologies, indicating that the requirement is met.

C5: The transformation tool should allow the ULO to be specified in any ontology language

This criterion should be measured by counting the number of ontology languages that can be used to specify an ULO at the time of evaluation. The tool only allows ULO's to be represented in OWL, so we assigned the Integer value 1 to criterion C5. This situation can be improved by adding translation rules for more languages to the repository of the translator.

C6: The editor should contain functions to add, load and save a domain ontology

In the evaluation we also used the editor that was generated (Figure 5.6) and we created (add function) and saved (save function) a mind map model, named MyMindMap. The project explorer displays the files, providing the possibilities to close and open (load function) them whenever necessary. The editor provides the functions add, load and save, so we can say that this criterion is fully met.

C7: The editor should be extendable

This criterion cannot be quantified solely based on the evaluation of the prototype tool. In section 5.1 we defined that the editor is extendable if the transformation tool itself is extendable. However, it is not possible to determine whether the tool is extendable based on only the evaluation. To discuss this criterion we need to take a look at some characteristics of the tool code. We concentrate on the editor generation part, since we want to determine whether the editor is extendable or not. The editor generator performs various steps, defined in various methods. These methods use resources, e.g. the Ecore model, stored in a .ecore-file in the file system. We could have loaded the resources once and then used them where we needed them. However, we chose to make every method load the resources itself and save and close them again when the method is finished. By doing this, we provided the possibility to extend the tool and change the workflow of the tool, e.g. by skipping a method or adding a method. This might not have been possible if we did not load and close the resources in each method, e.g. skipping a method would have resulted in skipping the loading of a resource, which might have caused subsequent methods to fail. By making this design decision, we provided the possibility to extend the editor generator at will, as long as one does not try to load resources that do not exist yet. For this reason we consider that criterion C7 has been met by the prototype tool.

C8: The generated DSL should comply as much as possible with the ULO given as input

The value to be provided for this criterion is defined to be the percentage of classes, attributes and references that need to be added or removed to make the generated DSL comply to our the ULO. This value is the result of one experiment and cannot be used for general conclusions, but it is an opportunity to see if there are unexpected results. In our evaluation it turned out that the cardinalities where not correctly translated from the ULO. All cardinalities were set to 0..* in the metamodel. We did not include the changes we had to make in the cardinalities in the calculation of the percentage, since cardinalities were not defined to be included in the calculation in section 5.1. The generated metamodel contained 11 classes, 9 attributes and 16 references (considering generalizations as references). We removed 7 classes, added 2 classes (enumerations), removed 3 attributes, added 2 attributes, removed 8 references and changed 2 references. This results in a percentage that had to be changed of 66,7%. If we include cardinalities in the calculation we get a percentage of 72,2%. In section 5.2, we indicated the predefined OWL concepts that should not be in the generated Ecore model. If we ignore the predefined OWL concepts, we get a percentage of 44%. The results of the evaluation are listed in Table 5.3.

Criterion	Description	Value
C1	The transformation tool should accept an ULO as input.	Fulfilled
C2	The transformation tool should generate as output an editor to be used by the domain specialist.	Fulfilled
C3	The transformation tool should generate a DSL from the ULO.	Fulfilled
C4	The generated DSL should allow domain ontologies to be described.	Fulfilled
C5	The transformation tool should allow the ULO to be specified in any ontology language.	1
C6	The editor should contain functions to add, load and save a domain ontology.	Fulfilled
C7	The editor should be extendable.	Fulfilled
C8	The generated DSL should comply as much as possible with the ULO given as input.	66,7%

TABLE 5.3 RESULTS OF THE EVALUATION

Aside from the predefined OWL concepts, we noticed another big influence on the correctness of the translation. Some constructs have been translated into the correct counterparts, but their properties were lost. Examples are the cardinalities of all references and attributes, the enumerated classes that are translated into normal classes, and the object type properties subtopics and parent, which are inverse properties of each other, which are translated into references, but are not EOpposites of each other. This indicates that the translation rules address the right constructs, but do not handle them totally correctly.

Containments cannot be explicitly defined in OWL, but we could represent them by using extra constraints in an additional language, e.g. SWRL [28]. Since our prototype only provides translation from OWL to Ecore, we could not use an extra language in our evaluation. Other ontology languages may be able to represent containments. In any case, for a given ontology language there should be a translation rule that translates a construct (or a combination of constructs and/or constraints) into a containment, or the language designer should be aware that containments cannot be modeled explicitly. There is one other solution to this problem, namely by interpreting all object type properties with the name "contains" as containments. However, this solution introduces a usage rule for the language designer, which does not allow him to use the name "contains" for other object type properties or use other names for containments. In case of our prototype tool, with only translation from OWL to Ecore and without the usage rule, we cannot provide the possibility to model containments in the ULO. This is a limitation of our tool.

6 FINAL REMARKS

This chapter identifies and briefly discusses some research, related to ours that has been done. Afterwards it presents the contributions of our work and draws our main conclusions. Furthermore, it discusses which points require further investigations. Section 6.1 elaborates on related work, section 6.2 presents the conclusions and the contributions of our work and section 6.3 discusses future work.

6.1 RELATED WORK

Assman et al [29] present a megamodel of ontology-aware Model Driven Engineering, in which descriptive ontologies and prescriptive models are combined in the OMG modeling infrastructure. Although ontologies and models are combined in this work, it does not provide means to translate between them. However, they do provide traceability from the ontology side to the model side and vice versa. Unfortunately, these traces are not valuable to our work, since we are interested in traces from ontology concepts to the concepts they are translated to (which are available by following the translation rules) and in traces between existing domain models and the new language, which are on the same side of the proposed megamodel. Bezivin et al [30] tried bridging model engineering and ontology engineering, however, they did it on the M3 level. In our work we are interested in translating from an ontology to a (meta)model on the M2 level. Hillairet developed the EMF4SW plug-in [25], which bridges the gap between EMF and some Semantic Web modeling languages. This plug-in is useful to our research, however, while the framework is fine, the translation itself could be improved. We used this plug-in as a basis for our translation. The EMF and GMF technology is suitable for generating graphical editors, however, there is no tool that automates this process. We used these technologies to generate an editor, but for our research that was not enough, since we wanted it to be done automatically.

6.2 GENERAL CONCLUSIONS

The objective of our research was to provide an architecture for automatic generation of tool support for domain specialists performing modeling tasks and to evaluate this architecture by means of a prototype tool. We performed our investigations in the scope of the CASP, which contemplates the existence

of domain models. This framework needed an editor to create and maintain these domain models. We proposed a tool that automatically transforms an ULO into an editor that is capable of creating and maintaining these models.

As a part of our requirements analysis, we performed a stakeholder analysis, identifying two main stakeholders: (i) the language designer and (ii) the domain specialist. The language designer is responsible for delivering the ULO that describes the language to be used for modeling domains. The domain specialist is the intended user of the resulting editor. We specified requirements for the tool, with the emphasis on some key aspects: the tool should be general and automatic and it should provide traceability of constructs to be used by both the language designer and the domain specialist.

We presented the design of our tool, which has three main components: (i) the translator, (ii) the editor generator and (iii) the construct tracer. The translator takes care of the translation from ULO to metamodel and of the tracing of constructs between the ULO and the metamodel. The editor generator is responsible for generating an editor from the metamodel. The construct tracer takes care of tracing constructs of domain ontologies, created with a previous version of the editor, to constructs in the new version of the editor, when the editor is updated. Each of these components takes care of a part of the key aspects of our tool. The translator should ensure that the tool is general and provide traceability to the language designer, the editor generator is responsible for making the tool work automatically and the construct tracer is responsible for providing the traceability to the domain specialist.

We evaluated our architecture by means of a prototype tool. For this prototype we have selected some parts of the design to implement. These parts were the editor generator and part of the translator. The construct tracer has been left out of the prototype tool. The part of the translator that has been implemented is only the use of the EMF4SW plug-in, which is the basis for our translator. We evaluated our prototype tool against the requirements. To be able to use our requirements as criteria for the evaluation, we tried to make them measurable. This was achieved by assigning a value to each of them.

The evaluation was performed by running the prototype tool with an example ULO (the mind map ULO) and documenting its output. By analyzing the

output, we could assign values to the requirements and present the results of the evaluation. The graphical editor generated by the prototype tool provided the possibility to model mind maps. This result shows the contribution of our work, on the one hand in the scope of the CASP and on the other hand in automating the editor generation. We can (semi-)automatically generate tool support for domain specialists performing modeling tasks. However, the evaluation also showed that some parts of the architecture require some more attention. Since we only implemented part of the translator, we could not expect the translation from ULO to metamodel to be fully correct, but we found that the translation also showed some unexpected inaccuracies. The current translation rules seem to be correct for the general structure, but some details are missing. Our prototype tool only provides translation from OWL to Ecore, which should be extended to multiple translations when implementing the whole tool. Each translation requires translation rules of its own. The inaccuracies we experienced in the evaluation just indicate that the translation rules for the translation from OWL to Ecore have to be reviewed.

This thesis indicates that our architecture provides the means to automatically generate tool support for specialists modeling domains. Although this research is done in the scope of the CASP, the proposed architecture can be used for other goals as well, e.g. in our evaluation we generated a mind map editor.

6.3 FUTURE WORK

The architecture we provided leaves room for further investigation. In this paragraph we summarize the issues that can be tackled:

- The needs of the domain specialist are not investigated extensively yet. By knowing the needs of the domain specialist regarding the tool support, more suitable functionality can be assigned to the generated editor. For example, the domain specialist may need functionality to validate models.
- The editor generated by the transformation tool should communicate with the CASP through the domain specialist interface. To save and load domain ontologies in the Ontology Registry of the CASP, communication with the domain specialist interface also is needed. The transformation tool also needs to communicate with the Ontology

Registry to be able to trace constructs used in the domain ontologies. Further investigation should determine the best way to establish this communication.

- Domain ontologies created with the generated editor are represented as Ecore models. Research on translating these models back to an ontology language is needed.
- Implementation of the whole transformation tool was not feasible within this research project. The development of the translator should be finished. The construct tracer should also be implemented.
- The translation rules for translating ontologies into metamodels need more research. We used the EMF4SW plug-in as a basis for our tool. Although the framework for translations this plug-in provides was easy to use and implement, the translation needs improvement. For example, the cardinalities of references, resulting from the translation, were all set to 0..* instead of the correct values. By improving these translation rules and describing them, it may be easier to define translation rules for other translations.
- To contribute to the generality of the tool, sets of translation rules for more ontology languages should be defined.
- To implement the construct tracer a method to compare the metamodels is needed. Investigation to existing techniques and tools in model comparison is needed to check whether an existing tool should be used or a new tool should be developed.

REFERENCES

[1] A.H. Doan, J. Madhaven, P. Domingos, and A. Halevy, "Learning to map between Ontologies on the Semantic Web," in *Proceedings of WWW2002*, Honolulu, Hawaii, USA, May 7-11, 2002.

[2] D. Fensel, I. Horrocks, F. van Harmelen, D. McGuinness, and P.F. Patel-Schneider, "OIL: Ontology Infrastructure to Enable the Semantic Web," *IEEE Intelligent Systems*, no. 16, 2001.

[3] T. Berners-Lee, J. Hendler, and O. Lassila, "The Semantic Web," *Scientific American*, May 2001.

[4] Anind K. Dey and Gregory D. Abowd, "Towards a Better Understanding of Context and Context-Awareness," in *HUC '99: Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing*, Karlsruhe, Germany, September 27-29, 1999, pp. 304-307.

[5] G.H. Mealy, "Another Look at Data," in *Proceedings of the Fall Joint Computer Conference*, Anaheim, California, USA, 1967, pp. 525-534.

[6] B. Chandrasekaran, John R. Josephson, and V. Richard Benjamins, "What Are Ontologies, and Why Do We Need Them?," *IEEE Intelligent Systems*, pp. 20-26, January/February 1999.

[7] L.O. Bonino da Silva Santos, L. Ferreira Pires, and M. van Sinderen, "Service Provisioning Support for Non-Technical Service Clients," in *Proceedings of the Seventh International Conference on Information Technology*, Las Vegas, Nevada, USA, 2010.

[8] Thomas Kühne, "Matters of (meta-) modeling," *Software and Systems Modeling*, vol. 5, no. 4, pp. 369-385, December 2006.

[9] Object Management Group, OMG: MDA Guide Version 1.0.1, 2003.

[10] Colin Atkinson and Thomas Kühne, "Model-Driven Development: A Metamodeling Foundation," *IEEE Sofware*, vol. 20, no. 5, pp. 36-41, Sept.-Oct. 2003.

[11] Object Management Group, OMG Unified Modeling Language Specification, 2003, http://doc.omg.org/formal/03-03-01.

[12] Object Management Group, Meta Object Facility (MOF) Specification, 2003, http://doc.omg.org/formal/02-04-03.

[13] Jean Bézivin, "On the unification power of models," *Software and Systems Modeling*, vol. 4, no. 2, pp. 171-188, 2005.

[14] M.P. Papzoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-Oriented Computing Research Roadmap," in *Dagstuhl Seminar Proceedings 05462*, March, 2006.

[15] Giancarlo Guizzardi, *Ontological Foundations for Structural Conceptual Models*. University of Twente, Enschede, The Netherlands, 2005, ISBN 90-75176-81-3.

[16] Giancarlo Guizzardi and Gerd Wagner, "A Unified Foundational Ontology and some Applications of it in Business Modeling," in *CAiSE Workshops (3)*, 2004, pp. 129-143.

[17] W. Degen, B. Heller, H. Herre, and B. Smith, "GOL: Towards an axiomatized upper level ontology," in *Proceedings of FOIS'01*, Ogunquit, Maine, USA, October 2001.

[18] C. Welty and N. Guarino, "Supporting ontological analysis of taxonomic relationships," *Data & Knowledge Engineering*, vol. 39, no. 1, pp. 51-74, October 2001.

[19] Arie Van Deursen and Paul Klint, "Domain-Specific Language Design Requires Feature Descriptions*," *Journal of Computing and Information Technology*, pp. 1-17, Oct. 2002.

[20] L.O. Bonino da Silva Santos, V.S. Sorathia, L. Ferreira Pires, and M.J. van Sinderen, "An Approach to Dynamic Provisioning of Social and Computational Services," in *Proceedings of the 6th IEEE Congress on Services*, Miami, Florida, USA, July 5-10, 2010, pp. 24-31.

[21] The Protégé Ontology Editor and Knowledge Acquisition System. [Online]. http://protege.stanford.edu/

[22] Holger Knublauch, Ray W. Fergerson, Natalya F. Noy, and Mark A. Musen, "The Protégé OWL Plugin: An Open Development Environment for Semantic Web Applications," in *The Semantic Web – ISWC 2004*, Hiroshima, Japan, November 7-11, 2004, pp. 229-243.

[23] N. Noy, R. Fergerson, and M. Musen, "The Knowledge model of Protégé-2000: Combining interoperability and flexibility," in *2nd International Conference on Knowledge Engineering and Knowledge Management (EKAW'2000)*, vol. 1937/2000, Juan-les-Pins, France, 2000, pp. 69-82.

[24] Eclipse - The Eclipse Foundation open source community website. [Online]. http://www.eclipse.org

[25] Richard C. Gronback, *Eclipse Modeling Project. A Domain Specific Language (DSL) Toolkit*. Boston, USA, 2009, ISBN-13: 978-0-321-53407-1.

[26] emftriple - (Meta)Models on the Web of Data. [Online]. http://code.google.com/a/eclipselabs.org/p/emftriple/

[27] Tony Buzan and Barry Buzan, *The Mind Map Book*. England: BBC Active, 2006, ISBN: 978-1-4066-1279-0.

[28] ATLAS Transformation Language. [Online]. http://www.eclipse.org/m2m/atl/

[29] Ian Horrocks et al. (2004, May) SWRL: A Semantic Web Rule Language Combining OWL and RuleML. [Online]. http://www.w3.org/Submission/SWRL/

[30] Uwe Assmann, Steffen Zschaler, and Gerd Wagner, "Ontologies, Metamodels, and the Model-Driven Paradigm," in *Ontologies for Software Engineering and Technology*.: Springer, 2006, pp. 249-273.

[31] J. Bézivin et al., "An M3-Neutral infrastructure for bridging model engineering and ontology engineering," in *Interoperability of Enterprise Software and Applications*.: Springer, 2006, pp. 159-171.