Use case modeling within object-role modeling

Ramírez Montaño, W.

Award date:
2013

Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science
Software Engineering & Technology Research Group

Use Case modeling within Object-Role Modeling

Conceptual modeling into the Object-Oriented paradigm

Waldo Ramírez Montaño

Graduation Supervisor: dr. ir. I. Barosan
Graduation Tutor: ir. F.A.I. Peeters
External Supervisor: dr. ir. R. Middelkoop

Assessment Committee:

prof. dr. ir. M.G.J. van den Brand
dr. ir. I. Barosan
ir. F.A.I. Peeters
dr. ir. R. Middelkoop
dr. E.P. de Vink

Eindhoven, November 2013

“Imposible” es una enorme palabra que gira alrededor de quienes prefieren el sólo vivir en el mundo que han recibido, en lugar de explorar la capacidad que tienen para cambiarlo. Imposible no es un hecho, es una opinión. Imposible no es una declaración, es un reto. Imposible es potencial. Imposible es temporal. Imposible es el todo o la nada... tú lo decides.

“Impossible” is an enormous word that twists around those who prefer to only live in the world that they have received, rather than to explore their capacity to change that world. Impossible is not a fact, but an opinion. Impossible is not a declaration, but a challenge. Impossible is potential. Impossible is temporal. Impossible is everything or nothing... you decide.

— 2012, Waldo Ramírez Montaño.

Abstract

The user requirements are typically the starting point in the development of a system. The elaboration of requirements involves several kinds of participants: from domain experts to stakeholders. Thus, the contents of requirements tend to be in Natural Language to enable that any participant can read and understand them. Nevertheless, the Natural Language statements may include ambiguities or lack of clearness in the specification of the expected goals and behavior of the system. Furthermore, the development cycle of the system might cause a loose (or even lost) relationship between the requirements and the resulting software components or technical documentation.

For the development of systems in the Object-Oriented paradigm, the Early Quality Assurance (EQuA) project offers a framework to motivate a formal specification of requirements with Model Driven Engineering (MDE) and Object-Role Modeling (ORM) techniques. This specification strengthens the relationship between requirements and the resulting software components because EQuA transforms the requirements in Natural Language into a Requirements Model. The main components of this model are: Actions, Rules, Relevant Facts and Quality Attributes. This model can be modified by domain experts or stakeholders with the Symbiosis tool, so that they can create the Object Model of the system under development.

The **Use Case modeling within ORM** thesis is a contribution to the EQuA project in the specification of the expected behavior of the system. After a research and analysis of Use Cases modeling and related topics, this contribution proposes a formal metamodel of Flows, which includes a Controlled Natural Language (CNL). This formalism allows the specification of normal or alternate flows in a Use Case Model. In addition, this formalism is designed to accept the definition of an external dictionary with the vocabulary to validate the CNL. The implementation of this contribution has been achieved as a prototype that increments the functionality of Symbiosis. As a result, Symbiosis can use elements of the EQuA framework to (i) prepare the dictionary and validate the vocabulary that is used in the use cases and (ii) link Actions with use cases. Another result is that the metamodel of Flows can be extended. This extension has been partially implemented and seeks to specify a lower layer of expected behavior in the system under development, that is, the normal or alternate flows of objects. Therefore, this extension can be suitable for the definition of the Interaction Model in the EQuA framework: the expected behavior of objects of the Object Model. The completion of this extension is part of the future work.

Finally, a preliminary case study is included in this report. This study is related to the ICONIX[1] software development methodology to analyse the functionality of the prototype.

[1] Lightweight Use Case Driven methodology [43] with UML scope

Preface

This thesis completes my experience as a master's student of Computer Science & Engineering in the Eindhoven University of Technology (TU/e), faculty of Mathematics and Computer Science (W&I) in the group of Software Engineering & Technology (SET). This thesis contributes in one of the sub-projects of the Early Quality Assurance in Software Production (EQuA) project funded by the Dutch Ministry of Education, Culture and Science. The contribution consists of a Controlled Natural Language (CNL) developed with Model Driven Engineering (MDE) techniques to model flows of behavior. The implementation has been achieved as a prototype that extends the Symbiosis tool of EQuA. This prototype is reviewed with a preliminary case study. Most of the work was achieved in the ISAAC Software Solutions BV. Additional activities have been done in the TU/e, the Laboratory for Quality Quality Software (LaQuSo) and the Fontys University of Applied Sciences.

In the context of this thesis, I deeply appreciate the collaboration and feedback of my graduation supervisors Ion Barosan, Frank Peeters and Ronald Middelkoop. The connection between LaQuSo and the SET would have not been possible without Ion. Frank is more than the graduation tutor as he has conducted the development of Symbiosis and his critics and guidance have been essential feedback. The concrete advises from Ronald at ISAAC helped me to avoid technical conflicts.
I also express my gratitude to Mark van den Brand to participate as member of the assessment committee. The lead of Mark in the SET group and his encouragement in Model Driven Software Engineering have been important for the improvements of this research. The revision of Erik de Vink as a formal methods specialist has provided a valuable perception of this work.

In the context that completes my experience as a master's student in Eindhoven, I profoundly appreciate the support from my family. Nothing of this would have been possible without you.
Finally, to all my peers and friends in any context, you know that I also appreciate and thank you.

Waldo Ramírez Montaño
Eindhoven, November 2013.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 EQuA project and the Symbiosis tool

The four-year project *Early Quality Assurance in software production* (EQuA) was funded by the Dutch Ministry of Education in November 2010[11]. EQuA is focused on the early error detection and correction along the software development cycle in order to achieve early high quality results. As the scope of EQuA is extensive, the structure of EQuA consists of five sub-projects with the collaboration of academic and industry partners. The parters that are directly related to this thesis are the *Fontys University of Applied Sciences* (Fontys) and the *Eindhoven University of Technology* (TU/e) in the academic sector, as well as the *ISAAC Software Solutions* (ISAAC) company in the industry sector. The sub-project that is directly linked to this thesis is the one that attends the field of **validation of requirements and models**[37] for the early detection of errors. In the rest of this report, 'EQuA' is utilized to refer to this sub-project.

EQuA includes research about the **industrial practices** that are used in the development of software. In particular, Vonken et al.[50] prepared an academic survey about the software engineering practices in The Netherlands. This survey reveals that agile methodologies do not guarantee early high quality results. Furthermore, some companies still endorse the waterfall methodology. Another result is that stakeholders are commonly and directly involved in the definition of requirements. To increase the satisfaction and quality expectations of stakeholders and software developers, they need a clear understanding of requirements. EQuA has been developing a framework to achieve it.

The EQuA framework considers the utilization of *Natural Language* (NL) to define requirements. This strategy aids the stakeholders by minimizing the learning of new technical knowledge. One challenge with NL is the elimination of semantic ambiguities with texts of free structure. For this challenge, the EQuA framework considers *Controlled Natural Language* (CNL) approaches. Another challenge is the traceability between requirements and the components of the system design. For example, the traceability between the class diagram and the functional requirements. Vonken et al. reveal that this traceability is ignored by half of the respondents of their survey. The main reasons are business or management issues, such as the available budget. As consequences of ignoring this traceability, the time and complexity increases in terms of the software implementation, the system tests or the system maintenance. The EQuA framework pursues to achieve this traceability.

The EQuA framework proposes the **Requirements Model** and the **Object Model**, as shown in Figure 1.1. These models represent the domain of reality, that is, the abstraction of requirements from the part of reality that is feasible for the system to be developed. The transformations of elements of the Requirements Model into elements of the Object Model are achieved by the EQuA framework, which uses the *Object-Oriented* (OO) paradigm with *Object-Role Modeling* (ORM)[25] and *Model Driven Design* (MDD) strategies. **Symbiosis** is the implementation of the EQuA framework with the *Model-*

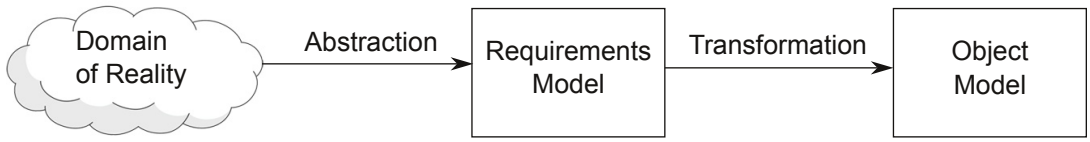
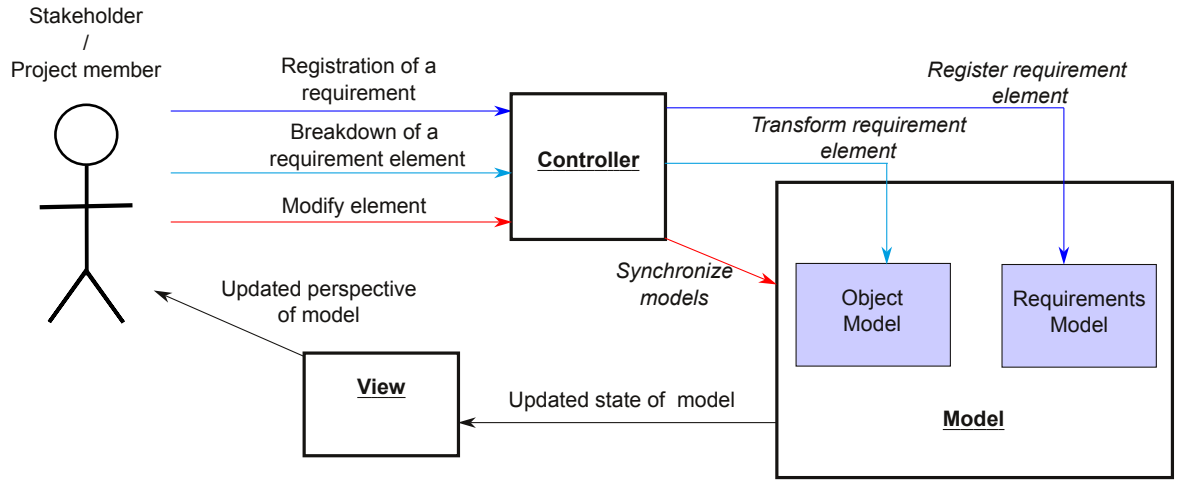


Figure 1.1: Requirements abstraction in EQuA

View-Controller (MVC) [41] design pattern as shown in Figure 1.2. Symbiosis enables the user to register the abstraction of the domain and to obtain the Object Model. The processing of user requests is done with the controller component. This component employs the EQuA framework to register, transform and synchronize elements of the Requirements and Object Models. Moreover, the controller applies updates to the view component, according to the states of the models. A general description of Symbiosis with an example project is available in Appendix A.1.



The abstraction of the domain of reality is done with the registration of requirements. The breakdown of requirements constructs the Object Model. The traceability is conducted with the synchronization of models.

Figure 1.2: MVC pattern in the Symbiosis tool

Additional information on EQuA and Symbiosis is available in [23], [42].

1.2 Motivation and problem definition

Diverse scientific and empirical studies agree with EQuA and confirm that clear abstraction and communication of requirements cause a successful design and implementation of the expected system. The achievement of this abstraction and communication are the base of the problem. The following ground work highlights concepts that are the base of the motivation. Bollen [9] argues that 'ready to use' solutions, such as *Enterprise Resource Planning* (ERP) systems, cannot guarantee a proper determination of requirements. As alternative, the explicit **semantic verification** of requirements with a conceptual modeling approach is suggested. This approach is based on the *Cognition enhanced Natural language Information Analysis Method* (CogNIAM), which is a variation of the *Object-Role Modeling* (ORM). Rosenberg et al. [43] propose an empirical Use Case Driven methodology, the *ICONIX*. This methodology confirms the importance of feedback from the stakeholders and users. Similar to EQuA, but without a formal approach, the ICONIX constructs the **static-behavior** of the domain of reality. Afterwards, the analysis of **dynamic-behavior** is encouraged with detailed discussions of Use Cases between project members and stakeholders. These behaviors are part of the motivation for a clear

BEHAVIOR	DEFINITION
Static	*Behavioral aspects of the system, which are defined by the classification of requirements in order to design the static structure of the system. Each class includes its static behavior as properties, (constrained) operations and relationships with other classes*. In the context of this thesis, the Object Model provides the *class diagram* as the design of the structure. Notice that this behavior does not contemplate time-driven behavior.
Dynamic	*Behavioral aspects of the system, which are defined by the time-driven flow of execution of the system*. In the context of this thesis, the dynamic behavior can be either *external* or *internal*. The external behavior refers to use cases as the flow of activities that the system should perform in collaboration with external users. The internal behavior refers to sequential communication between objects of the Object Model, according to the use cases. The objects use the static-behavior to accomplish the communication.

Table 1.1: Definitions of static and dynamic behaviors

communication of requirements. Their definitions are available in Table 1.1. Yang et al. [51] explore *Model Driven Architecture* (MDA) and *Unified Modeling Language* (UML) to fill the gap between the analysis of requirements and the design of solutions. These authors suggest to use **noun-verb-noun** statements in **active voice** for the specification of interaction between the user and the system, akin to ICONIX. Zikra et al. [52] propose Model Driven Development to achieve 'requirements-to-model' integration and produce reusable software components. This proposal contemplates *Natural Language Processing* (NLP) and 'guidelines' to create the models, which is analogous to Symbiosis.

The status quo of the EQuA research [37] is shown in Table 1.2. EQuA has utilized scientific and empirical knowledge to achieve a fairly mature stage of steps 1 to 6 for clear communication of requirements between project members and stakeholders. The abstraction of requirements uses a conceptual modeling similar to ORM. The formal results are the Requirements Model and the Object Model. These models derive the static-behavior of the objects that compose the system. The motivation of this thesis is how to represent the dynamic-behavior between users and the system, namely, the scenario-analysis in step 7 of the EQuA research. The dynamic-behavior can be articulated in two layers: the external-behavior and the internal-behavior, as specified in Table 1.1. The external-behavior is the scenario-analysis. The internal-behavior focuses on the interaction between objects of the Object Model. Thus, this thesis includes the analysis of internal-behavior as part of the motivation. The analysis of source-code generation is briefly inspected. Moreover, as EQuA is not only intended for academic purposes, industrial practices are considered. For the external-behavior, its modeling with use cases is part of the motivation. The problem definition is resumed in Table 1.3.

1. Construction of an Object Model from requirements.
2. Refinement of the Object Model by applying rules.
3. Visualization of the Object Model as a UML class diagram.
4. Document model elements so that all their sources are traceable.
5. Object Model transformation with Model Driven Design techniques.
6. Synchronization of model elements when a source changes.
7. **Refinement of the Object Model based on scenario-analysis.**
8. Automatic generation of test scripts based on scenarios and critical facts.
9. Generation of source code for all classes of the Object Model.

Table 1.2: Status quo of the EQuA research

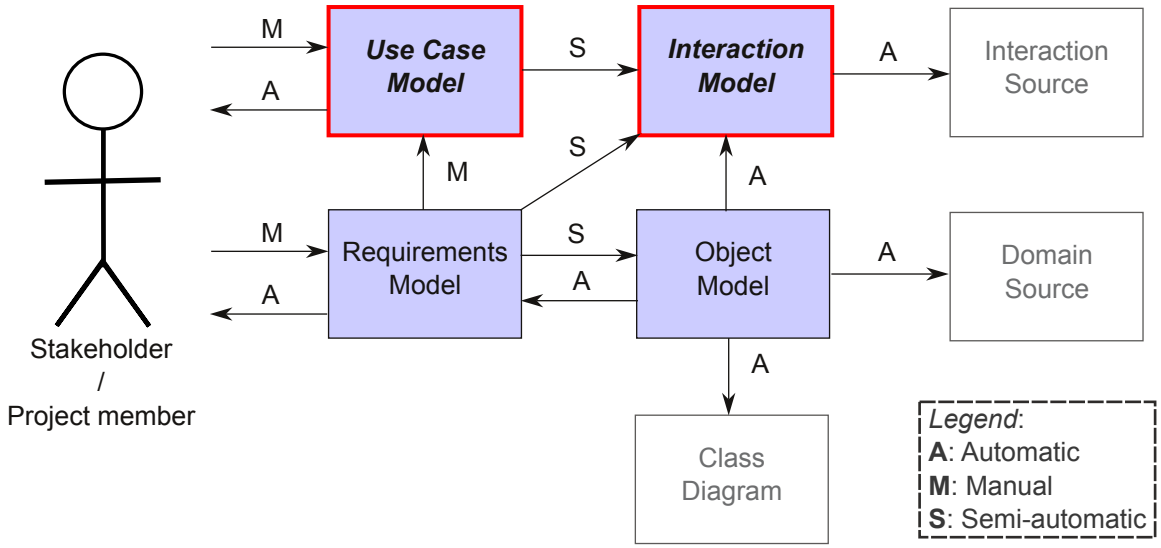
BEHAVIOR	PROBLEMS
External	- Propose a **Use Case Model** that is linked to the Requirements Model. - Validate the utilization of elements of the Object Model in the Use Case Model. - Implement a prototype for the modification and visualization of the Use Case Model. - The prototype should consider Natural Language. - Extend the Symbiosis tool with the prototype.
Internal	- Propose an **Interaction Model** that is linked to the Object Model and Use Case Model. - Propose a strategy to extend the behavior of objects. - Research the possibility to transform the Interaction Model into source code.

This thesis emphasizes the attention to the *external-behavior*, which is indicated as *scenario-analysis* in Table 1.2.

Table 1.3: Problem definition

1.3 Scope of the research

The design of the EQuA framework is depicted in Figure 1.3. The project members and stakeholders are the users of this framework. These users utilize *Natural Language* to manually register requirements in the Requirements Model. Similarly, these users should register use cases in the Use Case Model. These two models should provide automatic updates as responses, i.e., transformations or modifications to model elements. Symbiosis already comprises the Requirements Model and its updates. This thesis focuses on accomplishing the Use Case Model. Moreover, the **manual linkage** between these two models is also part of the scope.



The models highlighted in red represent the scope of this thesis. The inspections of the *Requirements Model* and *Object Model* are needed due to their linkage with the models in the scope. The *Interaction source*, *Domain source* and *Class diagram* are additional topics in the EQuA framework, which are out of scope in this thesis.

Figure 1.3: The EQuA framework

Symbiosis tool allows the users to semi-automatically create the Object Model from the Requirements Model. The traceability of requirements is managed with automatic updates in the Requirements Model due to modifications in the Object Model. Correspondingly, a semi-automatic creation of the Interaction Model from the Use Case Model should be expected. Furthermore, the modifications in the Object Model should trigger automatic updates in the Interaction Model. The scope of this thesis includes the Interaction Model.

1.4 Research questions

This thesis addresses the motivation and problem definition with two research questions. The first question encloses the main goals of this thesis as it refers to the definition of Use Case and Interaction Models according to the EQuA models. The second question is a consequence of the first question, as it refers to constraints that assume the availability of Use Case and Interaction Models. These questions are as follows:

RQ$_1$ *How can the Use Case and Interaction Models be designed based on the current Object Model?*

The answer of this question requires a broad analysis of the Object Model, Requirements Model and Symbiosis.

→ A formalization of dynamic external-behavior is expected with the Use Case Model, whereas, a formalization of dynamic internal-behavior is expected with the Interaction Model. Is it feasible to propose one model-driven formalism that is re-used in both cases?

→ The Use Case Model needs a strategy to link requirements with use cases. What cardinalities to use between use cases and the requirements in the Requirements Model?

→ In the Interaction Model, how to represent the sequential communication between objects of the Object Model?

→ How to categorize the use cases according to the Requirements Model?

→ How to utilize a Natural Language approach with the Object Model that becomes the source of vocabulary for the Use Case and Interaction Models? How to validate the syntax and semantic of the Natural Language approach?

→ How to acknowledge external sources of behavior in use cases? The use case actors should be included in the Use Case Model, but how they interact with the Object Model?

→ How could the Interaction Model foresee the operations of objects, as well as to scrutinize the possibility to add new operations?

→ How to maintain the traceability between the formalization of dynamic-behavior and the EQuA models?

→ The prototype is expected to be compatible with the Symbiosis tool.

RQ$_2$ *How can we handle the [manually added] rules?*

The Requirements Model recognizes 'rules' as a special kind of requirement. To answer this question, a deeper analysis of these rules is needed, as they serve as constraints for the dynamic-behavior.

→ A portion of rules is automatically generated by EQuA. Is it convenient to make the revision of these rules as part of the semantics validation?

→ The rest of rules are manually added by the user. What could be utilized to extend the validation of rules to include new rules?

1.5 Thesis outline

The remainder of this report is structured as follows. **Chapter 2** starts with the analysis of the Requirements and Object Models from the EQuA framework. This chapter finishes with an overview of related work. **Chapter 3** focuses on the $\mathbf{RQ}_1$ with special attention to the Use Case Model and the possibility of re-using it to propose the Interaction Model. This chapter considers suggestions of industrial practices, in particular, ICONIX. **Chapter 4** describes the formalization of the Use Case and Interaction Models with *Model Driven Engineering* (MDE) techniques via a *Domain Specific Language* (DSL) with *Controlled Natural Language* (CNL). Moreover, to answer the $\mathbf{RQ}_2$, this chapter discusses a middleware component to handle requirement rules and the linkage between the models in the EQuA framework. **Chapter 5** includes a detailed description of the implementation of Use Case and Interaction Models as a prototype. This description includes technical issues, the architecture of the prototype, as well as the validation of models with aid of the middleware component. This chapter closes with a preliminary case study to review the functionality of the prototype in Symbiosis. **Chapter 6** completes this report with conclusions and future work.

Chapter 2

Preliminaries and related work

This chapter introduces a comprehensive analysis of the foundations of this thesis. First, the reasoning on the EQuA framework models is resumed. Afterwards, the investigation of selected related work is presented.

The *User Requirements Specification* (URS) describes the **domain of reality** for the system to be developed. This specification is commonly written in *Natural Language* (NL). The NL benefits the stakeholders in the validation of requirements, although ambiguities may arise – specially in the understanding of requirements. The Requirements Model offers a structured representation of the URS to avoid its misconception. This model is the mediator between the abstraction of the domain of reality and the Object Model. Complementary, the Object Model is the mediator between the Requirements Model and the dynamic-behavior models: the Use Case and Interaction Models.

2.1 EQuA Requirements Model

The Requirements Model pursues the examination of the URS by means of *Model Driven Design* (MDD) with an *Object-Role Modeling* (ORM) perception. This model proposes four kinds of requirements, which are described in Table 2.1. Each kind is a component of the model. The *action*, *fact* and *rule* requirements conform the functional requirements. These three components, with emphasis on the *facts*, produce the **vocabulary** that should be used within the Use Case and Interaction Models. The detail of the *actions* should be specified with use cases. Thus, a **linkage** between the Requirements Model and the Use Case Model is noticed. The *rules* foreshadow constraints that create linkages between the Requirements Model and the Use Case, Interaction and Object Models. The constraints that are applied to the Object Model have a direct relation with the Interaction Model, as these constraints restrict the interaction between objects.

REQUIREMENT	DESCRIPTION
Action	General description of tasks or activities that are intended to be executed with the support of the system.
Fact	Evidence (i.e., verifiable truth) from the domain of reality. This component establishes the pillar of **conceptual modeling** in Symbiosis. The *facts* overlay the relationship between objects and underlay the *actions*.
Rule	Constraints with respect to *facts* and *actions*. Therefore, this component applies restrictions to either the static- or dynamic-behavior of the system.
Quality	Attributes stipulated by non functional requirements. These requirements have the scope of non-behavioral requirements[47], which is out of scope in this research.

Table 2.1: Components of the Requirements Model

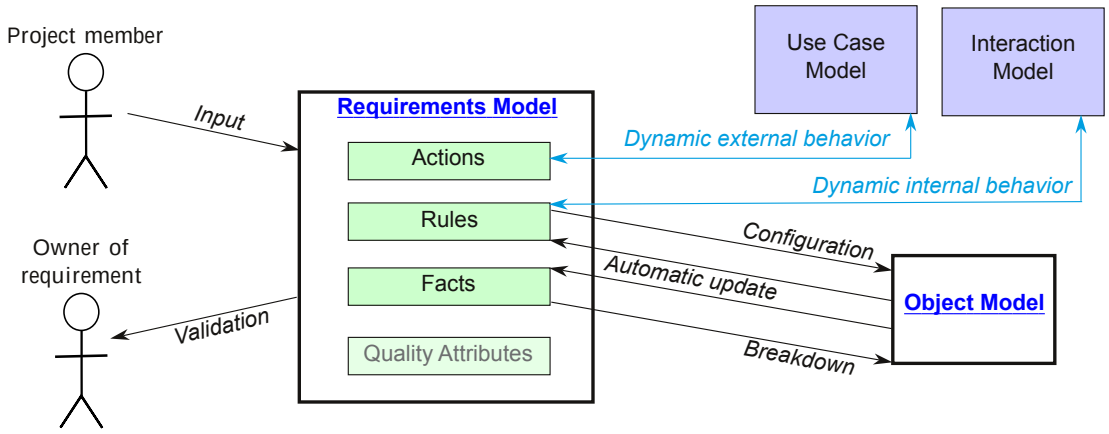


Figure 2.1: The context of the Requirements Model

Figure 2.1 depicts an overview of the input and validation of requirements. First, a *project member* or *product owner* registers a requirement in the Requirements Model. Second, a stakeholder becomes the owner of the requirement. This owner governs the life-cycle of the requirement. For instance, the approval or denial of the requirement. Next, this life-cycle is managed with Symbiosis. The *facts* receive semi-automatic breakdown transformations to populate the Object Model. Similarly, the *rules* receive semi-automatic transformations to configure the Object Model. The *facts* and *rules* are automatically updated according to modifications in the Object Model, which achieves the traceability between the Requirements and Object Models. This thesis seeks the definition of the linkages between the Requirements Model and the dynamic-behavior models, specifically, (i) between *actions* and the Use Case Model and (ii) between *rules* and the Interaction Model.

An example of requirements is available in Table 2.2. The *fact* specifies the context of the *action*. Also, the Requirements Model assumes *facts* as concrete evidence, rather than a generalization of evidence. In other words, this *fact* indicates the concrete proof of a unique book, order and price in order to aim for a clear abstraction of evidence. The context and the concrete evidence are ingredients based on the ORM[1], which are used to avoid ambiguity and to create a formal representation of requirements. The *rule* example is a result of a configuration in the Object Model. Specifically, the terms 'Book 9815 on order 27' and '23.50 euro' are recognized as formal elements, instead of simple NL. In addition, this *rule* is updated with NL for an easier understanding of stakeholders. Finally, the *quality attribute* example specifies an important security attribute, although it is out of scope.

REQUIREMENT	EXAMPLE
Action	A customer can order one or more books.
Fact	The price of Book 9815 on order 27 is 23.50 euro.
Rule	Two (or more) facts about "The price of <orderedBook:OrderedBook> is <price:Real> euro." with the same value on <orderedBook:OrderedBook> are not allowed.
Quality	Sign up by a customer is protected.

The **breakdown** of the *fact* is the source of the formal elements that are specified in the *rule*, between angle brackets. These formal elements are another representation of *facts*, but in the context of the Object Model. The visualization of this breakdown example in Symbiosis is available in Appendix A.2.

Table 2.2: Example of requirements

[1]A formal representation with ORM focuses on specifying **what** the system should do and not precisely **how** the system should do it, so that the *prediction* of the possible states of the system is feasible[17]. In Symbiosis, the main formal representation is the Object Model.

2.2 EQuA Object Model

The Object Model is the **conceptual model** of the domain of reality, because it contains type level components that play roles in this domain according to *facts* [24]. These components are created with the **breakdown** process, which is based on conceptual modeling methodologies [25]. Special attention is given in the following aspects of communication of information: registry of relevant *facts*, maintenance of *facts*, derivation of *facts* from other *facts* and retrieval of *facts* when requested [4]. These aspects are managed with the Object Model Meta-Model, which is presented in the next figure.

The breakdown of a *fact* returns its type level equivalent, namely the *fact-type*. An Object Model directly depends on *fact-types*. A *fact-type* is composed of *roles* because it represents the relationship in which *substitution-types* play a *role* in the source *fact*.

The *substitution-type* is implemented by either *object-types* or *base-types*. An *object-type* contains *roles*, but it *does not* represent a relationship between these *roles*. A *base-type* cannot contain roles because it represents a breakdown terminal. The type of object for a *base-type* is one of the following: `String`, `Boolean`, `Natural`, `Integer`, `Real` or `Character`.

The *fact-types* are also the generalization of *object-types*, hence, an *object-type* is a *fact-type*. The size of a *fact-type* (or *object-type*) is defined as its amount of *roles*.

A breakdown example is illustrated in Figure 2.3.

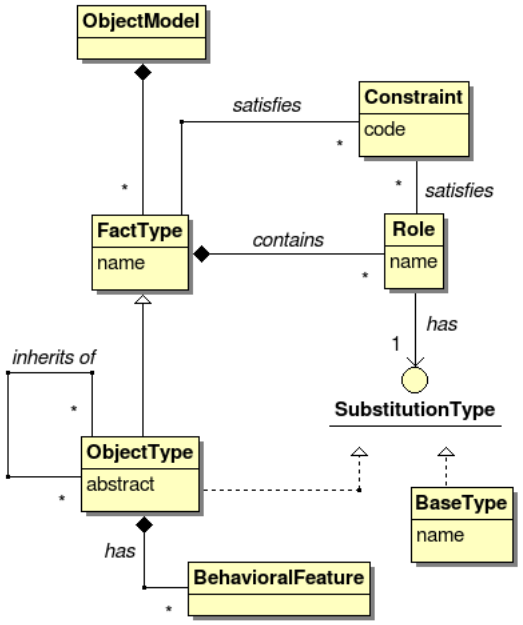


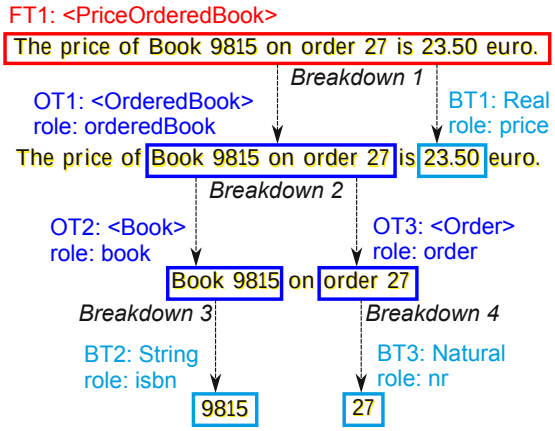
Figure 2.2: The Object Model Metamodel (simplified version)

The *constraint* and *behavioral-feature* elements are not precisely type level components of the metamodel. They represent meta-data for Object Models. The *constraints* provide configuration for the Object Model. This configuration includes the coverage of *rule* requirements. Therefore, the *constraints* should be used by the Interaction Model to restrict the dynamic-behavior between *fact-types*. A *constraint* example is depicted in Figure 2.4. The *behavioral-features* determine characteristics of the static-behavior in the Object Model, such as properties or operations of *object-types*. The *behavioral-features* should be used in the Interaction Model for the communication between *object-types*. On the other hand, the Use Case Model should utilize the type level components for the communication between users of the system and the Object Model. Therefore, the *behavioral-features* and the type level components should be part of the **vocabulary** for the Use Case and Interaction Models.

The Object Model should be 'elementary' to allow its transformation into the class diagram. The formal specifications[2] of the Object Model assert that elementary *fact-types* are the atomicity of an elementary Object Model. If a breakdown causes that the corresponding *fact-type* loses information, then this *fact-type* is elementary. So, the non-elementary *fact-types* should receive breakdown to achieve elementary *fact-types*. From the formal scope, the elementariness of a *fact-type* is reviewed with its size, its uniqueness constraints (uc_n) and the 'n-1 rule' formalism. A uc_n specifies a set of *roles* that should have unique values, i.e., no duplication of values in distinct instances of the *fact-type*. The 'n-1 rule' evaluates that the size of the smallest uc_n is at least one less than the size of the *fact-type*.

[2] Selected formalisms of the Object Model are available in Appendix B

The allocation of uc_n is semi-automatic and its implementation generates configurations such as the example in Figure 2.4. The analysis of this example with uc_n is depicted in Figure 2.5.



Legend:
FTn: *fact-type* n, OTn: *object-type* n,
BTn: *base-type* n

The *fact* that starts this example is the one from Table 2.2. The *breakdown 1* creates FT1 with two roles, one for OT1 and one for BT1. FT1 relates OT1 and BT1 with the verb 'is'. The *breakdown 2* creates two *roles* that are contained in OT1, one for OT2 and one for OT3. OT1 does not relate OT2 and OT3. No breakdown is applicable in BT1. The *breakdown 3* produces the *role* for BT2, which is contained in OT2. Similarly, the *breakdown 4* returns the *role* for BT3, which is contained in OT3. No breakdown is feasible for either BT2 or BT3.

Figure 2.3: Fact breakdown example

Rule: Two (or more) facts about "The price of <orderedBook:OrderedBook> is <price:Real> euro." with the **same value** on <orderedBook:OrderedBook> **are not allowed**.

FT1$_a$: The price of Book 9815 on order 27 is 23.50 euro.
FT1$_b$: The price of Book 9815 on order 27 is 32.00 euro.

The *rule* that initiates this example is the one from Table 2.2. This *rule* states a condition to control the instantiation of the *fact-type* described in Figure 2.3. The two example instances of FT1 have the same value on OT1. This situation violates the *rule* and would cause corruption of the Object Model, for instance, there would be conflict to indicate if either FT1$_a$ or FT1$_b$ is the valid instance of FT1.

Figure 2.4: Constraint example

PriceOrderedBook		
ROLE	TYPE	UC
orderedBook	OrderedBook	uc_{15}
price	Real	—

OrderedBook		
ROLE	TYPE	UC
book	Book	uc_3
order	Order	uc_3

Both *fact-types* are elementary. The size of PriceOrderedBook is 2 and its uniqueness constraint uc_{15} matches the 'n-1 rule' with 1 *role*. If two or more instances of this *fact-type* duplicate the value of the *role* 'orderedBook', the uc_{15} is disobeyed, such as exposed in Figure 2.4. The elementariness of OrderedBook is similar: its size is 2 and its uniqueness constraint uc_3 accomplishes the 'n-1 rule' with 2 *roles*. Hence, two or more instances of OrderedBook should avoid the same combination of values in the *roles* 'book' and 'order'. The display of uc_3 in Symbiosis is available in Appendix A.3.

Figure 2.5: Uniqueness constraint example

Additional *constraints* are important for the transformation of the Object Model into a class diagram. These are the dynamic constraints and the navigability. The dynamic constraints are the responsibilities of a *fact-type* over its *roles*. These responsibilities establish the allowed access to these *roles*. For instance, consider a *fact-type* with three responsibilities over one of its *roles* and that these responsibilities specify that this *role* is settable, retrievable and removable. In the class diagram, these

responsibilities are transformed into the setter, getter and remove operations in the *fact-type* for this *role*. The navigability specifies if a *role* is aware of its *fact-type* or not. If a *role* is aware then it is navigable. A *base-type* cannot be navigable as it is a breakdown terminal and no additional *fact-types* can be obtained from it. The navigability allows Symbiosis to detect the relevant *fact-types* that are inaccessible in the Object Model. In order to have access to these relevant *fact-types*, Symbiosis implements the factory design pattern with singleton classes in the class diagram. Each singleton class is known as a 'registry class' and corresponds to one of the relevant inaccessible *fact-types*. Appendix A.1 includes a class diagram example with dynamic constraints and registry classes.

The Object Model has been extensively researched in Symbiosis. At the moment of this writing the accessibility to the latest publication of the Object Model research is reserved to project members. Nevertheless, additional information about this model or the Requirements Model is available in the EQuA project website [11].

2.3 Related work

The Requirements and Object Models provide the core ingredients to model the dynamic-behavior in EQuA. This section resumes the survey of related work that was studied to match other studies with the core ingredients. First, this section presents the analysis of two methodologies that pursue goals that are similar to the ones of EQuA. Next, potential topics to achieve the modeling of dynamic-behavior are discussed.

The **Object-Oriented-Method** (OO-Method) [35], [36] intends to mix formal methodologies with industrial pragmatic experience in order to offer a formal basis known as OASIS, which is analogous to the Object Model. One difference between OO-Method and EQuA is the perception of behavior: for a class, the OO-Method specifies its attributes as the static components and its operations as the dynamic components. The OO-method has also considered NLP [15], [29] with special attention to syntax and semantics patterns in the Spanish language. Another difference with EQuA is that the OO-Method receives use cases scenarios as input, rather than *fact* requirements. The OO-Method formulates semantic patterns from use cases and suggests common behavior. To achieve this, an intermediate graph model is created from the use case scenarios. This graph model contains morphological, lexical and statistical information, which is then transformed into the OASIS model.

The **Object-Process Methodology** (OPM) [6] utilizes a holistic perspective with models as diagrams. These diagrams are known as *Object-Process Diagrams* (OPD) and are based on the *Directed Typed Graphs* (DTG) formalism. An OPD has the structural and behavioral aspects in itself. However, as OPD has hierarchical architecture, the allocation of 'sub-OPDs' is permitted. To obtain these sub-OPDs, the transformations known as 'inzoom' or 'outzoom' are utilized. These transformations are based on the DTG formalism. This formalism establishes constraints to filter and validate transformations. The OPM research has focused on the interoperability and interconnectivity of systems [30]. One of the results is the *Object-Process Language* (OPL), a *Controlled Natural Language* (CNL). The OPM could be considered to model use cases with OPL and interaction of *fact-types* with OPD. Nevertheless, this strategy could require a re-structure of the Object Model to adapt it as an OPD, which is out of scope.

After the analysis of EQuA, OO-Method and OPM, observations are discussed for the modeling of external- and internal-behavior in Table 2.3.

2.3.1 SBVR, Petri nets, Use Case modeling

Bollen [7] discusses the first release of *Semantics Of Business Vocabulary And Business Rules* (SBVR) as a specification that defines a structured sub-set of English vocabulary, which is based upon fact-oriented modeling (i.e., ORM or CogNIAM). Bollen proposes to provide a structure of 'verbalizable knowledge'. For instance, a noun concept in SBVR should be structured as a noun or noun phrase. This opens the

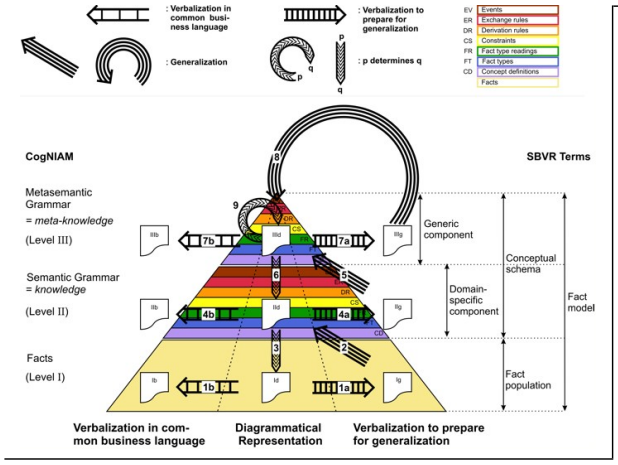
→	NLP or CNL strategies are utilized in any of the reviewed methodologies. It is definitely advantageous the review of those strategies to achieve the external-behavior. These topics are discussed in Section 3.2.
→	To predict or restrict the dynamic-behavior, the conceptual modeling (EQuA, OO-Method) or process modeling (OPM) focus on a delegate to validate constraints in the dynamic-behavior. The analysis of a standard specification to implement this delegate is convenient. CogNIAM is a variation of ORM that acknowledges the standard specification of SBVR. As EQuA is based on ORM, the analysis of CogNIAM and SBVR to control constraints is feasible.
→	The internal-behavior modelled as a flow or process in OPM invites the analysis of a standard directed graph specification, such as Petri nets.
→	EQuA and the OO-Method pursue industrial practices. The analysis of Use Case modeling alternatives with pragmatic objectives is beneficial.

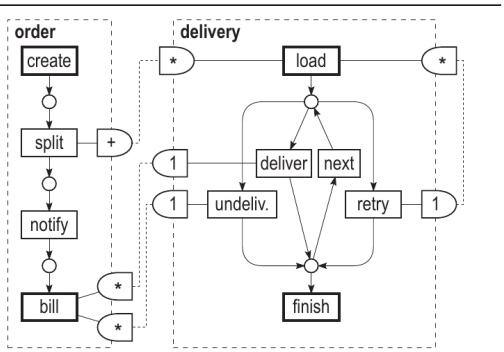
Table 2.3: Observations for the modeling of dynamic-behavior

possibility to structure SBVR noun concepts as EQuA *fact-types* or SBVR verb concepts as the relationship between *roles* in *fact-types*. Nijssen [32] suggests the utilization of CogNIAM to understand SBVR. The utilization of diagrammatic frameworks, such as the CogNIAM knowledge triange (Figure 2.6), should improve the description of core aspects of SBVR. Ross [44] considers that the essence of SBVR is the usage of 'pre-packaged semantics'. These semantics are a vocabulary that contains terms (e.g., the *fact-types* from EQuA) and their definition (e.g., the *behavioral-features* from EQuA). Bollen [8] agrees with the reasoning of Ross: "*The main building blocks for semantic in the SBVR are the following: vocabularies and terminology dictionaries, noun- and verb concepts, and definitional- and operational business rules*".
SBVR and EQuA are based on conceptual modeling, which is a clear advantage. Nevertheless, a disadvantage is that considerable time is required to analyse the complete SBVR specification and then to pursue its implementation. Furthermore, the new SBVR 1.1 specification has been recently released[3], which may require a new survey study.

The **Petri nets** could be an implementation of dynamic-behaviors as directed graphs. Hee et al. [48] present an approach to model use cases and object life-cycles as Petri nets. This approach composes Petri nets with the 'fusion' of places and transitions and decomposes Petri nets with the removal of places and (isolated) transitions. Thus, composed Petri nets represent the synchronization of use cases or object life-cycles. The transitions describe events in use cases and operations in object life-cycles. The places represent the states of the system. The goal is to compose a complex Petri net that represents the entire dynamic-behavior of the system. A similar approach is proposed by Cheung and Chow [12]. This approach composes or decomposes labelled Petri nets. The places use 'condition labels' and the transitions use 'event labels'. These labels could be utilized to represent *constraints* of the Object Model. Fahland and Woith [20] focus on behavior as adaptive processes with a set of 'scenarios', which they define as partial executions of the system. In this manner "*a scenario can be declared as possible, imperative, or forbidden*" [18], which could fit with *constraints* of the Object Model. The scenarios are represented as acyclic labelled Petri nets with the name of 'oclets'. The forbidden scenarios are predicted as 'anti-scenarios' with 'anti-oclets'. The prediction of an anti-oclet depends on the history of execution. Thus, an anti-oclet becomes detected if the history enables the state in which this anti-oclet could be executed. This strategy enables the modeling of exceptions, which could be perceived as alternate flows in use cases or object life-cycles. Another alternative is an extension of Petri nets known as 'proclets' [19]. A proclet models the life-cycle of an 'artifact' and includes 'ports' to enable interaction between other proclets; an example is displayed in Figure 2.6. The artifacts could represent *object-types* of the Object Model and ports could represent interaction between *object-types*.

[3] *Object Management Group* (OMG), September 2013, http://www.omg.org/spec/SBVR/1.1/





An artifact choreography of a delivery process where orders can be split into multiple deliveries that can be retried and that can fail.

Left: CogNIAM knowledge triange (figure taken from [32]). *Right*: Example of two proclets that model the life-cycle and interaction of the 'order' and 'delivery' artifacts (figure taken from [19]).

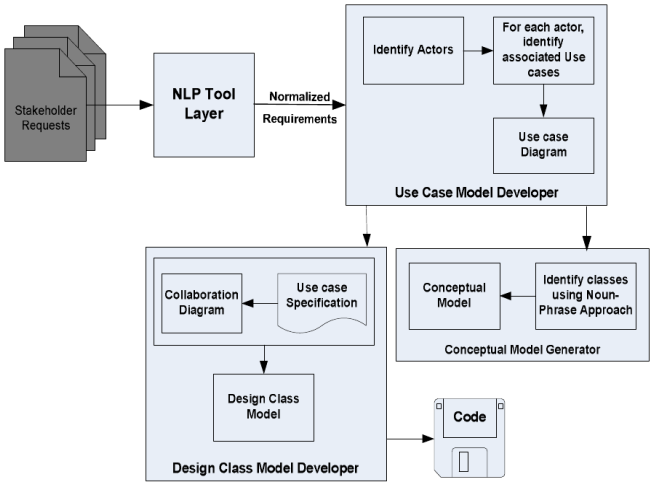
Figure 2.6: Selected excerpts of SBVR and Petri net literature

Petri nets is a formalism that could be assimilated by domain experts to model the interaction of *object-types*. Unfortunately, this formalism suffers a higher risk of acceptance from the stakeholders in the modeling of use cases.

Use Case modeling methodologies can help the project members and stakeholders in having a uniform understanding of the expected external-behavior of the system. Sinning et al. [46] discuss the misunderstandings between Software Engineering and Human Computer Interaction teams. These authors propose a 'separation of concerns' strategy to link use cases with a 'task model'. This model focuses on user-interface requirements and provides a visualization to the use case actors. Jorgensen and Bossen [26] propose 'Executable Use Cases' (EUC). An EUC consists of three tiers: (1) 'Prose tier' for an informal NL specification of use cases, (2) 'Formal tier' for an executable model to narrow the gap between informal and formal representations of use cases; model examples are UML state machines or Petri nets, and (3) 'Animation tier' for a graphic display of the formal tier in order to strengthen the communication between project members and stakeholders. Deeptimahanti et al. [14] focus on UML specifications and introduce the 'UML Model Generator from Analysis of Requirements' (UMGAR) technique. This technique uses NLP for the input requirements, which are then analysed with user feedback (i.e., semi-automatically). Moreover, UMGAR is based on 'Use-case Driven Object-Oriented Analysis and Design' (OOAD) techniques, such as ICONIX. The traceability of requirements is offered with information retrieval techniques and structured key words. In addition, a glossary is offered to facilitate the communication between project members and stakeholders. Event flows of use cases are specified in NL but with eight grammatical rules. The UMGAR architecture is depicted in Figure 2.7 and the grammatical rules are listed in Table 2.4.

The approaches of task model and EUC highlight the user interface layer, whereas EQuA and UMGAR focus in the formal specification of the dynamic-behavior. However, in contrast to EQuA, UMGAR generates the class diagram upon the use cases, that is, the static-behavior depends on the analysis of the dynamic-behavior. In EQuA, the Requirements and Object Models generate the static-behavior for then analyse part of the dynamic-behavior with use cases.

The intuition of Use-case Driven techniques and grammatical rules in UMGAR are conceived in this thesis as follows. The Use-case driven practices are analysed with the ICONIX methodology in Section 3.1. The concept of grammatical rules is substituted with the definition of a CNL in Section 3.2.2.



The process architecture of UMGAR

The *Use Case Model Developer* is the core component for the analysis of requirements. This component assumes pre-processed requirements with NLP techniques. The *Conceptual Model Generator* and *Design Class Model Developer* components are equivalent to the Object Model in EQuA (figure taken from [14]).

Figure 2.7: Selected excerpt of UMGAR literature

1. Subject (NP) in the sentence is considered as sender object.
2. Object (NP) is considered as receiver object. And Predicate (VP) can also contain noun phrase which can be treated as receiver object based on the VP structures.
3. The verb phrase between subject and object is taken as message passed between objects.
4. If sentence is having subject and predicate, without any object, then sequence stated in the use-case specification helps to identify the relation between both messages.
5. Conditional statements represent sequence of statements; and can be handled by keeping If clause at the beginning of the sentence and an end_If clause at the end of the sentence.
6. Concurrent statements show sequence of actions performed at the same time, and are handled by inserting Start_ConCurrent clause at the beginning and End_Concurrent clause at the end of concurrent statements.
7. Iterative statements are handled by inserting Start_While statement at the beginning and End_While at the end of the iterative statements.
8. Synchronization statements are handled by keeping Start_Sync word after the first sentence to show the synchronous message started and after the last sentence End_Sync word is used.

NP refers to 'Noun Phrase' and *VP* refers to 'Verb Phrase' (list taken from [14]).

Table 2.4: Grammatical rules of UMGAR

Chapter 3

Use Case Analysis and Design

This chapter presents the strategies for the analysis and design of use cases. ICONIX Use-Case driven practices fit in the analysis. These practices are contemplated in the design of the CNL, which is proposed as a DSL with MDE.

3.1 Analysis with ICONIX

Rosenberg and Stephens [43] propose ICONIX as a Use-Case driven iterative process with steps that are based on industrial empirical experience and with strengths in UML analysis and design. This process does not imply a strict project life-cycle and can be utilized in waterfall or agile methodologies. Nkandla and Dwolatzky [28] classify ICONIX as a process situated between *eXtreme Programming* (XP) and the *Rational Unified Process* (RUP). The core similarity between the EQuA framework and ICONIX is that a domain of reality is the first model that should be defined[1]. This model produces the elements of the static-behavior, which are then used in Use Case modeling. This strategy contrasts with the UMGAR process, in which the Use Case modeling is utilized to produce the elements of the static-behavior (see Figure 2.7). The ICONIX process iteration is organized in four phases, as follows,

1. REQUIREMENTS. The functional requirements are abstracted as an unambiguous domain model. Afterwards, the first-draft use cases are created and labelled as the **behavioral requirements**, which should utilize elements from the domain model.
2. ANALYSIS/PRELIMINARY DESIGN. The UML robustness diagrams of the behavioral requirements are created with **robustness analysis**. These diagrams are utilized to remark what objects of the domain model participate in use cases. As a result, updates in either first-draft use cases or the domain model are applied. This phase highlights the traceability between the domain model and use cases, as well as the proposal of operations for objects.
3. DETAILED DESIGN. The UML sequence diagrams of use cases are obtained with **sequence diagramming**. This strategy allows to assign the proposed operations to classes in the domain model. As a result, the domain model evolves into a UML class diagram.
4. IMPLEMENTATION. The **coding** of classes and their unit testing is performed. Subsequently, integration tests are prepared on the use cases to test their normal and alternate flows. Finally, the code and domain model are cleaned to prepare the next iteration.

The adoption of ICONIX in the EQuA framework is convenient as both methodologies focus on firstly defining the domain of reality to create the static-structure of the system. The identification of functional requirements in ICONIX is equivalent to the creation of the Requirements Model in EQuA. The manual modeling of functional requirements as the domain model in ICONIX is equivalent to the semi-automatic transformation of the Requirements Model into the Object Model in EQuA. Thus, the guidelines to specify behavioral requirements in ICONIX can be employed in the design of the Use Case Model in EQuA. This strategy becomes meaningful to partially answer the research question $\mathbf{RQ}_1$, that is, the formalization of the Use Case Model can consider these guidelines and the analysis

[1] This model is the *domain model* in ICONIX and the *Requirements and Object Models* in the EQuA framework.

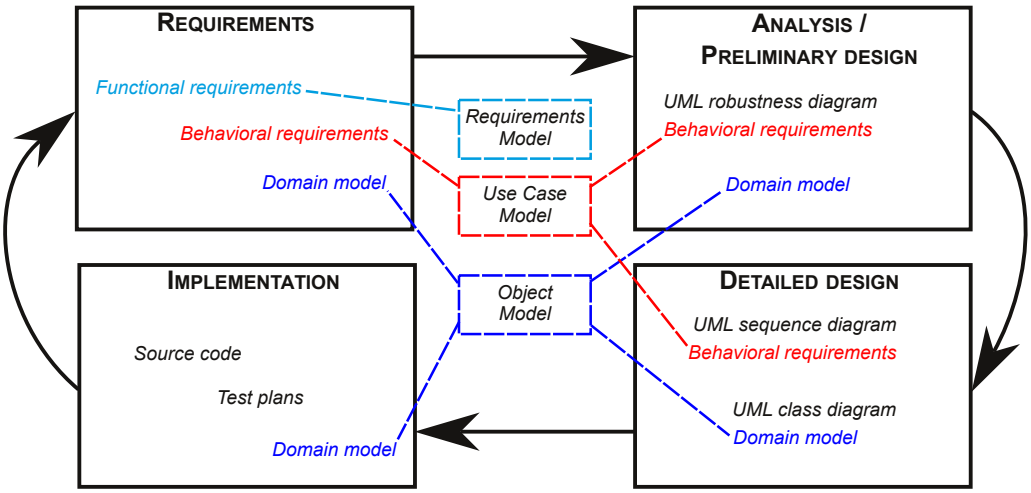


Figure 3.1: ICONIX phases related to the EQuA framework models

→	**Industrial practices** as the fundamentals with an Object-Oriented UML scope.
→	**Easy adoption**. ICONIX is a medium-sized process [28] without strict specifications for the life-cycle of a project.
→	**Domain model** before use case modeling. Equivalent to the strategy in the EQuA framework, where the domain model is the Object Model and the use case modeling is the external-behavior modeling.
→	**Traceability** of use cases with the domain model.
→	**Simple Natural Language** sentences in use cases.
→	**Precise flow structure** in use cases: event/response flows with normal and alternate flows.
→	**Easy understanding** of use cases for project members and stakeholders.

Table 3.1: Characteristics of ICONIX as benefits for the EQuA framework

of traceability of requirements is possible with the static-structure of the system. Figure 3.1 suggests the presence of the EQuA models in the ICONIX process and Table 3.1 résumés the benefits of ICONIX for the EQuA framework. The rest of this section focuses on the analysis of the ICONIX guidelines for the definition of use cases.

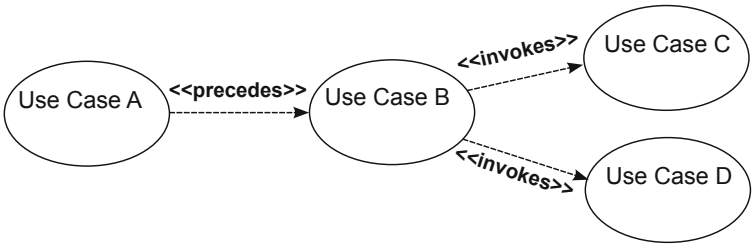
ICONIX remarks the questions *"what are the users of the system trying to do?"* and *"what is the user experience?"* for the definition of the dynamic-external-behavior. As a response, ICONIX proposes the utilization of concrete event flows in each use case according to the questions shown in Table 3.2. By acknowledging the context of the EQuA framework – specially the Object Model – and the reasoning of the $\mathbf{UCQ}_n$ questions, the guidelines are resumed in Table 3.3.

	QUESTION	REASOINING
$\mathbf{UCQ}_1$	*What happens?*	To outline the normal flow of the use case.
$\mathbf{UCQ}_2$	*Then what else happens?*	To complete the normal flow.
$\mathbf{UCQ}_3$	*What else might happen?*	To define alternate flows of the use case.

Table 3.2: ICONIX 'three magic questions' to write a use case

→	Use case should not exceed **two paragraphs**. Large use cases should be structured as a set of concrete scenarios to improve their re-usability.
→	The sentences should have **active voice** to state who or what performs the event.
→	The sentences should follow an **event/response flow** to describe the dialogue between the system and actors.
→	The sentences should have a **Noun-Verb-Noun structure** (N-V-N).
→	The sentences should include **vocabulary from the Object Model**. Some nouns as *fact-types* and their verbs as their operations. The rest of nouns as actors with their verbs as requests of persistence (i.e., CRUD — Create, Read, Update, Delete) for the system.
→	In industrial practices, the link of use cases is mostly achieved with the following use case **association stereotypes**: «invokes» and «precedes». These stereotypes replace the stereotypes «includes» and «extends» because they are sub-types of «invokes». An example of these stereotypes is available in Figure 3.2.
→	The use cases can become the base of the users guide: the **runtime behavior specification**.

Table 3.3: ICONIX Use case guidelines for the EQuA framework



The stereotype is read according to the direction of its arrow. In this example, the *Use Case A* precedes (i.e., must be completed before) *Use Case B*. On the other hand, the *Use Case B* invokes *Use Case C* and *Use Case C*.

Figure 3.2: ICONIX Use case association stereotypes

ICONIX suggests the organization of use cases per packages and actors. In the context of EQuA, this suggestion is substituted with the organization of use cases according to the **categories** of requirements defined in the Requirements Model[2]. In ICONIX, the presented guidelines correspond to the first-draft use cases, which are then formalized as UML diagrams in the forthcoming phases. Nevertheless, this thesis proposes a formal representation of use cases with NLP to define a CNL with MDE. The NL under consideration is the English language and the discussion about this proposal is initiated on the following section.

3.2 Connecting analysis and design

The phase 2 of the ICONIX process seeks to fill the gap in use cases between **what** to do (i.e., the analysis) and **how** to do it (i.e., the design) with the robustness diagram. However, Rosenberg and Stephens [43] recognize that this diagram actually depends on the NL text of use cases: *"a robustness diagram is an object picture of a use case. ... the trick is in writing your use case correctly"*. This reasoning spurs the research on NLP in accordance with the guidelines of Table 3.3.

[2]A category and use case example is available in Appendix A.4

3.2.1 Natural Language Processing (NLP)

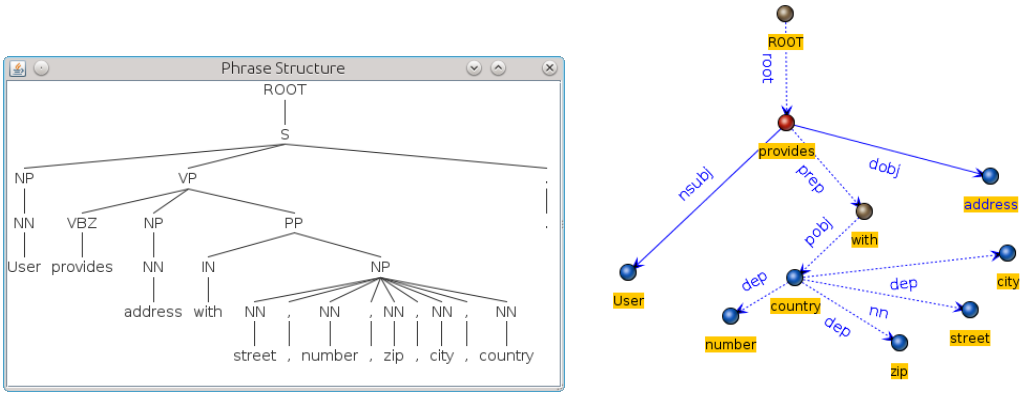
The NLP is an extensive research field with diverse outlooks. Brill [10] proposes a rule-based *Part Of Speech* (POS) tagger. This tagger is trained according to machine-learning and/or information retrieval techniques. After the learning process, an unknown word becomes tagged according to its last three letters. For instance, 'terhguous' would be tagged as an adjective because the 'ous'-ending is mostly used in adjectives. This proposal is unsuitable for the utilization of the ICONIX guidelines because the project member or stakeholder are assumed as the authors of use cases with vocabulary from the Object Model. Nevertheless, the POS tagger could be an extension of the breakdown of *fact* requirements from the Requirements Model in order to suggest *roles* and their relations.

Bajwa et al. [3] offer the Alchemy System[3] for the *Artificial Intelligence* (AI) field, including NLP knowledge extraction for OO modeling. Alchemy is related to Markov logics with declarative programming. Its NLP methodology has defined three classes according to the constraints of the NL text. These classes are: (1) *Object class* for objects or classes in a particular scenario: Main Actor Object, Co-Actor Object, Recipient Object and Thematic Object; (2) *Method class* for the function or method that is represented by the action performed in a sentence; (3) *Attribute class* to recognize adjectives in a sentence as attributes of an object. With these classes, the algorithm of Alchemy corroborates the convenience of the N-V-N structure for sentences in OO modeling. Moreover, this structure can be studied as **Noun Phrase - Verb Phrase** (NP-VP) due to the utilization of POS tagging in the algorithm. For the utilization of the ICONIX guidelines, the usage of the NP-VP structure is feasible to enrich the NL text of use cases in the EQuA framework.

The parsing of phrase structures, such as NP or VP, generate multi-word components to represent *syntax* with nested structures. The **typed dependencies** are also structures of sentences, but their parsing generates semantic trees to represent branched dependencies between words. De Marneffe et al. [13] suggest a method with two phases to extract typed dependencies. The first phase is the dependency extraction, in which any *Penn Treebank* [4] phrase structure grammar parser suffices, such as the *Stanford parser*[5] for the English language. This parser provides the phrase structure parse tree and suggest the 'head' of each component. The second phase is the dependency typing, in which the heads are used in the semantic analysis of all the words. These words are labelled (i.e., typed) with a grammatical relation. Each grammatical relation is matched with one of the patterns over the phrase structure parse tree: "*Conceptually, each pattern is matched against every tree node, and the matching pattern with the most specific grammatical relation is taken as the type of the dependency*". The graphical resulting representation of this method is a directed acyclic tree graph with one root. Figure 3.3 shows and example of phrase and typed dependency trees.

The **role posets** technique is proposed by Pérez-González and Kalita [38]. This technique consists of the utilization of the *4W language* (4WL) with partially ordered sets (posets) of roles. The 4WL is a semi-natural language with four main objectives *"... a 4W sentence tries to answer the following four questions related to a particular object: **W**hat does the object do?, **W**ho receives the action?, **W**hich other participates? and **W**hen does it happen? ..."*. The role posets is a framework based on the thematic theta roles from Chomsky, in which the role of a noun in a sentence depends on the relative position of the noun and on the semantic of the main verb in the sentence [39]. This analysis of roles is achieved with the 'roles machine', a semantic structure that groups verbs according to formal schemes. The target of role posets is to support decisions in the construction of the static-structure of the system. As a result, the *Graphic Object Oriented Analysis Laboratory* (GOOAL) [39] is an academic tool that supports sub-sets of English and Spanish languages and intends to emulate the reasoning that analysts perform in the abstraction of requirements. The role prosets framework is akin to the Object Model in EQuA. Similarly, the 4WL for role prosets is analogous to a CNL for the Object Model in EQuA.

[3] http://alchemy.cs.washington.edu/
[4] http://www.cis.upenn.edu/~treebank/
[5] http://nlp.stanford.edu/software/stanford-dependencies.shtml



Images generated with the Stanford parser and Grammarscope: `http://grammarscope.sourceforge.net/`

The sentence *'User provides address with street, number, zip, city, country'* is an example with active voice and NP-VP pattern. *Left image:* displays the phrase structure. The first NP-VP substructure yields low structural complexity in NP and high structural complexity in VP. *Right image:* displays the typed dependency structure. The root dependency is the verb and represents the communication between the noun subject (nsubj) and the direct object (dobj). Interestingly, the dependency of the preposition (prep) 'with' is with the verb, rather than with the dobj. This is due to the semantic link between verbs and 'with' in the English Language. For instance, if the prep 'in' is used instead of 'with', its dependency would be to the dobj instead than to the verb. In any case, notice that the complexity of dependencies increases in the object clause of the sentence (i.e., dobj, pobj).

Figure 3.3: Example of phrase and typed dependency structures in NL

The English language is built upon phrased structures. The specification of the CNL in this research should focus on patterns of phrase structure that match the N-V-N pattern. This is because Symbiosis users would create use cases based on the CNL instead of parsing raw NL (i.e., no NLP pre-processing is needed). The NP-VP pattern of phrase structure matches the N-V-N, as shown in Figure 3.3. Moreover, as N-V-N is simple, the typed dependencies should maintain the verb as the root dependency. Table 3.4 depicts observations about the NL, ICONIX guidelines and validation approaches for the design of the CNL ($\mathbf{RQ}_1$), which is discussed on next subsection.

→ The POS tagging strategy could be utilized to pre-process *fact* requirements for the Requirements Model and propose *fact-type* candidates.

→ The utilization of **active voice sentences** yields two structural patterns. In both patterns, the structural complexity increases in the object clause of the sentence.
(1) The NP-VP as a **pattern of phrase structure**. The VP can consist of the verb and another NP for the object clause. This pattern is feasible for the CNL for EQuA.
(2) The root-verb as a **pattern of typed dependencies structure**. The amount of branches that depend on the verb is higher in the object clause.

→ The CNL for the EQuA requires **semantic validation**. A CNL example with the support of a semantic component is the 4WL.

Table 3.4: NL observations for the design of the CNL

3.2.2 Controlled Natural Language (CNL)

Essentially, a CNL is a subset of the NL that consists of limitations of the vocabulary and grammar. The Object Model provides the vocabulary that comes from the abstraction of the Requirements Model. The limitations of grammar consist of a **formal language** to achieve syntax and semantic validations. A core definition of grammar for an English CNL tends to depend on the research context and a standard core definition is practically unavailable [33]. Therefore, the utilization of the reviewed ICONIX guidelines and NLP techniques are used in the design of the CNL to articulate flows of events. The architecture of the CNL is shown in Figure 3.4. This subsection discusses the phrase-structure grammar in detail and initiates the review of the context-free language and the CNL Dictionary.

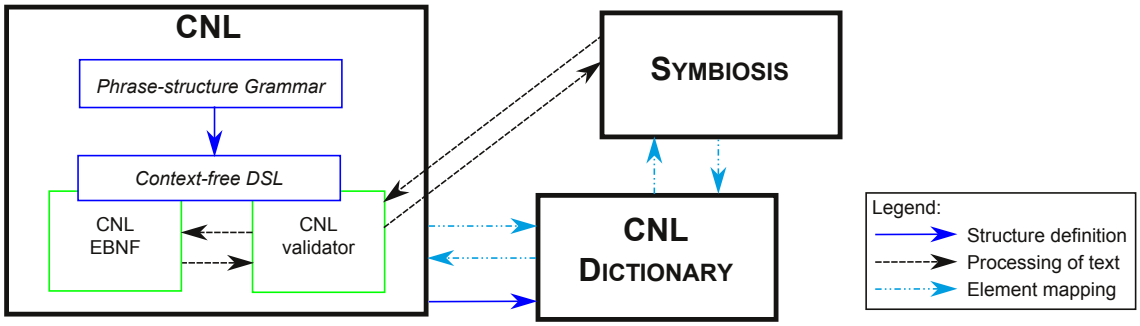


Figure 3.4: CNL architecture

The grammatical structures begin with the **atomic elements**, which constitute the lowest grammatical layer. These elements are the noun and verb, which are defined in Table 3.5. The guideline N-V-N has limitations for the Use Case or Interaction Models. For example, in the specification of properties of *fact-types*. Hence, this guideline has been adapted to the NP-VP pattern of phrase structure, as specified on Table 3.4. For this adaptation, auxiliary grammatical structures are proposed in Table 3.6. These grammatical structures aid in specifying attributes of *fact-types* and in joining grammar elements for an easier human reading. The CNL definition of NP ($\mathbb{NP}$) is also available in Table 3.6. The VP ($\mathbb{VP}$) has two structures. The first one is the callable verb phrase ($\mathbb{VP}_C$) and contains the verb ($\mathbb{V}$) and zero or more $\mathbb{NP}$. The $\mathbb{VP}_C$ is prominent in the design of sequential flows as it completes the pattern of phrase structure *NP-VP*. Nevertheless, the Interaction Model requires inner communication between flows. The interaction verb phrase ($\mathbb{VP}_I$) is the second kind of *VP* and it specifies the state of initialization or expectance of a $\mathbb{VP}_C$. The CNL definitions related to $\mathbb{VP}$ are depicted in Table 3.7.

DEFINITION 1 - NOUN ($\mathbb{N}$)	Element that represents either the semantic source of the verb (the subject) or the semantic receptor of the verb (the object). $\mathbb{N}$ can be actor ($\mathbb{N}_\alpha$) or non-actor ($\mathbb{N}_\beta$) and $\mathbb{N}_\alpha$ can be human ($\mathbb{N}_{\alpha H}$) or external-system ($\mathbb{N}_{\alpha E}$).
DEFINITION 2 - VERB ($\mathbb{V}$)	Element that represents the flow of action that is originated by a $\mathbb{N}$. $\mathbb{V}$ can be transitive ($\mathbb{V}_T$) or intransitive ($\mathbb{V}_I$) and in present tense.

Table 3.5: CNL atomic grammatical definitions

Definition 3 - Key Phrase ($\mathbb{KP}$)	Structure to specify a string that joins grammar elements or structures. $\mathbb{KP}$ is formed by syntax-free words and numbers which should be regulated with semantic validation.
Definition 4 - Base Phrase ($\mathbb{BP}$)	Structure that represents a *base-type* element that corresponds to a $\mathbb{N}$. $\mathbb{BP}$ supports the specification of a value for the *base-type*.
Definition 5 - $\mathbb{N}$ **particularization** ($part$)	Syntax-free string that specifies detail of a $\mathbb{N}$, such as a determiner (e.g., 'the', 'an') or adjectives.
Definition 6 - $\mathbb{N}$ **predicate** ($pred$)	Word that specifies the resulting CRUD state on $\mathbb{N}$.
Definition 7 - Noun Phrase ($\mathbb{NP}$)	Structure to specify a noun phrase: $\mathbb{NP} = part\ \mathbb{N}\ pred\ \mathbb{KP}\ \mathbb{BP}_1\ \mathbb{BP}_2\ \ldots\ \mathbb{BP}_i\ \ldots\ \mathbb{BP}_n$ where: $part$, $pred$, $\mathbb{KP}$ and $\mathbb{BP}_i : i > 0$ are optional.

Table 3.6: CNL auxiliary grammatical structures and NP definition

Definition 8 - Callable VP ($\mathbb{VP}_C$)	Structure to complete the pattern of phrase structure *NP-VP* by specifying either an intransitive verb phrase $\mathbb{V}_{CI}$ for unary actions without nested $\mathbb{NP}$ or a transitive verb phrase $\mathbb{V}_{CT}$ with nested $\mathbb{NP}$: $\mathbb{VP}_{CI} = \mathbb{V}_I\ \mathbb{BP}_1\ \mathbb{BP}_2\ \ldots\ \mathbb{BP}_i\ \ldots\ \mathbb{BP}_n$ where: $\mathbb{BP}_i : i > 0$ are optional and they correspond to the $\mathbb{N}$ subject. $\mathbb{VP}_{CT} = \mathbb{V}_T\ \mathbb{NP}_1\ \mathbb{NP}_2\ \ldots\ \mathbb{NP}_i\ \ldots\ \mathbb{NP}_n$ where $\mathbb{NP}_i : i > 1$ are optional.
Definition 9 - Remote Action ($\mathbb{R}$)	$\mathbb{R} = \mathbb{NP}\ \mathbb{VP}_C$ where: $\mathbb{VP}_C$ is optional.
Definition 10 - Interaction scope ($scope$)	Word that specifies if the interaction is to be called or expected.
Definition 11 - Interaction VP ($\mathbb{VP}_I$)	Structure to specify one or more remote actions: $\mathbb{VP}_I = scope_0\ \mathbb{R}_0\ \ \mathbb{KP}_1\ scope_1\ \mathbb{R}_1\ \ \mathbb{KP}_2\ scope_2\ \mathbb{R}_2\ \ldots$ $\mathbb{KP}_i\ scope_i\ \mathbb{R}_i\ \ \ldots\ \ \mathbb{KP}_n\ scope_n\ \mathbb{R}_n$ where: $\mathbb{KP}_i\ scope_i\ \mathbb{R}_i : i > 0$ are optional.
Definition 12 - Verb Phrase ($\mathbb{VP}$)	Structure to specify a verb phrase. This structure is either a $\mathbb{VP}_C$ or a $\mathbb{VP}_I$.

Table 3.7: CNL VP definition

The prior definitions are used in Table 3.8 and Table 3.9 to articulate event-types including their order of execution in the flow. In Table 3.8, the sentence ($\mathbb{AT}_S$) is the most common event-type, as it has the NP-VP pattern. The trigger event-type has two kinds. The consequence trigger ($\mathbb{AT}_{TC}$) represents pre-conditions that cause the flow. The second kind is the natural trigger ($\mathbb{AT}_{TN}$) for the typical initialization of flows. The end event-type has three kinds. The go-to event-type ($\mathbb{AT}_{EG}$) terminates

the sequential execution with a 'jump' to an event-type from the same or distinct flow. The second kind is the post event-type ($\mathbb{AT}_{EP}$), which specifies the result of a CRUD request and terminates the flow. The last kind of end event-type is the exit-use-case event-type ($\mathbb{AT}_{EE}$), which terminates the flow that executes the event and the specified flow.

Definition 13 - For-Loop ($\mathbb{L}$)	Structure to specify the initialization of a sub-flow for a particular $\mathbb{NP}$. This structure is either $\mathbb{L}_E$ or $\mathbb{L}_D$. $\mathbb{L}_E =$ 'for' 'each' $\mathbb{NP}_1$ $\mathbb{KP}$ $\mathbb{NP}_2$ $\mathbb{L}_D =$ 'for' 'every' $\mathbb{NP}$
Definition 14 - Conditional ($\mathbb{I}$)	Structure to specify a condition to initialize a sub-flow: $\mathbb{I} =$ 'if' $\mathbb{NP}$

Table 3.8: CNL loop and conditional grammatical structures

Definition 15 - Rank ($step$)	String that specifies the order of the event in the $\mathbb{FL}$.
Definition 16 - Sentence ($\mathbb{AT}_S$)	$\mathbb{AT}_S =$ $step$ $\mathbb{NP}$ $\mathbb{VP}$ '.' where: $\mathbb{VP}$ is optional.
Definition 17 - Consequence trigger ($\mathbb{AT}_{TC}$)	$\mathbb{AT}_{TC} =$ $step$ 'Cause:' $\mathbb{NP}_1\mathbb{NP}_2$ ... $\mathbb{NP}_i$ ... $\mathbb{NP}_n$ ':' where: $\mathbb{NP}_i : i > 1$ are optional.
Definition 18 - Natural trigger ($\mathbb{AT}_{TN}$)	$\mathbb{AT}_{TN} =$ $step$ $\mathbb{NP}_0$ $\mathbb{V}$ $\mathbb{KP}_1$ $\mathbb{NP}_1$ $\mathbb{KP}_2$ $\mathbb{NP}_2$ ... $\mathbb{KP}_i$ $\mathbb{NP}_i$ ... $\mathbb{KP}_n$ $\mathbb{NP}_n$ ':' where: $\mathbb{KP}_i, \mathbb{NP}_i : i > 0$ are optional.
Definition 19 - Go-to ($\mathbb{AT}_{EG}$)	$\mathbb{AT}_{EG} =$ $step$ 'Continue' FT $\mathbb{KP}$ $step$ '.' where: FT is optional.
Definition 20 - Post ($\mathbb{AT}_{EP}$)	$\mathbb{AT}_{EP} =$ $step$ 'Post' $\mathbb{NP}$ '.'
Definition 21 - Exit ($\mathbb{AT}_{EE}$)	$\mathbb{AT}_{EE} =$ $step$ 'Exit' FT '.'
Definition 22 - Sub-flow loop ($\mathbb{AT}_L$)	$\mathbb{AT}_L =$ $step$ $\mathbb{L}$ ':' λ $\mathbb{KP}$ $rank$ ';'
Definition 23 - Sub-flow conditional ($\mathbb{AT}_I$)	$\mathbb{AT}_I =$ $step$ $\mathbb{I}$ ':' λ $\mathbb{KP}$ $rank$ ';'
Definition 24 - Action-type (AT)	$AT = \{\mathbb{AT}_S, \mathbb{AT}_{TC}, \mathbb{AT}_{TN}, \mathbb{AT}_{EG}, \mathbb{AT}_{EP}, \mathbb{AT}_{EE}, \mathbb{AT}_L, \mathbb{AT}_I\}$
Definition 25 - List of action-types (λ)	$\lambda =$ Ordered list of events from the set AT.
Definition 26 - Flow-type (FT)	String that specifies the name of the flow and its type: normal or alternate.
Definition 27 - Actor list ($actors$)	List of N_α.
Definition 28 - Non-actor list ($nonActors$)	List of N_β.
Definition 29 - Flow ($\mathbb{FL}$)	$\mathbb{FL} = FT$ ':' $actors$ '.' $nonActors$ '.' λ where: $actors$ and $nonActors$ are optional.

Table 3.9: CNL Action-type and flow definitions

Moreover, the possibility to start nested flows (sub-flows) is offered with for-loop and conditional grammatical structures. These structures are shown in Table 3.8. The for-loop $\mathbb{L}_E$ focuses on a sub-flow for all the objects that correspond to a $\mathbb{NP}$ in a flow. The for-loop $\mathbb{L}_D$ focuses on a sub-flow for each $\mathbb{NP}_1$ that is contained in $\mathbb{NP}_2$. In the conditional structure $\mathbb{I}$, a $\mathbb{NP}$ is employed to establish the decision criteria to start the sub-flow. The rest of grammatical structures and definitions are available in Table 3.9. The $\mathbb{L}_E$, $\mathbb{L}_D$ and $\mathbb{I}$ are used to create two event-types, the sub-flow loop ($\mathbb{AT}_L$) and the sub-flow conditional ($\mathbb{AT}_I$). All the definitions of event-types are grouped in the set called *action-type* (AT). To complete the grammatical structures of the CNL, the representation of the flow is discussed. The flow ($\mathbb{FL}$) consists of the string *flow-type* (FT) and three lists. FT specifies the name of the use case or *fact-type* and the type of flow, that is, either normal or alternate flow. The first list specifies the set of $\mathbb{N}_\alpha$ and the second list specifies the set of $\mathbb{N}_\beta$. The last list is λ and it contains the *action-types* that represent the flow of events.

The grammatical structure definitions are the core to specify formal syntax and semantics of the CNL. These formalisms should achieve the linkage between the Use Case or Interaction Models and the Requirements or Object Models. The formal syntax does not need to define a general purpose language, as the language domain focuses on the $\mathbb{NP}$ and $\mathbb{VP}$ structures of the English language. Therefore, a Domain-Specific-Language (DSL) is suitable. In concrete, a context-free DSL is preferable due to the phrase-structure grammar of the CNL. The DSL is discussed in Section 3.3. The formal semantics accomplishes two objectives: the linkage between the aforementioned models and the semantic check of the DSL. These two objectives are designed as two components, the *CNL Validator* (CNLV) and the CNL Dictionary. The CNLV depends on the syntax validation of the context-free DSL and utilizes the Dictionary to obtain vocabulary and validate semantics. The Dictionary is an independent component that stores key strings of the language domain and exploits the advantage of Model Driven Engineering (MDE) by guarding object transformations of model components. The semantic formalisms are discussed in Sections 3.3, 3.4.

3.3 Domain Specific Language (DSL)

The expressiveness of the context-free language is limited to the phrase structures described on the previous section, yielding the convenience of a DSL rather than a general purpose language. In the computer science field, the equivalent to these structures can be an *Extended Backus-Naur Form* (EBNF) syntax notation. The proposed notation consists of production rules, terminal rules, keywords and key-symbols. The production rules are derived from the CNL grammatical definitions and structures. The terminal rules represent either *base-types*, CRUD predicates for *fact-types*, ICONIX actor guidelines, words, integer numbers or strings. The keywords are static groups of terminal symbols and the key-symbols are single terminal symbols. The keywords, key-symbols and terminal rules are utilized in the production rules. The formal definition of this EBNF as the syntax component of the CNL is presented in Table 3.10.

The CNL EBNF is utilized to achieve the **syntax validation** of the textual input. This validation applies the lexical analysis of the input to create the stream of tokens to be validated. Afterwards, the R rules validate the tokens to generate the *Abstract Syntax Tree* (AST) of the input. The strategy adopted for this syntax validation is available in the discussion of the prototype implementation, Section 5.3. As an AST example, Figure 3.5 illustrates the AST of the normal flow of use case `AddBook`. The definitions of the Subsection 3.2.2 are utilized to explain this example. The `Flow` start symbol corresponds to the grammatical structure of Def.29. The λ list (Def.25) is represented by `ActionTypeList`. Figure 3.5 provides details of the sixth `ActionType` by highlighting its derivation to a NP-VP sentence (Def.16) with `NounPhraseExp` and `VerbPhraseExp` symbols. The `VerbPhraseExp` uses Def.8 to contain an additional `NounPhraseExp` and achieve the N-V-N guideline. Moreover, the derivation of this additional `NounPhraseExp` adds information of the noun 'StockBook' with a `BasePhraseExp` (Def.4).

DEFINITION 30 - CNL EBNF Tuple $G = (V, \Sigma, R, \texttt{Flow})$, where:

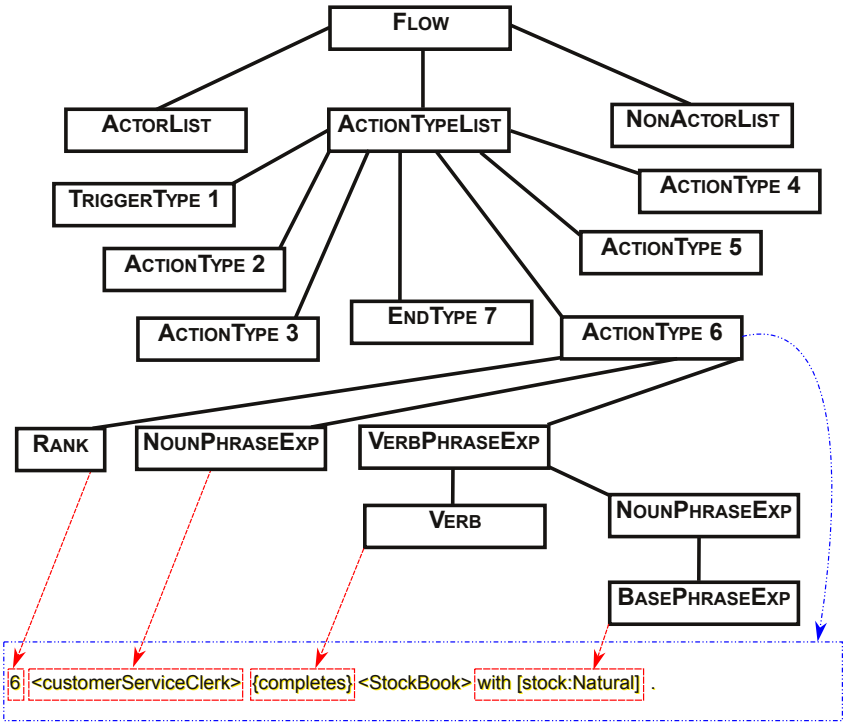
1. V is the set of non-terminal symbols that represent the grammatical structures of the CNL.
2. Σ is the set of terminal symbols that conform the alphabet. This set contains letters, numbers, key-words and key-symbols: $\Sigma = \{a,b..y,z,A,B,..,Y,Z,0,1,..,8,9\} \cup \textit{Key-words} \cup \textit{Key-symbols}$. where: *Key-words* and *Key-symbols* are specified in Appendix C.1.
3. R is the set of production and terminal rules. Each $r \in R$ has the form: $r = (v, w)$, where $v \in V$ and $w = (V \cup \Sigma)^*$. The textual mode of r is: $v ::= w$
4. `Flow` is the start symbol, $\texttt{Flow} \in V$.

The V, R and textual mode of each $r \in R$ are available in Appendix C.1.

DEFINITION 31 - CNL $CNT(G) = \left\{ w \in \Sigma^* : \texttt{Flow} \overset{*}{\Rightarrow} w \right\}$, where:

w are the strings containing only terminal symbols that can be derived from the start symbol `Flow`. The derivation ($\overset{*}{\Rightarrow}$) is the chained application of $r \in R$, that is, the repetitive application of rules to produce w from `Flow`.

Table 3.10: CNL formal definition



The `AddBook` use case is available in Appendix A, Figure A.6

Figure 3.5: Abstract Syntax Tree (AST) of `AddBook`

The AST example depicts that the design of the CNL EBNF is not strongly attached to the type level definitions of the Object Model. The only type level that is syntactically defined is the *base-type*. The rest of type level elements are abstracted as nouns by the CNL Validator (CNLV). This design issue is important in two senses related to the $\mathbf{RQ}_1$. First, the minimization of the maintainability work of the CNL EBNF with respect to the Object Model. Second, the possibility to re-use the CNL EBNF for the Interaction Model. The first reason separates syntax tasks from semantics tasks [21] and offers cohesion to the CNL EBNF. The semantics tasks are redirected to the CNLV. The second reason pursues the utilization of the CNL to represent the interaction between *fact-types*[6]. As noted in Figure 3.4, the CNLV completes the DSL and establishes the linkage between the CNL and Symbiosis. The design of the CNLV should be compatible with the Model-Driven environment in Symbiosis. Therefore, Model-Driven Engineering techniques are appropriate to complete the DSL and to propose the Use Case and Interaction Models.

3.4 Model Driven Engineering (MDE)

The CNL Validator (CNLV) accomplishes the validation of semantics with two schemes. In the first scheme, CNLV checks restrictions that do not depend on the Object Model. For example, an *action-type* with a N-V sentence (i.e., V is a unary operation) should be executed by *nonActors* and not by *actors*. In the second scheme, CNLV checks restrictions that depend on the Object Model. For instance, the base phrases ($\mathbb{BP}$) of a noun phrase ($\mathbb{NP}$) should correspond to the *base-types* of a *fact-type*. As the type elements in the Requirements and Object Models are defined with metamodels, the convenience of MDE techniques is highlighted in the formal design of the semantics validation. Details about these schemes are presented in Section 4.2. The CNLV can be perceived as a Semantic Model [21] for the DSL. In fact, the DSL of the CNL becomes a *Domain Specific Modeling Language* (DSML) [45] to facilitate the design of the Use Case and Interaction Models.

The CNLV requires vocabulary of the CNL to accomplish both phases of validation. The inclusion of vocabulary in the CNLV would augment its maintenance complexity. As an alternative, a loosely tied component is proposed, the CNL Dictionary. The metamodel of this dictionary is shown in Figure 3.6. This metamodel facilitates the classification and variation of the contents of a dictionary with respect to the CNL. The unique element that should be tightly coupled with the CNL grammar is the *Keyword* element. The *ReservedWord* and *ReservedVerb* elements are utilized to validate the CNL key phrases ($\mathbb{KP}$s). The $\mathbb{KP}$s aid in the validation of semantics with logical constraints rather than with syntax structures. In other words, this approach minimizes the definition of semantics with syntax by substituting an excess of keywords with reserved words or verbs for $\mathbb{KP}$. Furthermore, the logical constraints should allow a richer validation of $\mathbb{KP}$ instead of only syntax errors. The *RegularExpression* element permits a higher abstraction of strings to validate strings with respect to their context or type. Thus, the composite design pattern is suggested with the *ExpressionPart* abstract class to allow the manipulation of children classes to *RegularExpression*. Finally, *BehavioralFeatureMap* and *FlowMap* elements represent the mapping of elements between the CNL and the Object and Requirements Models.

The metamodel of the CNLV is also depicted in Figure 3.6. The *Restriction* element is the core of the semantics validation. The *actorKindSet* attribute configures the kinds of noun as actor, such as $\mathbb{N}_{\alpha H}$, $\mathbb{N}_{\alpha E}$ and $\mathbb{N}_{\beta}$ in the CNL grammar. The *interactionKindSet* attribute configures the kinds of interaction scope, such as 'calls:' or 'expects:' in the CNL grammar. The *Rule* element specifies the semantics constraints for the CNL. This element contributes in answering the $\mathbf{RQ}_1$ with emphasis in the formalization of the semantics validation. Moreover, this element proposes the hypothesis to answer the $\mathbf{RQ}_2$ by abstracting the *rules* of the Requirements Model. The connection between Symbiosis and the CNL is achieved by the *Validator* element. This element requests the access to a flow ($\mathbb{FL}$) in CNL and allows Symbiosis to use *Restriction* to validate the flow with vocabulary from the Object Model.

[6] Examples of `Flow` for *fact-types* are available in Appendix C.2

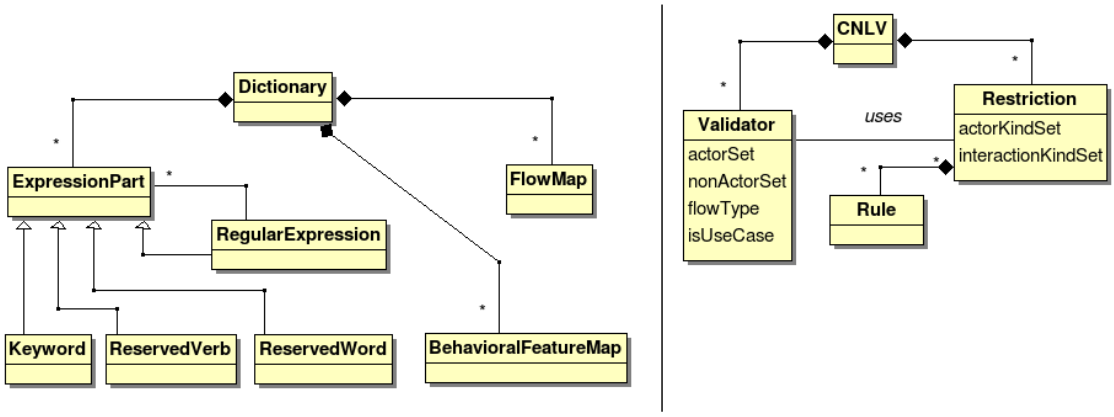


Figure 3.6: CNL Dictionary and CNLV Metamodels

The Dictionary provides an indirect support to the *Validator* and Symbiosis: The *Validator* loads the mapping of $\mathbb{FL}$s into the Dictionary for Symbiosis and Symbiosis loads the vocabulary into the Dictionary for the *Validator*. Before further detail in the semantics validation, the CNL EBNF needs to be considered in the MDE paradigm to facilitate its communication with the CNLV. This approach requires the modeling of keywords, key-symbols, production and terminal rules as part of a metamodel for the CNL EBNF. Thus, this metamodel should provide structure to the EBNF in order to enrich its textual syntax formalisms. The discussion of the design of a CNL EBNF metamodel actually completes the definition of the CNL as a DSML. This discussion motivates the review of technical specifications of Symbiosis because (i) the design of the prototype becomes deeply involved and (ii) the availability of standard frameworks that are compatible with Symbiosis could facilitate the design and implementation of the prototype. The discussion of CNL as a DSML is continued in Section 4.1 and further details of semantics validation are available in Section 4.2.

Chapter 4

Use Case and Interaction Models

This chapter discusses the CNL as a DSML, presents the Use Case and Interaction Models as an extension of Symbiosis and provides details about semantics validation. The CNL Dictionary as an external service is briefly examined.

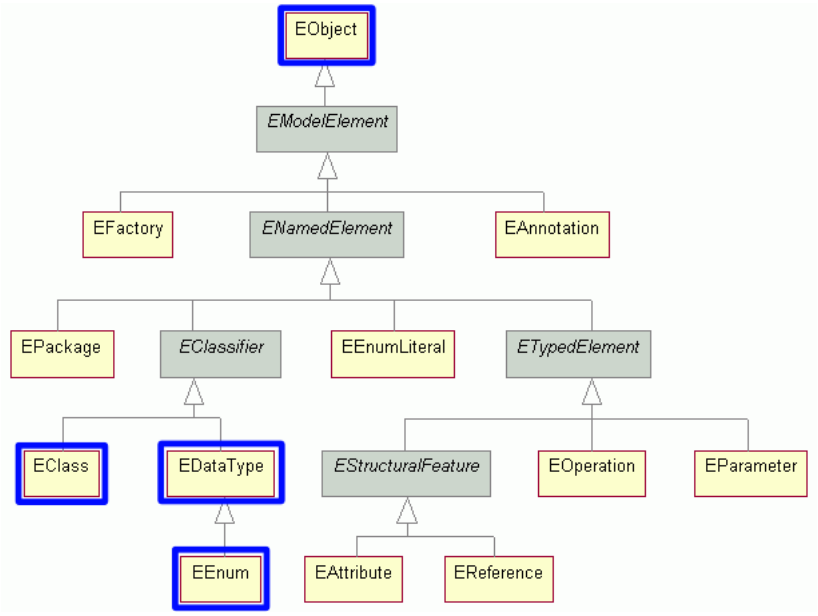
4.1 CNL with MDE: DSML

The architecture of the CNL (Figure 3.4) remarks that the formal definition of the context-free DSL is based on the phrase-structure grammar of the limited English NL. As the Symbiosis tool utilizes JRE7 as the platform for its execution, the research of frameworks related to structural MDE approaches in Java has been performed [27] [5]. A standard and mature Java framework that focuses on structured models is the *Eclipse Modeling Framework* (EMF)[1]. This framework employs the *XML Metadata Interchange* (XMI) to specify models, although their specification as Java interfaces or UML class diagrams is also supported. EMF offers runtime support and tools to transform these models into Java classes accompanied with 'adapter' classes to permit visualizing and command-based editing of models, as well as a basic model editor. The models of this framework are defined by the **Ecore metamodel**. This metamodel is defined by itself with `EObject` implementations. Figure 4.1 shows that all the classes of the Ecore metamodel implement the `EObject` interface, either directly or indirectly. Hence, the Ecore metamodel becomes the meta-metamodel of the EMF.

EMF does not need a special Java Virtual Machine (JVM), which maintains the compatibility with Symbiosis. The reworking of the CNL EBNF as an Ecore metamodel is the task to complete the design of the CNL as a DSML. The adaptation of terminal and production rules as EObject classes is a feasible alternative. For this adaptation, an extension of the EMF is available, the **Xtext language development framework**[2]. Xtext is focused on the development of DSLs and functions as a Language Workbench [21]. This is a clear advantage because Xtext includes documentation and tools to accelerate the evolution of DSLs under EMF without the need to manually model the Ecore metamodel. Furthermore, the Xtext documentation [16] assures that the resulting DSL can be utilized without the necessity of the Eclipse IDE, which suits with the needs of Symbiosis: *"The compiler components of your language are independent of Eclipse or OSGi and can be used in any Java environment. They include such things as the parser, the type-safe abstract syntax tree (AST) ... and static analysis aka validation and last but not least a code generator or interpreter. These runtime components integrate with and are based on the Eclipse Modeling Framework (EMF)"*.

Xtext offers the Grammar Language[3] as the core textual definition for new DSLs. This language opens the possibility of grammar re-usage, which Xtext refers to as 'grammar mixin'. The Grammar

[1] `http://www.eclipse.org/modeling/emf/`
[2] `http://www.eclipse.org/Xtext`
[3] `http://git.eclipse.org/c/tmf/org.eclipse.xtext.git/tree/plugins/org.eclipse.xtext/src/org/eclipse/xtext/Xtext.xtext`



The shadowed elements are abstract classes. The elements highlighted in blue are mentioned in this section (image based on http://help.eclipse.org/indigo/index.jsp?nav=/22)

Figure 4.1: Ecore metamodel class hierarchy diagram

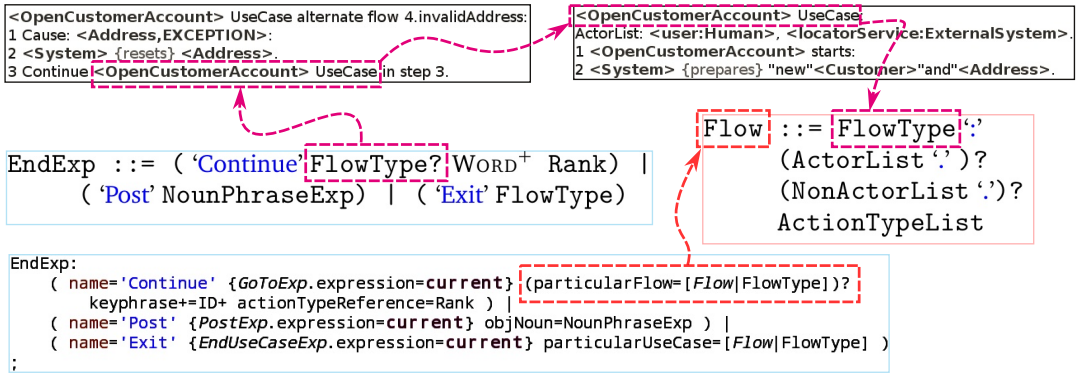
Language expects textual definitions that are based on an EBNF with an OO scope. Apart from describing the concrete syntax, the Grammar Language allows the mapping of the EBNF syntax to EObject classes. Hence, the refactoring of the CNL EBNF Tuple $G = (V, \Sigma, R, \texttt{Flow})$ produces the **Xtext CNL EBNF**, which should derive the Ecore metamodel. This refactoring is simplified with (i) the utilization of the Xtext alphabet as Σ and (ii) adjustments according to the Grammar Language $\forall r \in R$ to create the refactored rules R'. Xtext offers a common representation of terminal symbols with the grammar of Common Terminals[4]. The mixin of the grammar of Common Terminals in Xtext CNL EBNF allows to assure that $\forall r' \in R'$, r' represents a class that implements the `EObject` interface. The rules R' finish being of four types, as described in Table 4.1. The structure of the Ecore metamodel is automatically defined by Xtext according to the types of classes returned by the rules R'. The Xtext CNL EBNF and the Ecore metamodel complete the **Xtext CNL**, a Domain Specific Modeling Language (DSML).

TYPE OF RULE	DESCRIPTION
Terminal	Transforms textual input into a single terminal symbol and returns an `EDataType` (`EString` by default). A terminal rule can be configured to return other type of `EDataType`, such as `EInt`.
Data-type	Similar to a terminal rule. The main difference is that a data-type rule uses terminal symbols instead of textual input and returns an structured `EDataType`.
Enum	Enumeration of strings. This rule can be seen as a subset of data-type rules that return `EEnum`. An `EEnum` class is formed by an enumerator of strings.
Parser	These rules are employed by the parser to transform the terminals into the AST (syntax validation) and to create the Ecore metamodel with `EClass` classes.

Table 4.1: Types of grammar rules in Xtext CNL

[4] http://help.eclipse.org/helios/index.jsp?topic=/org.eclipse.xtext.doc/help/CommonTerminals.html

Xtext utilizes the terminal rules in the lexer phase to obtain a raw object representation of the textual input. The `EDataType` and `EEnum` classes are used in the parsing phase to represent data-structures in the Ecore metamodel. The `EClass` classes are utilized to complete the parsing phase and represent the logic structure of the AST, the Ecore metamodel. This metamodel is achieved with metadata in the parser rules. Xtext refers to this metadata as 'features' and 'actions'. The features are the assignment of attributes or relationships to an `EClass` by means of an `EObject` or a cross-reference. An `EObject` should be assigned as either an attribute of the `EClass` or a composition relationship with another `EObject`, whereas a cross-reference should be assigned as an association relationship with another `EObject`. On the other hand, the 'actions' specify which type of `EObject` should be explicitly returned by the parser rule. Therefore, an action can specify a generalization relationship between `EObjects`. An exemplification of features and actions is provided in Figure 4.2.



The first row of images presents two fragments of use case flows. The left-side flow uses the `EndExp` rule on the third step, which makes cross-reference with the `Flow` rule on the right-side flow. The second row of images displays the CNL EBNF of the `EndExp` and `Flow` rules. These rules clarify that the cross-reference is represented by a third rule, namely, the `FlowType`. The bottom row illustrates the refactored `EndExp` rule, which returns an `EndExp` class. This class receives the feature called *name* as an EString attribute. In addition, this class employs actions to specify that the feature called *expression* is used to generalize three classes: `GoToExp`, `PostExp` and `EndUseCaseExp`. The `GoToExp` class obtains the following features: *particularFlow* as an association relationship with the `Flow` class (by means of its `FlowType` feature), *keyphrase* as an attribute with a list of `IDs` and *actionTypeReference* as a `Rank`. The types of `ID` and `Rank` depend of the type of their rules. Similarly, the assignment of features is done for `PostExp` and `EndUseCaseExp`.

Figure 4.2: Features and Actions in Xtext CNL

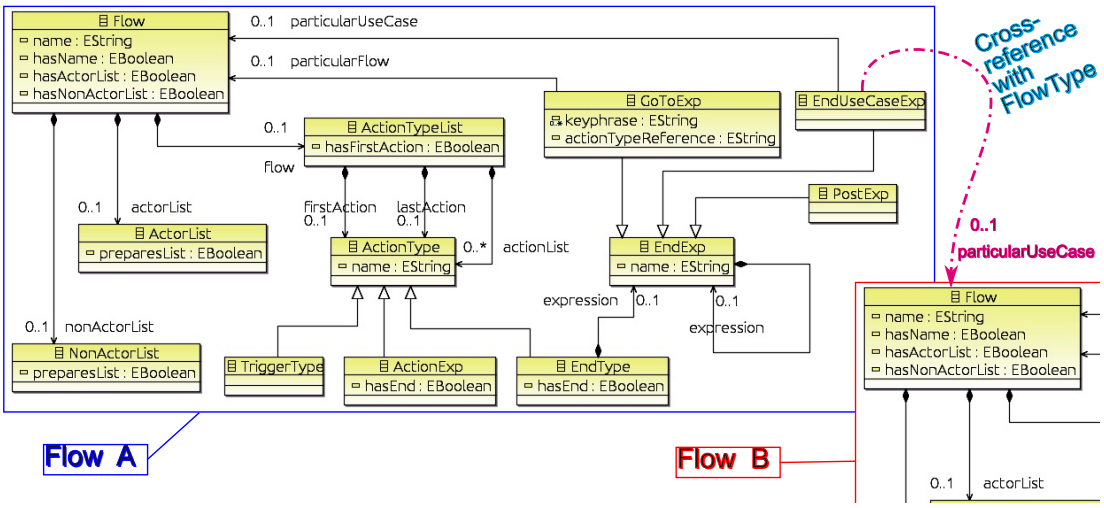
The MDE strategies evoke the following modularity in the Xtext CNL:

→ The **Xtext CNL EBNF** is the syntax metamodel that is defined by the CNL EBNF grammar refactored with the Grammar Language. This metamodel returns the structural inference of the textual input. This inference is the AST model, which Xtext labels as the 'Node Model'.

→ The **Ecore metamodel** is the semantic metamodel that is defined by the OO metadata specified in the Xtext CNL EBNF. This metadata produces the `EObject` classes that are modeled as the logic inference of the AST. This inference is the Ecore model of the CNL EBNF grammar.

→ The **CNLV metamodel** is the semantics validation metamodel that has been presented in Section 3.4. The definition of this metamodel depends on the EMF. The resulting models should perform semantics validation of Ecore models.

→ The **CNL Dictionary metamodel** is the vocabulary metamodel that has been presented in Section 3.4. This metamodel is loosely coupled with the CNL but enriches its semantics validation with vocabulary sources, such as Symbiosis.



Flow A: fragment of the transformation of the Ecore metamodel with the classes that are closely related to the `Flow` root class. The `EndExp` and `GoToExp` classes show the relationships that were cited in Figure 4.2. *Flow B* simulates to be another instance of the same fragment. The magenta arrow speculates a logical association relationship between classes of distinct instances. This speculation has been confirmed in Figure 4.2.

Figure 4.3: Semantic metamodel as a class diagram

The EMF offers alternate MDE functionalities to improve the CNL and to propose strategies of communication between the Xtext CNL metamodels and Symbiosis, specifically the Use Case and Interaction Models. Figure 4.3 presents one fragment of the Ecore metamodel as a UML class diagram that has been used to analyse the communication between the Ecore and CNLV metamodels[5]. The `EBoolean` attributes facilitate the review of contents in the EObject classes. The composition relationships improves the design of the sequence of validation. The generalization relationships are useful to specify class castings within the validation. The association relationships depict the cross-references that are helpful in the object-linking validation, specially in situations between distinct model instances, as shown in Figures 4.2, 4.3. The design of semantics validation for the Use Case and Interaction Models is discussed on next section.

4.2 Semantics Validation

As mentioned in Section 3.4, the CNLV suggests two schemes to apply validation with 'semantic restrictions'[6]. These schemes are **validation with isolated vocabulary** and **validation with vocabulary interoperability**. The vocabulary is provided by the CNL Dictionary in both schemes. In the first scheme, the vocabulary is available in the *Keyword*, *ReservedWord*, *ReservedVerbs* and *RegularExpression* components of the Dictionary. The nouns, verbs and *base-types* that are introduced in flow models are unknown and assumed as valid for the context of use cases or object life-cycles. In the second scheme, the vocabulary is extended with the *BehavioralFeatureMap* and *FlowMap* components of the Dictionary. *BehavioralFeatureMap* should be utilized by Symbiosis to include the nouns, verbs and *base-types* to employ in the Xtext CNL, whereas *FlowMap* should be used by the Xtext CNL to include the flow models that should be acknowledged by Symbiosis. The architecture of the Use Case and Interaction Models is presented in the next subsection, followed by subsections with the discussion of semantic restrictions and the possibility of utilizing the CNL Dictionary as an external service.

[5]The Appendix C.3 contains another fragment of this transformation, as well as a fragment of the transformation of the Xtext CNL EBNF into a syntax graph.

[6]These restrictions correspond to the Restriction component of the CNLV metamodel.

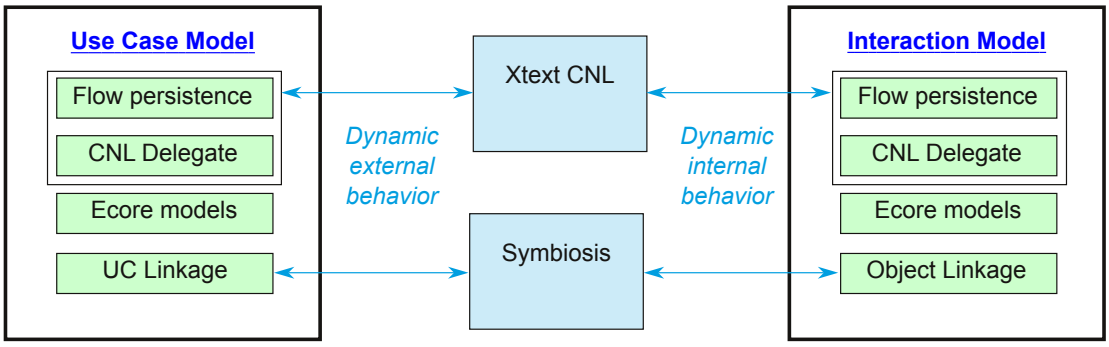


Figure 4.4: Architecture of Use Case and Interaction Models

4.2.1 Design of UseCase and Interaction Models

The Use Case and Interaction Models are proposed to extend the Symbiosis tool. Most of the research has been focused on the Use Case Model and the Interaction Model is briefly discussed. Nevertheless, these models have strong similarities in their architectures and the Xtext CNL has been designed to suit both models. This study is suggested as ground work for further research in the Interaction Model. The architecture of both models is depicted in Figure 4.4. The *Flow persistence* and *CNL Delegate*[7] components represent the interoperability between Xtext CNL and Symbiosis. Xtext CNT stores the flows according to the URI specifications of the EMF and these specifications are managed by *Flow persistence*. This persistence component is proposed because the serialization of projects in Symbiosis uses a distinct strategy. The functionalities of *Flow persistence* could be assigned to *CNL Delegate*, depending on the implementation of the Use Case and Interaction Models. *CNL Delegate* prepares and sends information to the Xtext CNL. This information includes the type and contents of flows (i.e., use case or object flows) as well as vocabulary. As a response, *CNL Delegate* receives the *Ecore models* and the results of their validation from Xtext CNL. These *Ecore models* are instances of the the Ecore metamodel and this metamodel is the formalization of the dynamic-behavior that is re-used for the Use Case and Interaction Models ($\mathbf{RQ}_1$). The *UC Linkage* and *Object Linkage* components maintain the interdependence between Ecore models and the Requirements Model. *UC Linkage* should specify the cardinality between use cases and *action* requirements, as well as the categorization of use cases ($\mathbf{RQ}_1$). *Object Linkage* should specify the correspondence between *fact-types* and *rule* requirements ($\mathbf{RQ}_2$).

Additional discussion is provided for the validation with vocabulary interoperability. The CNL Delegate should utilize the Object Model to obtain the list of *fact-types* and remove the *fact-types* that are systematically generated. In other words, the *fact-types* to consider are the ones that have been prepared by the users of Symbiosis. Afterwards, for each one of these *fact-types*, the *base-types* are collected. If a *fact-type* is an *object-type*, its methods are also collected. The CNL Delegate prepares all the collected information for the BehavioralFeatureMap component of the Dictionary.

Currently, the *rule* requirements would not be added to the Dictionary because the definition of rules in *CNL Delegate* is distinct to the definition or rules in Symbiosis. Further research is required to specify a transformation between these definitions of rules.

[7] This component is a delegate of the *Validator* component of the CNLV metamodel.

4.2.2 Semantic restrictions

The CNL use cases or object flows are analyzed as CNL Ecore models. Thus, their contents are manipulated as `EObjects` by the *Restriction* component of the CNLV metamodel (Figure 3.6). This component represents the semantic restrictions and is configured with (i) the kind of actors and interactions and (ii) *Rule* components. The first configuration should match with the EBNF rules ACTORKIND (Table C.2) and `InteractionScope` (Table C.3). The second configuration consists of semantic rules that validate `EObjects`. This subsection presents the semantic rules for the schemes introduced in Section 4.2. In the scheme of validation with isolated vocabulary, the semantic rules are referred to as **isolated rules**. These rules are the base of the semantic validation, as they use the vocabulary that is known by default. In the scheme of validation with vocabulary interoperability, the semantic rules are referred to as **interoperability rules** and they extend the isolated rules. In both schemes, the rules check the states of the `EObjects`. The valid states depend on the CNL Dictionary and on the structure of the Ecore (i.e., semantic) model.

The root of the semantics validation is formed by four isolated rules that are presented in Table 4.2. These four rules correspond to the root `EObject` (i.e., `Flow`) and the `EObjects` that compose it: `FlowType`, `ActorList`, `NonActorList` and `ActionTypeList` (Figure C.5). The exception of having a composition relationship is `FlowType`, which is designed as an `EDataType` that is manipulated as a String[8]. Hence, `FlowType` becomes the attribute 'name' of `Flow` (Figure 4.3). As design issue, the rules I_R are directly executed by the *Validator* component of the CNLV metamodel, whereas the rest of rules (i.e., rule delegates, Table 4.2) are directly executed by I_R. This issue is feasible due to the OO structure of the Ecore models. The attributes and relationships of `EObjects` provide to I_R the possibility of 'navigating' the Ecore model. Accordingly, each I_R can utilize other `EObjects` for their validations as well as call one or more χ_o rule delegates.

DEFINITION 32 - ROOT RULES (I_R)	Set of isolated rules that are the base of semantics validation, $I_R = \{\xi_{Flow}, \xi_{ActorList}, \xi_{NonActorList}, \xi_{ActionTypeList}\}$, where: ξ_{Flow} validates with `Flow` and `FlowType` *EObjects*, $\xi_{ActorList}$ validates with `ActorList` *EObject*, $\xi_{NonActorList}$ validates with `NonActorList` *EObject*, $\xi_{ActionTypeList}$ validates with `ActionTypeList` *EObject*.
DEFINITION 33 - RULE DELEGATE (χ_o)	Isolated or interoperability rule that is utilized by I_R, where o is the *EObject* to validate. The χ_o rule may use another rule χ'_o to complete the validation. Nevertheless, the χ_o rules should not utilize I_R.

Table 4.2: Semantic rules general definition

The ξ_{Flow} validates the states of `Flow` and `FlowType`. The states of `Flow` depend on its attributes and relationships. The states of `FlowType` depend on reserved words from the *ReservedWords* component (Figure 3.6). These words are used to create **key phrases**, which are lists of Strings with variable values. In this context, the reserved words are referred to as the set of words RW. The `FlowType` has four valid syntax representations according to its production rule. Consider this production rule to enumerate its WORD non-terminal symbols in order to create the data structure

[8]The reason for using `EDataType` instead of `EClass` has been to enable the utilization of `FlowType` as a String for cross-references in Xtext. In this manner, the traceability of alternate flows from its normal flow is accomplished.

$S_{FlowType}$, which represents the state candidates of `FlowType`,

`FlowType` ::= '<' WORD_a '>' ((WORD_b WORD_c) | (WORD_d WORD_e `Rank` '.' WORD_f) | ('UseCase') | ('UseCase' WORD_g WORD_h `Rank` '.' WORD_i))

then, $S_{FlowType} = \{ s1_{FlowType}, s2_{FlowType}, s3_{FlowType}, s4_{FlowType} \}$

where: $s1_{FlowType} = \{\text{WORD}_a, \text{WORD}_b, \text{WORD}_c\}$, $s2_{FlowType} = \{\text{WORD}_a, \text{WORD}_d, \text{WORD}_e, \text{WORD}_f\}$,
$s3_{FlowType} = \{\text{WORD}_a\}$, $s4_{FlowType} = \{\text{WORD}_a, \text{WORD}_g, \text{WORD}_h, \text{WORD}_i\}$

ξ_{Flow} considers {'normal','alternate','flow'} $\subset RW$, thus, `FlowType` is valid if:

for $s1_{FlowType} \rightarrow \text{WORD}_b$='normal' $\wedge$ WORD_c='flow',
for $s2_{FlowType} \rightarrow \text{WORD}_d$='alternate' $\wedge$ WORD_e='flow',
for $s4_{FlowType} \rightarrow \text{WORD}_g$='alternate' $\wedge$ WORD_h='flow'

This validation of states of `FlowType` with reserved words replaces the overuse of keywords in the static structure definition of `FlowType`. An important benefit is that the reserved words could be modified changing the structure of the syntax and semantic models. This simplifies their maintainability.

ξ_{Flow} also obtains metadata for other semantic rules. To represent the classification of `Flow`, the boolean `isUseCaseFlow` of *Validator* is used. Its value is `false` in the case of $s1_{FlowType}$ and $s2_{FlowType}$, its value is `true` in the case of $s3_{FlowType}$ and $s4_{FlowType}$. Moreover, data structure of `FlowType` is stored in *Validator*. In this manner, the attribute 'name' for the `Noun` (Figure C.4) that should represent to the `Flow`, is acknowledged with WORD_a. In the case of $s2_{FlowType}$ (or $s4_{FlowType}$), `Rank` specifies the step in the normal flow that should be the source of the use case (or object) alternate flow. To complete the validation of `Flow`, ξ_{Flow} validates the state of `Flow` with its `EBoolean` attributes 'hasName', 'hasActorList', 'hasNonActorList' and with its composition relationship with `ActionTypeList`. This composition has the name 'flow' in the Ecore model. Hence, `Flow` is valid if:

$$\text{hasName} \wedge \text{flow.size} = 1$$

In addition, ξ_{Flow} may provide warning feedback to the user:

if !hasActorList $\wedge$ isUseCaseFlow, notify the absence of actor nouns.
if !hasNonActorList, notify the absence of non actor nouns.

This validation result of ξ_{Flow} does not employ interoperability with Symbiosis. In the contrary case, *Validation* should prepare the *FlowMap* component of the Dictionary. Therefore, ξ_{Flow} may utilize the rule delegate χ_{Flow}. This delegate uses the metadata generated by ξ_{Flow} and validates that the `Flow` is registered in the *FlowMap*. In addition, if `Flow` is an alternate flow, the cross-reference to the normal flow should exist via the `Rank` of `FactType`. In other words, χ_{Flow} confirms that `Flow` is valid if,

`Flow` $\in$ *FlowMap* $\wedge$ `FlowType` has state $s1_{FlowType}$ or $s3_{FlowType}$
(i.e., the flow is known in Symbiosis as a normal flow)

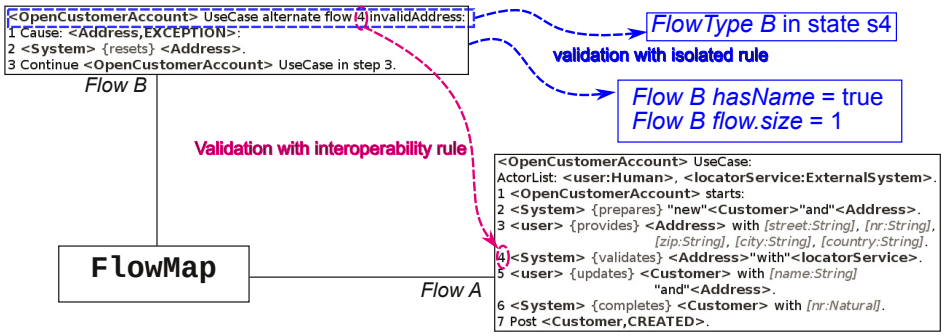
or if `Flow` $\in$ *FlowMap* $\wedge$ `FlowType` has state $s2_{FlowType}$ or $s4_{FlowType}$
(i.e., the flow is known in Symbiosis as an alternate flow)

then, $\texttt{Flow}_n \in$ *FlowMap* $\wedge$ $\texttt{FlowType}_n$ has state $s1_{FlowType}$ or $s3_{FlowType}$ $\wedge$
$\texttt{ActionTypeList}_n$ of $\texttt{Flow}_n$ has the `Rank`.
(i.e., the normal flow is known in Symbiosis and has a step identified by the `Rank`)

where: $\texttt{Flow}_n$ is the normal flow of `Flow`, $\texttt{ActionTypeList}_n$ composes $\texttt{Flow}_n$

An exemplification of one `Flow` instance in a valid state is illustrated in Figure 4.5.

The definitions of ξ_{Flow} and χ_{Flow} in the prior paragraphs use unconventional formal specification. Part of the future work is to propose their standard formal specification. Due to the strategy of validating with states of `EObjects` and the structure of the semantic metamodel, the specification of semantic rules with *Structural Operational Semantics* (SOS) [1] [31] is promising.



ξ_{Flow} validates that *Flow B* is an alternate use case. The reserved words 'alternate' and 'flow' pursue a valid *FlowType B*; *hasName* is true due to the double colons after *FlowType B*; *flow.size*=1 because *Flow B* has only one list of *action-types* (i.e., the steps 1, 2 and 3 represent the only `ActionTypeList` instance that composes *Flow B*). As the FlowMap is available, ξ_{Flow} also employs the delegate rule χ_{Flow} to extend the validation of *Flow B*. χ_{Flow} confirms the existence of *Flow B* and its normal flow, *Flow A*. The navigation in the steps of *Flow A* validates the existence of the `Rank` specified in *FlowType B*.

Figure 4.5: Validation with isolated and interoperability rules

A general form of either isolated or interoperability rules could be based on the definition of deduction rules in the SOS:

$$\frac{\langle Premises \rangle}{S \xrightarrow{t} S'}$$

where: $\langle Premises \rangle$ are the statements or configurations that anticipate a conclusion,
$S \xrightarrow{t} S'$ is the conclusion that declares a predicate from state S to state S',
$\xrightarrow{t}$ symbolizes the transition of states labelled as t.

The deduction of the transition t holds if the configurations of the premises are valid. Recalling the validation of `FlowType` in rule ξ_{Flow}, consider the deduction rule:

$$\frac{\left\langle \texttt{FlowType}, s1_{FlowType}, \textsc{Word}_b = \text{'normal'}, \textsc{Word}_c = \text{'flow'}, \{\text{'normal','flow'}\} \in RW \right\rangle}{\texttt{FlowType} \xrightarrow{UCNF} \texttt{FlowType}_{UCNF}}$$

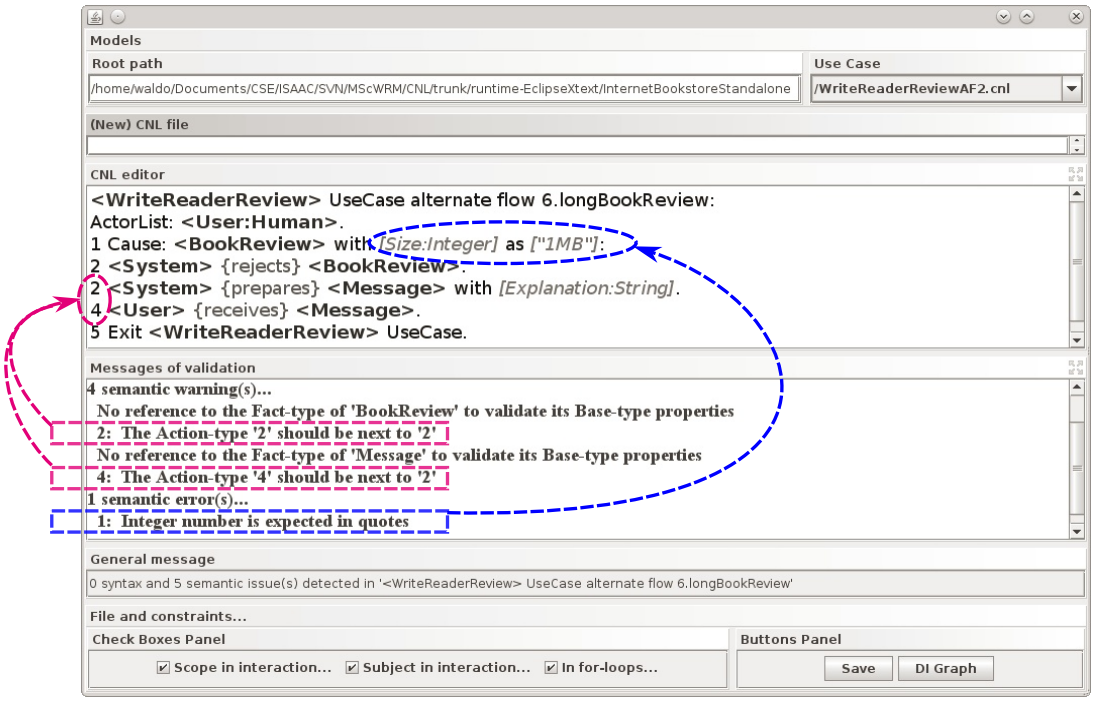
The premises specify the configuration as a `FlowType` in state $s1_{FlowType}$ with the value of each corresponding WORD and the required reserved words. If the premises are valid, the transition from `FlowType` to $\texttt{FlowType}_{UCNF}$ is deducted, where the label $UCNF$ refers to 'use case normal flow'. Similarly, the premises and conclusions could be defined for the transitions from `FlowType` to $\texttt{FlowType}_{UCAF}$ (use case alternate flow), $\texttt{FlowType}_{ONF}$ (object normal flow) and $\texttt{FlowType}_{OAF}$ (object alternate flow). These deduction rules of `FactType` could be utilized as premises for the deduction rule to validate `Flow`:

$$\frac{\langle \texttt{Flow}, (\texttt{FlowType}_{UCNF} \vee \texttt{FlowType}_{UCAF} \vee \texttt{FlowType}_{ONF} \vee \texttt{FlowType}_{OAF}), \text{hasName}, \text{flow.size} = 1 \rangle}{\texttt{Flow} \xrightarrow{VF} \texttt{Flow}_{VF}}$$

The transition labelled as VF (valid flow) represents the conclusion of $\texttt{Flow}_{VF}$ as a valid state if the premises hold. Further study is required in the analysis and specification of isolated or interoperability rules with SOS scope. As the SOS is focused on the specification of (dynamic) internal-behavior, a potential advantage of this further work would open the opportunity to design a formal transformation from the *rules* of the Requirements Model into SOS deduction rules of the Interaction Model.

In the remainder of this section, an informal description of the missing I_R rules is provided. The $\xi_{ActorList}$ validates the state of each `Noun` that composes the `ActorList` in the Ecore model. (Figure C.4). First, the *EBoolean* attribute 'preparesList' of `ActorList` indicates if the keyword 'ActorList:' is present or not. This attribute provides a message that indicates how to specify actors when this keyword is written but the list of actors is empty (which is an invalid state of `ActorList`). Afterwards, the OO polymorphism is used in the validation, because $\xi_{ActorList}$ checks that each `Noun` in the list is extended as an `Actor` type. The `ActorList` becomes invalid if a `NonActor` type is included in the list. Next, the validation reviews that no `Noun` is duplicated. This includes the checking of name duplication between `Nouns` from the `ActorList` and from the `NonActorList`. The `ActorList` achieves the state of valid if no duplication is found. Finally, $\xi_{ActorList}$ transforms the list of `Actors` as a set of `Actors`, that is, the `actorSet` attribute of the Validator component. This transformation is to achieve an easy mapping of actors for the validation of *action-types*.

The $\xi_{NonActorList}$ is similar to $\xi_{ActorList}$, as it validates the state of each `Noun` that composes the `NonActorList` (Figure C.4). The main difference is the utilization of `NonActor` instead of `Actor`.The functionality of the *EBoolean* attribute 'preparesList' is the same as in `ActorList`; the difference is the keyword that is represented by this attribute: $\xi_{NonActorList}$ utilizes 'NonActorList:' rather than 'ActorList:'. Before validating the no duplication of `NonActors`, $\xi_{NonActorList}$ reviews that the use



The semantics validation reports one error that is assigned to the `BaseType` located in the `TriggerType` (blue arrow). This error does not affect the valid state of the `ActionTypeList`, which contains the five *action-types*. However, the enumeration of these *action-types* is not sequential, which assigns two warnings to the `ActionTypeList` (magenta arrows). This prototype was utilized to simulate the communication with Symbiosis in order to prepare the loading of `NonActors` from an external data source: the `NonActorList` is not written in this flow example, but this prototype prepares it from an external data input. However, the two warnings that state *"No reference to the Fact-type of..."* reveal that the external data input is limited. This limitation restricts some χ_o rules to only use isolated vocabulary.

Figure 4.6: Preliminary prototype

case (or object) name (i.e., the WORD_a from `FlowType`) is used to specify the `NonActor` that represents the `Flow`. If this `NonActor` is not specified, a warning message is prepared. Furthermore, if the `Flow` is a use case flow, $\xi_{NonActorList}$ reviews that the `NonActor` that represents the system under development is on the `NonActorList`. This `NonActor` should utilize the name 'System' and if it is not in the list, a warning message is prepared. The `NonActorList` becomes valid if no duplication of `Nouns` is detected. Finally, the list of `NonActors` is transformed into the attribute `nonActorSet` of the Validator component.

The $\xi_{ActionTypeList}$ validates all the *action-types* (i.e., steps) of the `Flow`. These *action-types* can be `TriggerType`, `ActionType` or `EndType` (Figure C.5). In fact, as the `TriggerType` and `EndType` are extensions of `ActionType` and `ActionType` composes `ActionTypeList` (Figure 4.3), the $\xi_{ActionTypeList}$ employs $\chi_{ActionType}$ to validate any *action-type*. Furthermore, after each call to $\chi_{ActionType}$, $\xi_{ActionTypeList}$ validates that the sequence of *action-types* is correctly enumerated. This is achieved with the sequential validation of the `Rank` (i.e., the step number) via χ_{Rank}. The valid state of `ActionTypeList` is defined with the correct structure of the list of *action-types*. The state remains valid even if the enumeration is incorrect, although a warning message is prepared to report it[9]. The invalid states detected by $\chi_{ActionType}$ are assigned to the correspondent `ActionType`; an example is available in Figure 4.6. The $\xi_{ActionTypeList}$ starts with the validation of `TriggerType` as the initial *action-type*. This is achieved with the `EBoolean` attribute 'hasFirstAction' of `ActionTypeList` and $\chi_{ActionType}$. Afterwards, $\chi_{ActionType}$ is utilized to validate the list of *action-types* that follow the `TriggerType`. Finally, $\chi_{ActionType}$ validates the `EndType`.

There is no clear specification to assure that a χ_o is either an isolated or interoperability rule. This depends on the availability of the CNL Dictionary, rather than on the specifications of the χ_o rules. Thus, the Dictionary could be perceived as a *service* for the Validator. This intuition is exemplified in Figure 4.6. If the service is unavailable or partially available, the χ_o rules that need vocabulary interoperability might be limited to work as isolated rules. Oppositely, if the service is available, the χ_o rules that require it would function as interoperability rules. Appendix C.4 provides a description of each χ_o rule. Some of these descriptions highlight the advantage of utilizing reserved words in keyphrases, as they diminish the structural dependence between syntax and semantics. In other words, the utilization of keyphrases allows to extend the variability of semantics validation with less dependence on the complexity of the EBNF rules.

The next subsection initiates the discussion of the Dictionary as a service for future improvements of the Xtext CNL.

[9] This warning (and other warnings) can become an invalid state with low workload in the adjustment of the implementation. This is an advantage of MDE with structured semantic rules.

4.2.3 CNL Dictionary as middleware between Xtext CNL and Symbiosis

In general, the contents of the CNL Dictionary are (i) keywords, (ii) key phrases and (iii) vocabulary from Symbiosis. The keywords should not be modified as they are tightly tied to the syntax structure. The key phrases are formed by reserved words, verbs and regular expressions. These phrases become lists of Strings with variable values that are loosely tied to the syntax structure. The vocabulary from Symbiosis depends on each project and consists of *fact-types* and the map of flows. The key phrases and vocabulary from Symbiosis could be perceived as an ontology that is based on the projects of Symbiosis. Thus, the Dictionary as middleware could benefit the evolution of key phrases and vocabulary in multiple projects of an enterprise. Furthermore, to extend the standardization of the Dictionary, the analysis of its similarities with SBVR could motivate the evolution of the Dictionary as a standard SBVR-based service for extended compatibility.

An initial analysis of similarities between the CNL Dictionary and SBVR is presented as follows. Table 4.3 discusses a syntactical comparison. Next, the class relationships and semantic concepts are discussed. This analysis is based on the comparison of Bajwa et al. [2] between SBVR and the Object Constraint Language (OCL).

SBVR vs CNL Dictionary	Comparison
Vocabulary vs *Key phrases*	The vocabulary in SBVR utilizes keywords and user defined elements. The keywords are predefined and are an auxiliary part of the SBVR rules. The keywords and reserved words are part of the isolated vocabulary in the Dictionary. The keywords are predefined and aid in the syntax structure of the CNL. The reserved words are similar to the user defined elements in SBVR. These words can be grouped as key phrases and achieve a meaning that depends on the `EObject` that contains them. The key phrases could utilize reserved words that are based on user defined elements of SBVR practices.
Noun Concept vs *Fact-type*	In SBVR, the Noun Concept is classified as either an 'object type' or an 'individual concept'. Usually, common nouns in English are used as object types and proper nouns are used as individual concepts. The *fact-types* in Symbiosis are already classified in the Object Model. The Dictionary could offer an ontology approach to re-use *fact-types*. Moreover, the classification of nouns in the CNL could be enriched in the Dictionary. For instance, the non-actor nouns could be acknowledged as *fact-types* or flows. The actors in the CNL could be classified as proper nouns.
Verb Concept vs *Behavioral Feature*	The verb concept are commonly operations of business entities in SBVR. The Dictionary has a similar interpretation of verbs, as it stores operations of *object-types* from the behavioral features provided by Symbiosis.

Table 4.3: Syntactical comparison between SBVR and CNL Dictionary

The association, composition and generalization class relationships are prepared by the Object

Model in Symbiosis. A transformation could be applied to these relationships to store them in the Dictionary according to SBVR, that is: the associations as 'associative fact types', the compositions as 'categorization fact types' and the generalizations as 'partitive fact types'.

SBVR utilizes two basic types of rules: structural and operative. The structural rules define conditions and restrictions according to the structure of business models. The operative rules focus on the behavior of business activities and operations. The analysis of semantic rules I_R and χ_o with SOS could provide a formal representation to them that considers the structure of the semantic model and the (states of) behavior for the Use Case and Interaction Models. Thus, a transformation from the SOS to SBVR definitions might be feasible.

The complete specifications of SBVR are available in [34].

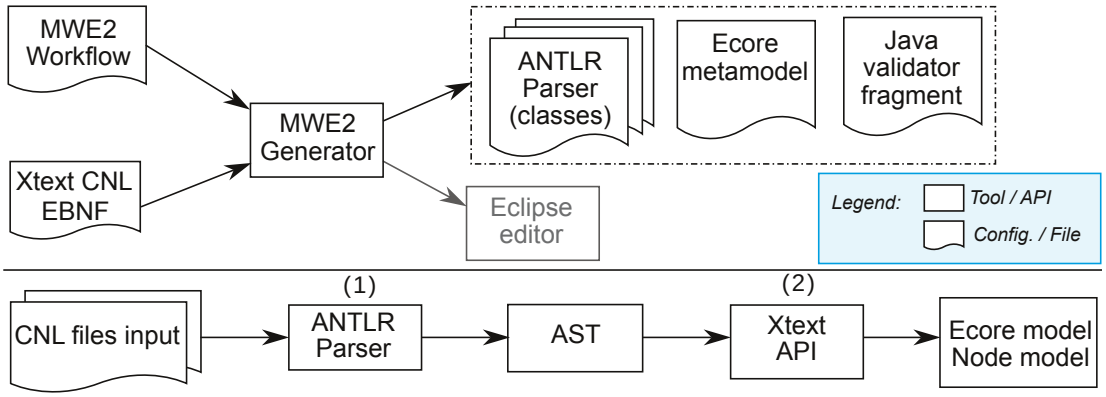
Chapter 5

Prototype design and implementation

This chapter presents the technical details about the implementation of the Xtext CNL and the Use Case Model. A preliminary case study is included to consider future modifications to complete the extension of Symbiosis.

5.1 Xtext

The utilization of Xtext has been introduced in Section 4.1 within the context of EMF and grammar rules. This section discusses the implementation of Xtext 2.4.2 to achieve the prototype for Symbiosis according to the documentation [16]. Xtext is classified as a DSL Workbench for External DSLs [22]. An External DSL requires a distinct infrastructure from the one used by the host language. The CNL (external DSL) requires a parser not used for the host language, Java. The top image of Figure 5.1 depicts that Xtext uses the *Modeling Workflow Engine 2* (MWE2)[1] generator to create the core of the CNL API. This generator uses the workflow and the CNL grammar[2] to create the *ANother Tool for Language Recognition* (ANTRL)[3] parser, the Ecore metamodel and the 'Java validator fragment' class that enables the utilization of the CNLV. The MWE2 generator also creates the Eclipse plug-ins for a CNL Eclipse editor. The general processing of plain text files with the '.cnl' extension consists of two stages, as illustrated in the bottom image of Figure 5.1.



Top image: illustrates the generation of the CNL API core and the Eclipse IDE with the MWE2 generator. Bottom image: shows the general stages to process CNL files.

Figure 5.1: Processing stages in Xtext

[1] http://www.eclipse.org/Xtext/documentation.html#MWE2
[2] The Xtext CNL EBNF is available in Appendix D.1.
[3] http://www.antlr.org/

In stage (1) the ANTLR parser processes the files and returns their ASTs. In stage (2) the Xtext API – which is based on the EMF API – uses the Ecore metamodel and transforms the ASTs into 'intermediate representations', the semantic (Ecore) and syntax (Node) models. These representations are logical abstractions which could be transformed into 'final representations', such as Java source code[4] or Ecore serialization. The inclusion of the semantics validation extends the general processing with a third stage.

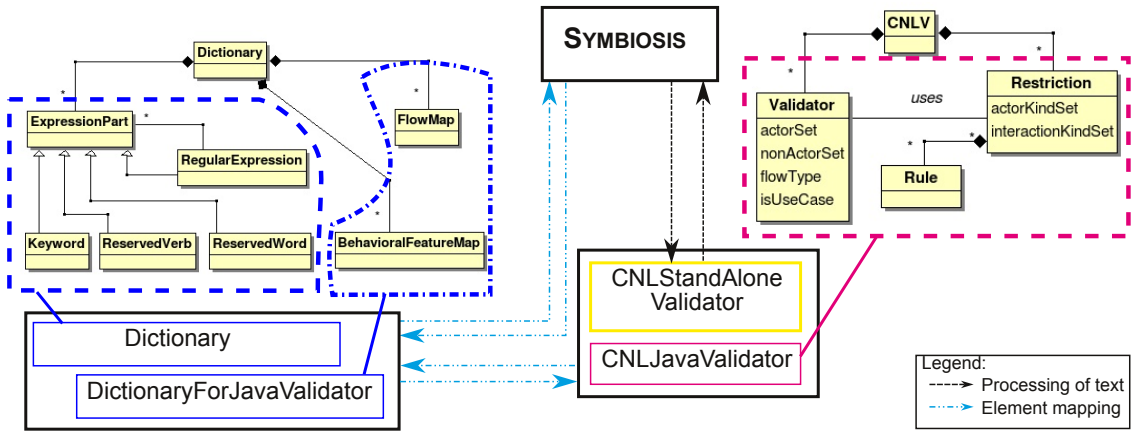


Figure 5.2: Implementation of the CNL architecture

The third stage depends on the CNL architecture (Subsection 3.2.2) with the implementation of CNLV and the Dictionary. Figure 5.2 includes the outline of these implementations. The components of CNLV are represented by the `AbstractDeclarativeValidator` class of the Xtext API. This class prepares the general configuration to access Ecore models for its validation. The 'Java validator fragment' class automatically extends the `AbstractDeclarativeValidator` with the `AbstractCNLJavaValidator` class, which particularizes the configuration for the Ecore models that result from stage (2). The implementation of CNLV is the manual extension of the `AbstractCNLJavaValidator` class with the `CNLJavaValidator` class, which implements the root isolated rules I_R, the χ_o rules, as well as the attributes of the Validator and Restriction components. Regarding I_R and χ_o rules, the Xtext offers a declarative call of rules with the `@Check` annotation. This annotation was utilized according to the design of the CNLV, that is, only the I_R rules should use this annotation. For instance,

```
(A) @Check public void checkActionTypeList( ActionTypeList instance )
(B) private void checkActionType( ActionType instance )
```

(A) corresponds to $\xi_{ActionTypeList}$ and utilizes `@Check` whereas (B) corresponds to $\chi_{ActionType}$ and does not utilize the annotation. Hence, (A) should be called by the CNLJavaValidator and (B) should be called by an I_R or χ_o rule.

The third stage is completed with the implementation of the components of the CNL Dictionary. These components are implemented with the two classes that are depicted in Figure 5.2. `Dictionary` is an abstract class that contains the isolated vocabulary[5]. This class is extended by the `DictionaryForJavaValidation` class in order to include the vocabulary of interoperability. So, to enable the send/receive information to/from Symbiosis, Figure 5.2 highlights the class that interoperates with Symbiosis: `CNLStandAloneValidator`. This class avoids the utilization of the Eclipse editor and

[4]This transformation is supported by Xtext and it could be an advantage for the Interaction Model, because the object flows could be transformed into Java code for the developers of the System.

[5]This vocabulary is available in Appendix D.3.

functions as the controller of `CNLJavaValidator` for Symbiosis. However, the implementation of `CNLStandAloneValidator` requires the configuration of **API dependencies** to obtain the CNL environment in a JVM without Eclipse. The strategy to satisfy these dependencies is discussed in the next section.

5.2 Dependency Injection pattern

The CNL API needs the Xtext API at runtime. Similarly, the Xtext API requires diverse APIs along its processing stages. This strategy consists of transitive dependencies at the granularity of objects. That is, an object may have one or more relationships of dependency with other object(s) that could be from other API and that could have other dependencies and so on. This chain of dependencies could be represented as an object graph [40] also known as **dependency graph**. A conceptual analogy to the dependency graph but with API granularity is shown in Figure 5.3. The greyish APIs are not required in the prototype for Symbiosis.

This section focuses on the design pattern used to *configure* the dependencies for the `CNLStandAloneValidator`.

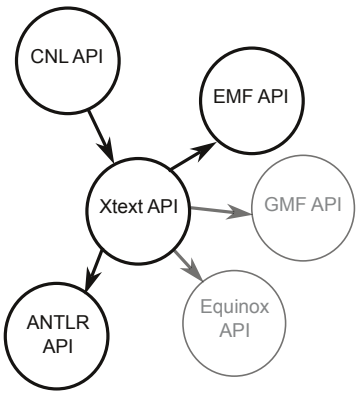


Figure 5.3: Dependency graph analogy

The **dependency injection** is a software design pattern that focuses on an efficient construction of dependency graphs in which the objects have a convenient separation of 'configuration' from 'behavior'. In this context, 'configuration' is the approach employed to 'inject' (i.e., satisfy) the dependencies that an object requires to provide its behavior. 'Behavior' represents the functionalities offered by the object. Thus, an object can be perceived as a *dependency* if its functionalities are required to fulfill the functionalities of another object. On the other hand, an object can be a *dependent* if it needs one or more dependencies to execute its functionalities. In Java, the class structure can provide an intuition of dependencies with the attributes and signatures of constructors and methods. For instance, some private attributes could become *encapsulated dependencies* if the signatures of constructors or methods do not request them. Thus, the instantiation of dependencies needs to be done in the body of constructors or methods. The encapsulation of dependencies causes that the dependent becomes the responsible to satisfy dependencies, which usually causes hard-coding rather than configuration of dependencies. Oppositely, the dependent could receive the dependencies [49] and forget about where or how to obtain them. The challenge is to formulate a configuration that constructs or finds the dependencies.

Xtext utilizes the **Google Guice framework**[6] to create the configuration and inject dependencies. Guice proposes a dependency `Injector` class that handles the 'wiring' of dependencies to create the dependency graph. In this manner, the dependency and dependent objects can focus on providing their functionalities and skip the responsibility of solving dependencies. Guice proposes a strategy of configuration that exploits benefits of the Java type-safe nature. This strategy is the specification of configuration in Java classes. These classes should implement the Guice `Module` interface and the implementations basically consist of bindings that map types of objects[7]. The `Injector` uses these bindings to wire the dependencies. The validation of the configuration (i.e., the implementations of the Module interface) is equivalent to the type check of the bindings, which is done by the Java compiler. This is the principal reason to avoid the specification of configuration with other sources (e.g., XML files).

[6] `http://code.google.com/p/google-guice/`

[7] The mapping of types is currently limited in the Java language (e.g., generics), which has been the main motivation to offer the Guice framework.

Dependent: `BillingService`

```
public interface BillingService {
    public void chargeOrder(
        ProductOrder order, int price );
    public ProductOrder provideOrder();
}
```

Execution that uses the (implemented) `BillingService`:

```
public static void main(String[] args) {
    Injector injector = Guice
        .createInjector(new BillingInjectModule());
    RealBillingService billingService =
        injector.getInstance(RealBillingService.class);
    billingService.chargeOrder(
        injector.getInstance(ProductOrder.class), 20 );
    ProductOrder order = billingService.provideOrder();
    System.out.println( order.getPrice() );
}
```

Injector

```
public class BillingInjectModule extends AbstractModule{
    @Override protected void configure() {
        bind(RealBillingService.class);
        bind(TransactionLog.class)
            .to(DatabaseTransactionLog.class);
        bind(CreditCardProcessor.class)
            .to(PaypalCreditCardProcessor.class);
    }

    @Provides ProductOrder provideOrder(){
        return new PizzaOrder();
    }
}
```

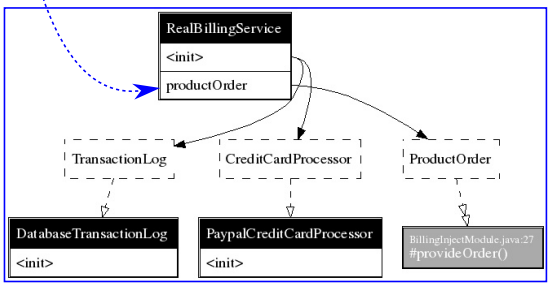
Configuration: `BillingInjectModule`. `AbstractModule` implements `Module` and is part of the Guice API. `@Provides` annotates the methods that should specify the construction of objects.

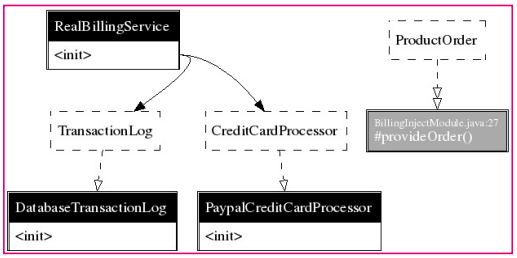
Implementation$_1$: `RealBillingService`

```
public class RealBillingService implements BillingService {
    private final CreditCardProcessor processor;
    private final TransactionLog transactionLog;
    @Inject private ProductOrder productOrder;

    @Inject public RealBillingService(
        CreditCardProcessor processor,
        TransactionLog transactionLog ) {
        this.processor = processor;
        this.transactionLog = transactionLog;
    }

    @Override public void chargeOrder(
        ProductOrder order, int price ) {
        productOrder = order;
        productOrder.setPrice(price);
    }

    @Override @Provides public ProductOrder provideOrder(){
        return productOrder;
    }
}
```

Implementation$_2$: `RealBillingService`

```
public class RealBillingService implements BillingService {
    private final CreditCardProcessor processor;
    private final TransactionLog transactionLog;
    private ProductOrder productOrder;

    @Inject public RealBillingService(
        CreditCardProcessor processor,
        TransactionLog transactionLog ) {
        this.processor = processor;
        this.transactionLog = transactionLog;
    }

    @Override public void chargeOrder(
        ProductOrder order, int price ) {
        productOrder = order;
        productOrder.setPrice(price);
    }

    @Override @Provides public ProductOrder provideOrder(){
        return productOrder;
    }
}
```





The difference between the two implementations is the injection of `ProductOrder`. Interestingly, the `Injector` needs to construct the `ProductOrder` in both cases because no binding specifies how to map this type. On the left hand side graph, the `Injector` can deduce that the `RealBillingService` depends on `ProductOrder`, whereas on the right hand side graph the dependency is not clearly specified, but the `Injector` can satisfy it due to the `@Provides` annotation. In both graphs, the solid edges represent dependencies, the dashed edges represent bindings and the double arrowed edges involve the `@Provides` annotation. In the execution, first, the `Injector` is generated by Guice with the configuration. Second, the instance of the service is created according to the required bindings. Finally, the service requests a `ProductOrder` instance to charge the price without worrying how to obtain this instance.

Figure 5.4: Dependency graphs with Guice

Furthermore, the configuration can include methods that specify an alternative to construct object instances. These methods would be utilized by the `Injector` if it needs to provide instances and the bindings do not suffice. Figure 5.4 shows a service implemented with the `Injector` in two slightly

different manners, but with the same configuration. The resulting dependency graphs are distinct. The configuration includes bindings and the construction of one dependency.

```
* generated by Xtext
package equa.project.behavior;

/**
 * Initialization support for running Xtext languages
 * without equinox extension registry
 */
public class CNLStandaloneSetup
extends CNLStandaloneSetupGenerated{

    public static void doSetup() {
        new CNLStandaloneSetup()
            .createInjectorAndDoEMFRegistration();
    }
}
```

GUICE EMF

The dependency graph for the utilization of the CNL is created with Guice in Xtext. Thus, the initialization of the `Injector` is required, such as the initialization shown in Figure 5.4 or the initialization included in the Eclipse editor. For runtime environments without the Eclipse editor, Xtext automatically generates a class that (i) creates the `Injector` and (ii) configures the EMF environment for the Ecore models. Both tasks are easily achievable with the `doSetup()` method.

Figure 5.5: Guice and EMF in Xtext without Equinox

The `CNLStandaloneSetup` (Figure 5.5) is used by the `CNLStandAloneValidator` to prepare the CNL environment for Symbiosis or another non-Eclipse environment. For example, the preliminary prototype shown in Figure 4.6 also uses the `CNLStandAloneValidator`. This preliminary prototype includes a functionality to obtain the visualization of the dependency graph for the CNL. Figure 5.6 depicts a fragment of this graph. Appendix D.4 includes an outline of the entire graph.

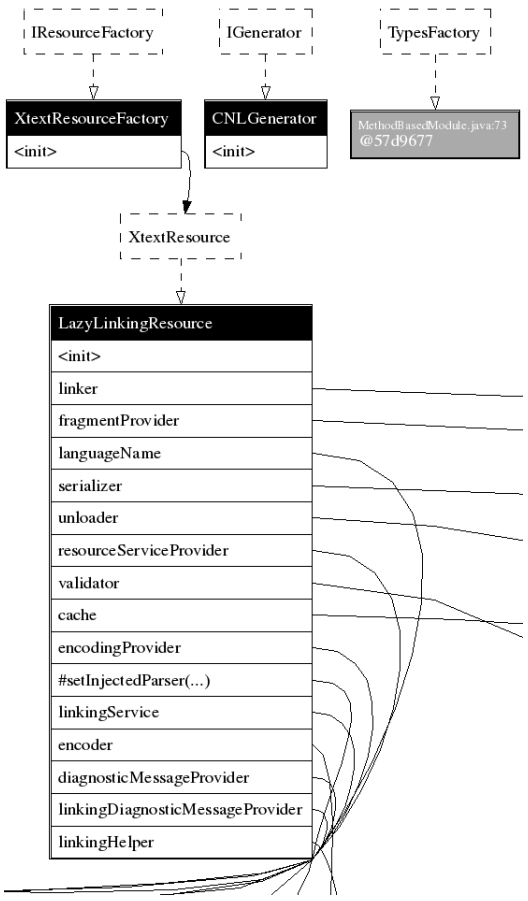
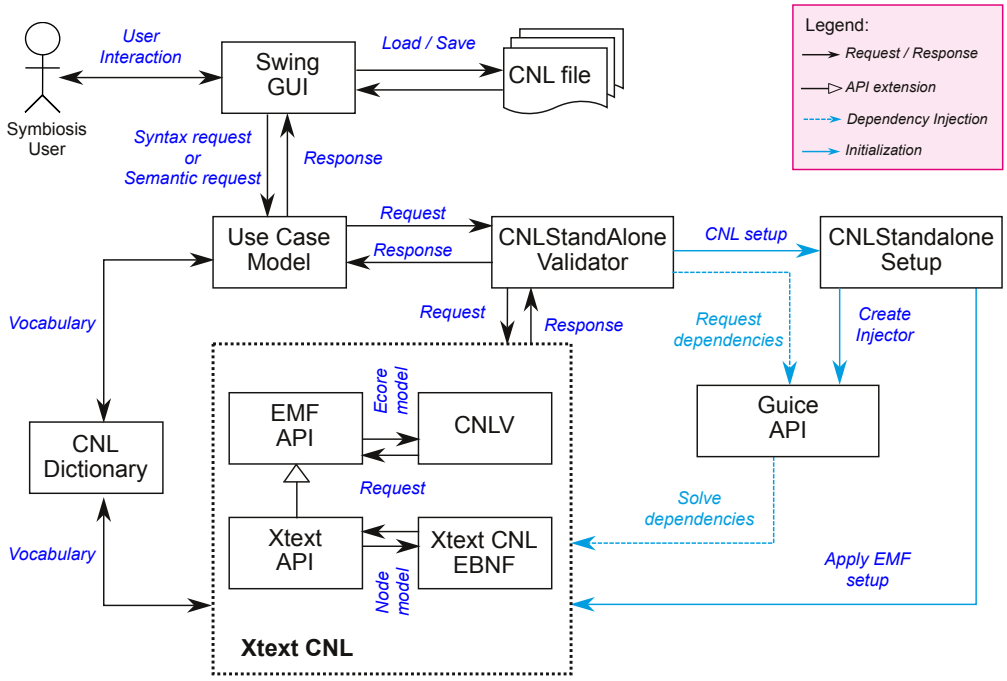


Figure 5.6: Fragment of the dependency graph for the CNL

5.3 Architecture and implementation as a Symbiosis component

From the scope of implementations, the contributions of this thesis can be perceived in two groups: the Xtext CNL and the Xtext CNL prototype for Symbiosis. The first group has been discussed in prior sections, such as the implementation of the CNL architecture in Section 5.1. The proposal to achieve the second group is now discussed. Briefly recalling, Symbiosis[8] is a stand alone application that follows the MVC design pattern in the *Java Platform Standard Edition*[9] 7 (Java SE 7) with Swing components[10] for the GUI. The technical requisite for the prototype is to be Java SE 7 compliant, which is accomplished without conflicts. The architecture of the prototype is a detailed extension of the CNL architecture (Figures 3.4, 5.2) and is depicted in Figure 5.7. The Swing components maintain the GUI structure of Symbiosis. These components send validation requests (of syntax or semantic nature) to the Use Case Model (Figure 4.4). The syntax and semantic requests are either manually caused with the load of CNL files or semi-automatically caused with modifications on the loaded CNL files. The manual requests are triggered with the specification of the root location of CNL files or with the creation of a new file. The semi-automatic requests depend on a `Timer` class that is controlled by the Swing component that displays the contents of CNL files. The `Timer` checks that after a modification of contents from the user, if a time period t without new modifications happens, then the requests of validation are sent[11]. Thus, the user implicitly request validations with the modification of contents of CNL files. This strategy requires to save the CNL file in order to send the requests. Therefore, the `CNLStandAloneValidator` utilizes temporal files. The user can save the modifications in the original files via the GUI.



The syntax requests are attended by Xtext CNL EBNF and the semantic requests are attended by CNLV.

Figure 5.7: Architecture of the prototype

[8] Additional information about Symbiosis is available in Section 1.1 and Appendix A.
[9] `http://www.oracle.com/technetwork/java/javase/overview/index.html`
[10] `http://docs.oracle.com/javase/7/docs/technotes/guides/swing/`
[11] The implementation utilizes $t = 2000$ milliseconds.

The Use Case Model updates the CNL Dictionary and uses the `CNLStandAloneValidator` to attend the validation requests. The implementation of the `CNLStandAloneValidator` uses the CNL dependencies and the EMF setup with the `CNLStandaloneSetup` class (Figure 5.5) and offers public methods to do the validations. The syntax validation is done by Xtext CNL EBNF during the load of CNL files and the results are the Node models, which are the Xtext representations of the ANTLR ASTs. These models are useful to enrich error or warning messages with raw content (i.e., values of tokens) of the CNL file. The semantic validation is achieved with the Ecore models of all the Node Models. The Ecore models are available with the `XtextResourceSet`[12] class, which maps the set of Ecore models to their URIs. The `XtextResourceSet` is employed by CNLV to execute the root semantic rules I_R (and implicitly the χ_o delegates) over the `EObjects` of the Ecore models. Furthermore, the CNL Dictionary is used in the semantic validations to review the mapping of elements between the Symbiosis and Xtext CNL. These elements are the *fact-types* and its behavioral features, as well as the use case Ecore models from the `XtextResourceSet`[13]. The `CNLStandAloneValidator` sends the response of syntax or semantic validations to the Use Case Model. This response is displayed to the user with the Swing components.

5.3.1 Use Case Model Implementation

This implementation of the Use Case Model architecture (Subsection 4.2.1) is divided in two groups: the *compliance with the Symbiosis MVC Model* and the *Swing GUI*. The first group implies that the Use Case Model should follow the implementation strategy of the Requirements and Object Models. The second group consists of the development of Swing components according to the Swing design of Symbiosis.

The implementation of *compliance with the Symbiosis MVC Model* is pursued with the extension of the Symbiosis abstract `Model` class[14], as well as with the implementation of the `Serializable` interface. The extension of `Model` allows the manipulation of the Use Case Model with the Symbiosis MVC Controller, so that the communication with the Requirements and Object Models is consistently obtained. The implementation of `Serializable` is deeply related to the file saving of Symbiosis projects. The runtime instance of the Use Case Model should be serializable in order to save its state in the file that saves the Symbiosis project. An entire serialization of the Use Case Model would imply the inclusion of the CNL files, which is not affordable due to two reasons, (i) these files are already serialized (i.e., its state is already saved) and (ii) the transformation of these files as Ecore models is responsibility of the Xtext CNL. Intuitively, a feasible alternative is the usage of *transient* attributes in the Use Case Model, mainly due to reason (i). These attributes should avoid saving the state of the CNL files. Additionally, based on reason (ii), the set of Ecore models can also avoid serialization because the Xtext CNL regenerates them from the CNL files. The elements that are related to the CNL files and are not generated by Xtext CNL are the *UC Linkage*s, which should be serialized. Briefly recalling, these elements specify the relationships of use cases with the Requirements Model, such as the *action* requirements that are linked to a use case. Thus, the proposed implementation of the Use Case Model uses transient attributes as currently discussed and serializes the *UC Linkage*s. The *Flow persistence* and *CNLDelegate* components of the Use Case Model are implemented as a separate controller class that manages the processing of CNL files and updates the CNL Dictionary.

The implementation of *Swing GUI* consists of two Swing classes, the `UseCaseViewer` and `UseCaseEditorDialog`. The `UseCaseViewer` is implemented as a `JPanel` that is accessible in Symbiosis as the tab that is red highlighted in Figure 5.8. This figure also groups the contents of this tab in four sections and describes them. The `UseCaseEditorDialog` is a pop-up `JDialog` that allows the user to modify use case flows. This `JDialog` can be accessed in two manners: from menu bar of

[12] `XtextResourceSet` and `XtextResource` are part of the Xtext API that extends the EMF API. This extension could be utilized in other EMF environments. For instance, the casting of these two classes to ResourceSet and Resource respectively, could be used for pure Ecore model manipulation without conflicts.

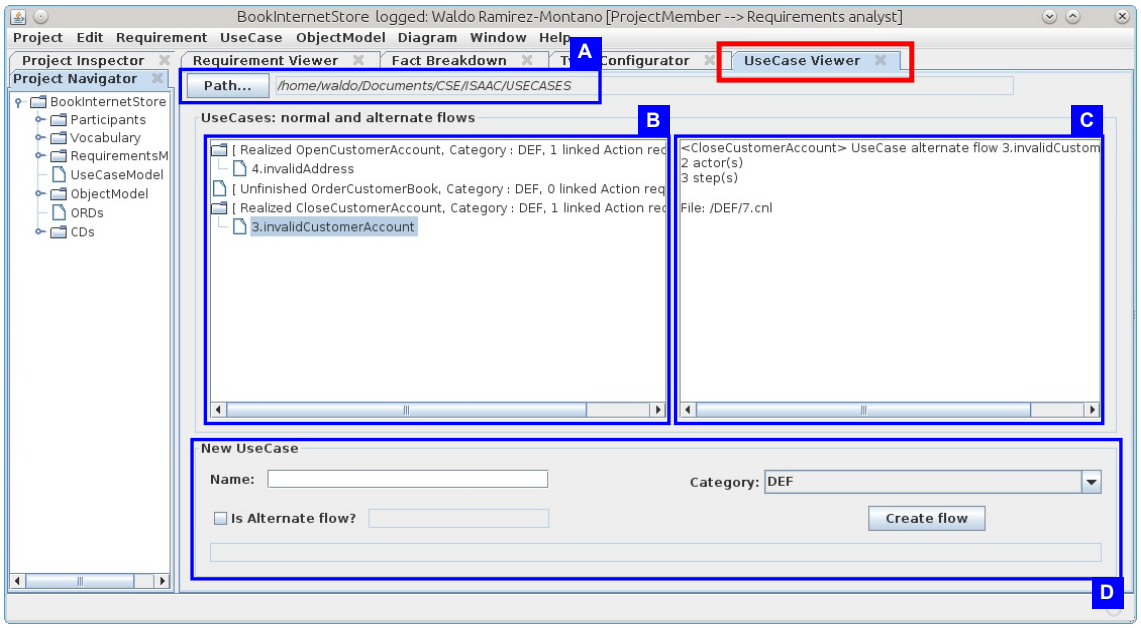
[13] The non use case Ecore models (i.e., object flows) are not mapped.

[14] The access to this class is reserved to Symbiosis project members.

Symbiosis in the 'UseCase' drop down menu, or by selecting a flow in the UseCase Viewer tab for then right-clicking on it in order to select the option 'Edit flow...' from the pop-up menu. Figure 5.9 depicts the `UseCaseEditorDialog` and groups its contents in four sections with their descriptions.

The Interaction Model could be implemented similarly. The Xtext CNL supports object flows (see Appendix C.2) and the architectures of the Use Case and Interaction Models have the same structure. Furthermore, the creation of Java source code from the object flows could be accomplished with an extension of Xtext CNL. Xtext supports a transformation of the DSLs models into source code by means of templates and the Xtend[15] language. Further research would be required to consider this transformation.

The API involved in the development of the prototype for Symbiosis is listed in Appendix D.2.



A `JButton` and `JTextField` that specify the root path that contains the CNL files.

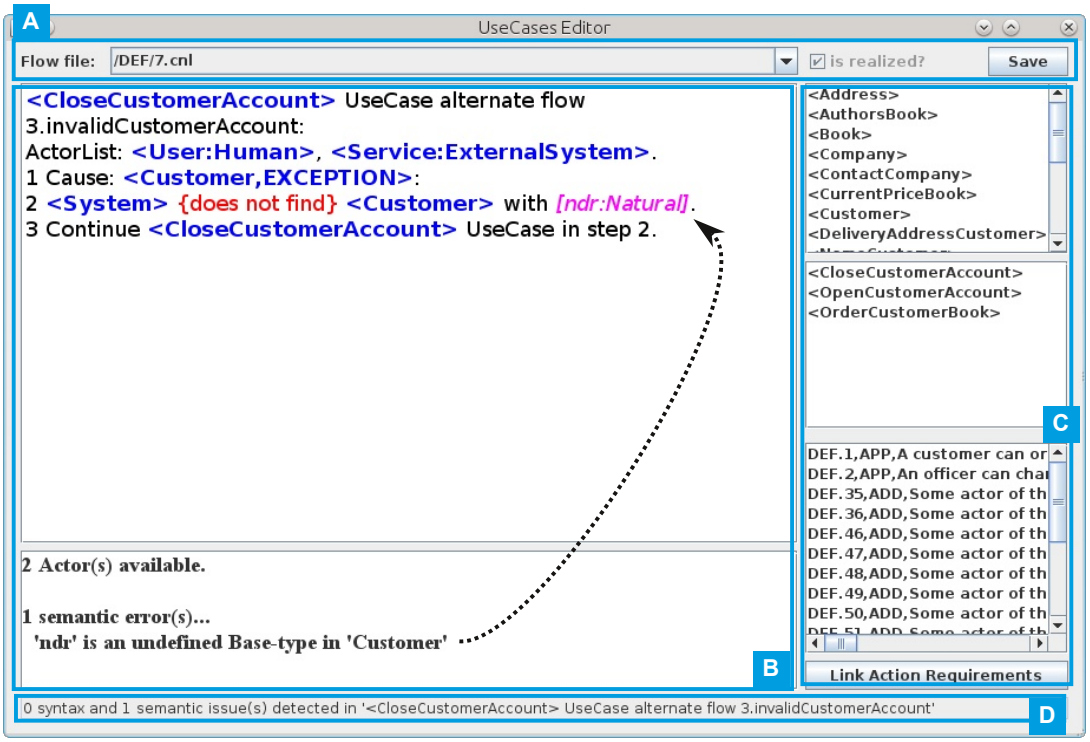
B `JTree` that organizes the use cases of the project. Each branch is a normal flow and displays the status of the use case (either realized or unfinished), its category and the amount of linked *action* requirements. Each leaf of a branch is an alternate flow and displays its label, which is the source step of the normal flow followed by a descriptive string.

C `JTextArea` that presents the flow-type, amount of actors, amount of steps and the file that contains the flow.

D `JTextField`, `JComboBox`, `JCheckBox` and `JButton` represent the fields to create a new use case flow. The `JTextField` expects the name of the use case, the `JComboBox` indicates the available *categories* from the Requirements Model, the `JCheckBox` specifies if the new flow is normal (unchecked) or alternate (checked) and the `JButton` submits the request of new use case flow.

Figure 5.8: UseCaseViewer Swing component

[15] `http://www.eclipse.org/xtend/`



A `JComboBox`, `JCheckBox` and `JButton`. The `JComboBox` allows to select which flow to visualize. The `JCheckBox` permits to specify if the use case is realized (checked) or unfinished (unchecked). This specification can only be done in normal flows (i.e., it becomes read-only for the alternate flows). The `JButton` saves the changes of the flow.

B `JTextPane` and `JTextArea` that represent the main request and response Swing components for the Use Case Model. The `JTextPane` is the text editor for the contents of the flow and utilizes the following style of font colors: blue for nouns, magenta for *base-types* and red for verbs. The `JTextArea` is a read-only area that displays the responses from the Use Case Model. This figure shows a semantic error caused by a *base-type* that does not exist in the *fact-type* 'Customer'.

C Three read-only `JLists` and a `JButton`. The `JLists` display related project information (from top to bottom): list of *fact-types*, list of use cases and list of *action* requirements with the same category than the current use case. Several *action* requirements can be selected so that the `JButton` links them with the current use case.

D Read-only `JTextField` that specifies the amount of syntax and semantic issues (i.e., warnings and errors) are in the current state of the flow.

Figure 5.9: UseCaseEditorDialog Swing component

5.4 Preliminary case study

The preliminary case study simulates a domain of reality with requirements for a book-store system over the Internet. The Symbiosis prototype is utilized for the conceptual modeling of this domain in a project named *BookInternetStore*. The Requirements and Object Models have been previously generated in order to pursue the experimentations of use case modeling. An excerpt of these models[16] is available in Table 5.1. Before the experimentations, consider the following exemplification of the abstraction of the domain with Symbiosis. First, the text 'Book 87104 has currently a price of 22.50 euro' is a fact of the domain. This text is manually registered in the Requirements Model as the *fact* requirement DEF.6. 'DEF' is the default category of requirements in Symbiosis. Finally, the semi-automatic breakdown of DEF.6 is done to register the *fact-type* `<CurrentPriceBook>` into the Object Model. This *fact-type* represents the relationship between the *base-type* `<price:Real>` and the *object-type* `<Book>`[17]. This example recalls the non-deterministic nature of this approach, because part of the criteria of abstraction depends on decisions of the users of Symbiosis. The manual registration of *quality*, *rule* and *action* requirements is similar to the registration of *fact* requirements and they are not transformed with breakdowns.

Some *rule* requirements are derived from the constraints of static-behavior in the Object Model. These constraints are focused to guarantee the elementariness of the Object Model so that this model can be utilized in use case modeling.

NAME	FACT REQUIREMENT FROM THE REQUIREMENTS MODEL
DEF.4	The price of book 9815 on order 27 is 23.50 euro.
DEF.6	Book 87104 has currently a price of 22.50 euro.

NAME	ACTION REQUIREMENT FROM THE REQUIREMENTS MODEL
DEF.52	A fact about "There are currently <stock : Natural> pieces of <book : Book> in stock." may change after input of such a fact.
DEF.82	Some actor of the (sub)system must get the opportunity to add a fact of <Book> later on.

NAME	RULE REQUIREMENT FROM THE REQUIREMENTS MODEL
DEF.18	Two (or more) facts about <Book> with the same value on <isbn : String> are not allowed.
DEF.30	Every <Book> cannot exist without a fact about "<book : Book> has currently a price of <price : Real> euro.", without any consideration.

KIND	NAME	FACT-TYPE EXPRESSION FROM THE OBJECT MODEL
OT	Book	book <isbn : String>
FT	CurrentPriceBook	<book : Book> has currently a price of <price : Real> euro.

The *quality* requirements are out of scope in this research. The processing of *rule* requirements as constraints of the dynamic-behavior of objects is part of further research with the Interaction Model. The linkage of *action* requirements with use cases is proposed in this research.

Table 5.1: Excerpt of the Requirements and Object Models

The total amount of *action* requirements is 18 (Table E.1), from which the (dynamic) external-behavior is specified according to the changes of state of *fact-types*. These changes are analogous to the CRUD storage persistence. For example, consider the *action* DEF.82 from Table 5.1, which is analogous to the CRUD 'create' function for the `<Book>` *fact-type*. The use case that is linked to DEF.82 is `<AddBook>` and consists of two flows. The normal flow (left hand side of Table 5.2) is the ideal behavior in which the step 4 is linked to the DEF.82. In addition, it was noticed that DEF.52 (Table 5.1) could be also linked to this use case. The text in quotes in DEF.52 is the expression that represents

[16] Complete perspectives of these models are available in Appendix E.

[17] The *object-type* is the sub-type of *fact-type* that does not represent a relationship between other *fact-types*

Linked action requirements: DEF.82, DEF.52.

<AddBook> UseCase: ActorList: <customerServiceClerk:Human>. 1 <AddBook> starts: 2 <customerServiceClerk> {provides} <Book>. 3 <System> {reviews} <Book,EXISTS>. 4 <System> {adds} <Book>. 5 <System> {prepares} <StockBook> "of"<Book>. 6 <customerServiceClerk> {completes} <StockBook> with [stock:Natural]. 7 Post <StockBook,CREATED>.	<AddBook> UseCase alternate flow 3.bookExists: 1 Cause: <Book,EXISTS>: 2 <System> {displays} "existent"<Book>. 3 Exit <AddBook> UseCase.

Some of the related definitions:
Fact requirement DEF.12: *There are currently 45 pieces of book 23491 in stock.*
Type expression of `<StockBook>` : *There are currently <stock : Natural> pieces of <book : Book> in stock.*

Table 5.2: `<AddBook>` use case

Linked action requirements: DEF.2, DEF.47, DEF.48.

<ChangePriceOfBook> UseCase: ActorList: <officer:Human>. 1 <ChangePriceOfBook> starts: 2 <officer> {provides} <Book> with [isbn:String]. 3 <System> {searches} <Book>. 4 <System> {displays} <CurrentPriceBook>. 5 <officer> {updates} <CurrentPriceBook> with [price:Real]. 6 Post <CurrentPriceBook,UPDATED>.	<ChangePriceOfBook> UseCase alternate flow 3.invalidBook: ActorList: <officer:Human>. 1 Cause: <Book,EXCEPTION>: 2 <System> {notifies} "nonexistent"<Book>. 3 Continue <ChangePriceOfBook> UseCase in step 2.
	<ChangePriceOfBook> UseCase alternate flow 2.changeCancellation: ActorList: <officer:Human>. 1 Cause: <ChangePriceOfBook,EXCEPTION>: 2 <officer> {cancels} <ChangePriceOfBook>. 3 Exit <ChangePriceOfBook> UseCase.

Some of the related definitions:
Fact requirement DEF.6: *Book 87104 has currently a price of 22.50 euro.*
Type expression of `<CurrentPriceBook>`: *<book : Book> has currently a price of <price : Real> euro*.

Table 5.3: `<ChangePriceOfBook>` use case

the *fact* requirement DEF.12 (Table E.1). This expression corresponds to the *fact-type* `<StockBook>` (Table E.3). Thus, the input of the `<StockBook>` is pursued with the steps 5 and 6. On the other hand, an alternate flow (right hand side of Table 5.2) could be initiated according to step 3 of the normal flow. The predicate 'EXISTS' of the `<Book>` represents that the *fact-type* has been added previously and thus, it should not be duplicated.

The actor of the `<AddBook>` use case is `<customerServiceClerk>`, which is one of the three human actors that have been proposed. The second actor is explicitly mentioned in DEF.2 (Table E.1) as the 'officer' who can change the price of a book. The `<ChangePriceOfBook>` use case has been designed for that purpose, as shown in Table 5.3. The *fact* requirement that is the source of the *fact-type* `<CurrentPriceBook>` is included in the comments of this table. The update of this *fact-type* links the use case with the *action* requirements DEF.2, DEF.47 and DEF.48 (Table E.1). The last two basically refer to the same behavior of DEF.2, but without a particular actor. The `<ChangePriceOfBook>` use case includes two alternate flows. Both of them are caused by an unexpected change of state that is additional to the CRUD analogies. The prototype represents it as the 'EXCEPTION' predicate for *fact-types* and use cases. The first exception is the request of an nonexistent `<Book>` in the step 3 of the normal

The equivalence to «invokes» in the CNL is 'calls:'

<StartCustomerSession> UseCase: ActorList: <client:Human>. 1 <StartCustomerSession> starts: 2 <client> {provides} "credentials of"<Customer>. 3 <System> {acknowledges} <Customer>. 4 <StartCustomerSession> calls: <AddOrder> or calls: <RemoveOrder> or calls: <EndCustomerSession> or calls: <UpdateContactCompany> or calls: <UpdateDeliveryAddress>. 5 Continue in step 4.	<StartCustomerSession> UseCase alternate flow 3.invalidCustomer: 1 Cause: <Customer,EXCEPTION>: 2 <System> {replies} "invalid"<Customer>. 3 Exit <StartCustomerSession> UseCase. <EndCustomerSession> UseCase: ActorList: <client:Human>. 1 <EndCustomerSession> starts: 2 <client> {terminates} <StartCustomerSession>. 3 Exit <StartCustomerSession> UseCase.

The five use cases 'invoked' in `<StarCustomerSession>` are designed for the human actor `<client>`.

Table 5.4: `<StartCustomerSession>` and `<EndCustomerSession>` use cases

The equivalence to «precedes» in the CNL is 'expects:'

<AddCustomer> UseCase: ActorList: <client:Human>. 1 <AddCustomer> starts: 2 <AddCustomer> expects: <PrepareAddress>. 3 <System> {creates} <Customer>. 4 <System> {creates} <DeliveryAddressCustomer> "with given"<Address>. 5 <client> {provides} <NameCustomer>. 6 Post <Customer,CREATED>.	<PrepareAddress> UseCase: ActorList: <client:Human>, <locatorService:ExternalSystem>. 1 <PrepareAddress> starts: 2 <client> {provides} <Address> with [street:String], [nr:String], [zip:String], [city:String], [country:String]. 3 <System> {checks} <Address,EXISTS> "with the"<locatorService>.

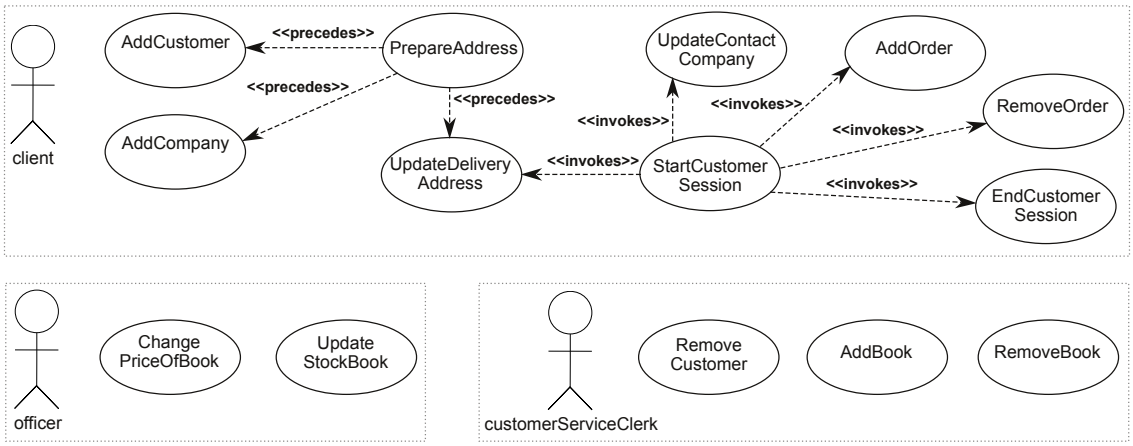
`<AddCustomer>` has no alternate flows and `<PrepareAddress>` has two alternate flows, one for an invalid `<Address>` and another for the `<client>` to cancel the use case. These flows are available in Appendix E.3.

Table 5.5: `<AddCustomer>` and `<PrepareAddress>` use cases

flow. The second exception is a non-common behavior, which is labelled as 'changeCancellation' from the step 2 of the normal flow: the `<officer>` prefers to cancel the `<ChangePriceOfBook>` rather than to (re)specify the `<Book>`.

The third human actor is proposed as `<client>` and it participates in the majority of use cases. Interestingly, this actor is represented by the *fact-type* named `<Customer>` (or `<Company>`) in order to identify the relationships between the `<client>` and other *fact-types* such as `<Order>`. The only exceptions are the use cases where the `<Customer>` (or `<Company>`) are created. To avoid duplication of behavior, the `<StartCustomerSession>` use case is proposed. This use case utilizes the equivalence in the CNL to the «invokes» use case relationship proposed by ICONIX. Furthermore, the utilization of 'particularization' in nouns (i.e., the quoted text before nouns, such as in the step 2 of `<StartCustomerSession>`) could be considered to add vocabulary (e.g., new attributes for an *object-type*) or more predicates for the *fact-types* or use cases (e.g., "invalid" in the step 2 of the alternate flow of `<StartCustomerSession>`).

An additional actor is proposed but as an external system. This actor simulates a service that is utilized in the `<PrepareAddress>` use case to confirm the existence of an `<Address>` *fact-type*. This use case «precedes» the `<AddCustomer>` use case, as shown in Table 5.5. In `<PrepareAddress>`, the `<client>` actor does not require a representation with `<Customer>`. Thus, this use case is not invoked by `<StartCustomerSession>`. The `<locatorService>` is the external system and the `<System>` uses it to confirm the correctness of `<Address>`. Afterwards, `<AddCustomer>` continues to create the `<Customer>` for the `<client>`.



<PrepareAddress> avoids the duplication of behavior in three use cases. <UpdateDeliveryAddress> requires the precedence of <PrepareAddress> upon the calling of <StartCustomerSession>.
The total amount of (normal and alternate) flows is 29; they are available in Appendix E.3.

Figure 5.10: Use Cases diagram

USE CASE	LINKED ACTION(S)
<AddCustomer>	DEF.84.
<AddCompany>	DEF.84.
<PrepareAddress>	—
<UpdateDeliveryAddress>	DEF.49.
<UpdateContactCompany>	DEF.46.
<StartCustomerSession>	—
<AddOrder>	DEF.1, DEF.35, DEF.50, DEF.86.
<RemoveOrder>	DEF.36, DEF.87.
<EndCustomerSession>	—
<ChangePriceOfBook>	DEF.2, DEF.47, DEF.48.
<UpdateStockBook>	DEF.51, DEF.53.
<RemoveCustomer>	DEF.85.
<AddBook>	DEF.52, DEF.82.
<RemoveBook>	DEF.83.

The use cases with 0 linked *actions* represent common behavior. This modularity focuses on the re-usability of use cases to avoid the duplication of behavior.

Table 5.6: Linkage between use cases and *action* requirements

The total amount of modeled use cases is 14, which are displayed in Figure 5.10. The linkage between use cases and *action* requirements is shown in Table 5.6. Each use case is linked to 0 or more *action* requirements, where the use cases linked to 0 represent common behavior. Oppositely, each *action* requirement should be linked to 1 use case to reduce replication of behavior or ambiguity. The exception is DEF.84 and the reasons are that (i) the <Company> *fact-type* is an extension of the abstract <Customer> *fact-type* and (ii) no *action* requirement explicitly specifies the addition of a <Customer> as a <Company> although there is an *action* requirement (DEF.46) that requests the update of information related to a <Company>. This exception could be discussed with the project members and stakeholders to resolve it. In fact, this is one of the results with the proposed use case modeling: the possibility to discuss the Requirements and Object Models according to the modeling of external behavior.

Additional points conclude the discussion of the preliminary case study:

→ The use cases with CNL are used to model the flow of the behavior rather than the description of the behavior. The details in the flows depend on the vocabulary that the prototype can utilize.

→ Suggestions of ICONIX are feasible in use case modeling with the CNL. The highlighted suggestions are: simple sentences (noun-verb-noun) with active voice, the «invokes» and «precedes» relationships and the request-response scope of use cases to represent the interaction between the system and actors.

→ The proposal of new *actions* or *fact-types* might arise during use case modeling. Moreover, the possibility to (re-)define *action* requirements for common behavior in the Requirements Model might be feasible.

→ In addition to the classification of use cases per category of requirements, a classification per actors could provide an organization from the behavioral point of view. This classification could be used by *action* requirements.

→ In the CNL for use cases, most of the verbs are freely defined by the user according to the understanding to *action* requirements. Further work is feasible to limit the available verbs according to a classification of verbs. For instance, in a Symbiosis project, a cumulative classification of verbs as CRUD verbs could concretize the actual objective of the verbs in the system and could suggest new classifications. The 'EXCEPTION' predicate in the prototype is an example that extends CRUD to CRUDE for the predicates of *fact-types*.

→ The predicates of *fact-types* are utilized to represent some of their possible states. These states 'predict' behavior in the use cases. These predictions define alternate flows.

→ A simulation of the flows of use cases could test the possible traces of use cases. The CNL partially contributes with the validation of enumeration and cross-reference. Further work could create a simulator/tester that 'follows' the enumerations and cross-references to detect (in)correct behavior.

Chapter 6

Conclusions and Future work

This chapter concludes with a general overview of the research, followed by concrete review of the research questions, which include information about future work.

The *Object-Role Modeling* (ORM) and *Model Driven Design* (MDD) techniques of the EQuA framework are exploited by the project members or stakeholders in the development of a system with the Symbiosis tool. This tool semi-automatically transforms user requirements in *Natural Language* (NL) into formal models. These requirements define the domain of reality for the system and their transformations are the Requirements and Object Models. The first model establishes *facts*, *actions*, *rules* and *quality attributes* as requirements. The latter model represents the *facts* as *fact-types*, which compose the Object Oriented abstraction of the domain of reality. This abstraction includes the static behavior and the class diagram of the system. The *actions* include tasks between users and the system. These tasks describe the dynamic behavior in an informal manner. The *rules* impose constraints of static or dynamic behavior in the system. The *quality attributes* stipulate non functional requirements.

This research mainly contributes with formal model representations of use cases that are linked to the *actions* of the Requirements Model. These models are the Xtext CNL and the Use Case Model. The first model uses *Controlled Natural Language* (CNL) to propose a restricted grammar structure of the English Language with vocabulary that can be extended by Symbiosis. The second model joins to the Requirements and Object Models in order to manually link CNL use cases with *actions*. In both models, some of the suggestions of ICONIX have been considered to follow a scope of industrial practices. The implementation of these models has been achieved as a prototype in Symbiosis that uses Xtext with dependency injection configured via Guice. This prototype allows the project members and stakeholders to model use cases that represent the flows of dynamic external-behavior between actors and the system. A preliminary case study has confirmed the functionality of the prototype, highlighting the traceability between the vocabulary of Symbiosis and the CNL, the re-usability of use cases and the linkage between use cases and *actions*.

Additional contributions have opened the possibility to utilize the CNL and the Interaction Model to model the dynamic internal behavior. This is future work that focuses on behavior between *fact-types*. The validation of this behavior could be improved with Structural Operational Semantics specifications. Furthermore, some functionalities of Xtext might be considered to transform the CNL flows of objects into the base source code of the system under development.

RQ$_1$ *How can the Use Case and Interaction Models be designed based on the current Object Model?*

→ Is it feasible to propose one model-driven formalism that is re-used in both cases?
The Xtext CNL is the model-driven formalism that models dynamic internal- and external-behaviors. The implementation of the Use Case Model utilizes the Xtext CNL. The proposed design of the Interaction Model re-utilizes the Xtext CNL and could follow a similar implementation. As future work, this implementation would confirm the re-utilization of the Xtext CNL.

→ What *cardinalities* to use between use cases and the requirements in the Requirements Model?
The linkage between the use cases and the (*action*) requirements is managed by the Use Case Model. The preliminary case study suggests that the cardinality between use cases and *action* requirements should be of 0 to many. A use case without linkage corresponds to re-usable behavior, that is, 'auxiliary' use cases. In fact, ICONIX suggests the extraction of common behavior as use cases that are utilized with the «precedes» and «invokes» relationships. The case study confirms this pattern of behavior, as it avoids the duplication of behavior.
The opposite cardinality should be 1. If an *action* requirement has more than one linked use case, it could be decomposed into simpler *action* requirements or its linked use cases might be duplicating behavior.

→ In the Interaction Model, how to represent the sequential communication between objects of the Object Model?
Intuitively, each object life-cycle could be modeled with normal and alternate CNL flows. The scopes of interaction 'calls:' and 'expects:' could be utilized to represent the communication between many life-cycles. This intuition is exemplified in Appendix C.2. Further work would apply this communication between objects of the Object Model.

→ How to categorize the use cases according to the Requirements Model?
The prototype obliges the selection of a category that is defined in the Requirements Model. This functionality allows the automatic filtering of candidate action requirements that could be linked to the use case.

→ How to utilize a Natural Language approach with the Object Model that becomes the source of vocabulary for the Use Case and Interaction Models? How to validate the syntax and semantic of the Natural Language approach?
The `CNLStandAloneValidator` allows the communication between the Use Case Model and the CNL Dictionary to provide the vocabulary of the Object Model. As future work, the Interaction Model could extend the vocabulary with information related to *rules* of the Requirements Model. The `CNLStandAloneValidator` aids the Xtext CNL to receive the requests of the prototype and apply the required validations, as well as to send the responses to the prototype.

→ How to acknowledge external sources of behavior in use cases? The use case actors should be included in the Use Case Model, but how they interact with the Object Model?
The abstraction of actors as typed nouns of the CNL allow their recognition and inclusion in the Use Case Model. This approach is also applied to *fact-types* and use cases. Moreover, these nouns form part of Noun Phrases that permit the specification of details about the nouns. For instance, the *base-types* of *fact-types* or the 'particularization' of nouns. The interaction between nouns is represented by a compact syntax (mostly noun-verb-noun sentences) and regulated with semantics validation.

→ How could the Interaction Model foresee the operations of objects, as well as to scrutinize the possibility to add new operations?
The CNL Dictionary permits to Symbiosis the storage of behavioral features of *fact-types*, which include the operations of objects. The Xtext CNL could propose new operations in the CNL Dictionary so that the Interaction Model, in Symbiosis, could examine them and possibly include them in the Object Model.

→ How to maintain the traceability between the formalization of dynamic-behavior and the EQuA models?
This traceability can be perceived in three scopes: traceability with facts, actions and rules. The first scope is achieved with the CNL Dictionary, as it guarantees that the *fact-types* utilized in flows are located in the Object Model. The second scope is accomplished with the linkage proposed in the Use Case Model (or Interaction Model), as it connects the flows with actions (or objects). The third scope requires further work, specially for the Interaction Model, as it would represent the constraints of rules in the flows of objects. This last scope is related to the $\mathbf{RQ}_2$.

RQ$_2$ *How can we handle the [manually added] rules?*

→ A portion of rules is automatically generated by EQuA. Is it convenient to make the revision of these rules as part of the semantics validation?
Part of the rules that are automatically generated by Symbiosis represent constraints for registries of *fact-types*. These registries could be implemented with design patterns (e.g., factories) and would not need the semantics validation. On the other hand, the rest of automatically generated rules and the semi-automatically generated rules should be considered as part of semantics validation because they represent constraints of the structure of the Object Model (i.e., static behavior). Further work is needed to include these rules in the semantics validation.

→ The rest of rules are manually added by the user. What could be utilized to extend the validation of rules to include new rules?
An approach similar to the formalization of flows could be applied to these rules by providing a CNL that limits the structure of the manually added rules. In this manner, their transformation into parts of the flows of behavior might be simpler. Even their input directly into the flows could be an alternative. Additional discussion and research is suitable for these rules.

Bibliography

[1] Andova, S., van den Brand, M. G. J., Engelen, L. J. P. and Verhoeff, T. (2012). MDE Basics with a DSL Focus. In *SFM*. Vol. 7320 of *Lecture Notes in Computer Science*. Springer. pp. 21–57. 33

[2] Bajwa, I. S., Bordbar, B. and Lee, M. G. (2011). SBVR vs OCL: A Comparative Analysis of Standards. In *Proceedings of the 14th IEEE International Multitopic Conference (INMIC 2011)*. IEEE. pp. 261–266. 37

[3] Bajwa, I. S., Samad, A. and Mumtaz, S. (2009). Object oriented software modeling using NLP based knowledge extraction. *European Journal of Scientific Research* **vol. 35,** pp. 22–33. 18

[4] Bakema, G., Zwart, J. P. and van der Lek, H. (2002). *Fully Communication Oriented Information Modeling (FCO-IM)*. FCO-IM Consultancy. 9

[5] Bézivin, J., Brunette, C., Chevrel, R., Jouault, F. and Kurtev, I. (2005). Bridging the Generic Modeling Environment (GME) and the Eclipse Modeling Framework. In *In Proceedings of the OOPSLA Workshop on Best Practices for Model Driven Software Development*. Vol. 5. 27

[6] Bibliowicz, A. and Dori, D. (2012). A graph grammar-based formal validation of object-process diagrams. *Software & Systems Modeling* **vol. 11,** pp. 287–302. 11

[7] Bollen, P. (2008). SBVR: A fact-oriented OMG standard. In *Proceedings of the OTM Confederated International Workshops and Posters on On the Move to Meaningful Internet Systems: 2008 Workshops*. OTM'08. Springer-Verlag. pp. 718–727. 11

[8] Bollen, P. (2009). The orchestration of fact-orientation and SBVR. In *Enterprise, Business-Process and Information Systems Modeling*. Vol. 29 of *Lecture Notes in Business Information Processing*. Springer. pp. 302–312. 12

[9] Bollen, P. (2013). Enterprise resource planning requirements process: The need for semantic verification. In *Innovation and Future of Enterprise Information Systems*. Eds. F. Piazolo and M. Felderer. Vol. 4 of *Lecture Notes in Information Systems and Organisation*. Springer Berlin Heidelberg. pp. 53–67. 2

[10] Brill, E. (1992). A simple rule-based part of speech tagger. In *Proceedings of the third conference on Applied natural language processing*. ANLC '92. Association for Computational Linguistics. pp. 152–155. 18

[11] Brunekreef, J. (2010). Early Quality Assurance in Software Production. *EQuA website (Visited: Sep 2013)* `http://www.equaproject.nl/`. 1, 11

[12] Cheung, K. and Chow, K. (2007). A petri net based method for refining object oriented system specifications. *Electronic Notes in Theoretical Computer Science* **vol. 187,** pp. 161 – 172. 12

[13] De Marneffe, M. C., Maccartney, B. and Manning, C. D. (2006). Generating typed dependency parses from phrase structure parses. In *Proceedings of Languare Resources Evaluation Conference*. Ed. ELRA. Fifth International Conference LREC. pp. To–be–defined. 18

[14] DEEPTIMAHANTI, D. K. AND SANYAL, R. (2011). Semi-automatic generation of UML models from natural language requirements. In *Proceedings of the 4th India Software Engineering Conference*. ISEC'11. ACM. pp. 165–174. 13, 14

[15] DÍAZ, I., MORENO, L., FUENTES, I. AND PASTOR, O. (2005). Integrating natural language techniques in OO-Method. In *Proceedings of the 6th international conference on Computational Linguistics and Intelligent Text Processing*. CICLing'05. Springer-Verlag. pp. 560–571. 11

[16] ECLIPSE.ORG (2013). Xtext 2.4 documentation. *Xtext website (Visited: July 2013)* `http://www.eclipse.org/Xtext/documentation.html`. 27, 39

[17] EVANS, K. (2005). Requirements engineering with ORM. In *On the Move to Meaningful Internet Systems 2005: OTM 2005 Workshops*. Eds. R. Meersman, Z. Tari, and P. Herrero. Vol. 3762 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg pp. 646–655. 8

[18] FAHLAND, D. (2009). Oclets - scenario-based modeling with Petri nets. In *Proceedings of the 30th International Conference on Petri Nets and Other Models Of Concurrency*. Eds. G. Franceschinis and K. Wolf. Vol. 5606 of *Lecture Notes in Computer Science*. Springer-Verlag. pp. 223–242. 12

[19] FAHLAND, D., DE LEONI, M., VAN DONGEN, B. AND VAN DER AALST, W. (2011). Many-to-many: Some observations on interactions in artifact choreographies. In *Proceedings of the 3rd Central-European Workshop on Services and their Composition, Services und ihre Komposition, ZEUS 2011*. Eds. D. Eichhorn, A. Koschmider, and H. Zhang. Vol. 705 of *CEUR Workshop Proceedings*. `http://CEUR-WS.org/`. pp. 9–15. 12, 13, 71

[20] FAHLAND, D. AND WOITH, H. (2009). Towards process models for disaster response. In *Business Process Management Workshops*. Eds. D. Ardagna, M. Mecella, and J. Yang. Vol. 17 of *Lecture Notes in Business Information Processing*. Springer Berlin Heidelberg. pp. 254–265. 12

[21] FOWLER, M. (2010). *Domain Specific Languages* 1st ed. Addison-Wesley Professional. 25, 27

[22] GHOSH, D. (2011). *DSLs in Action* 1st ed. Manning Publications Co. 39

[23] HAGEMEIJER, M. (2013). *EQuA - Symbiosis: multi user aspects*. ISAAC, Software Solutions. 2

[24] HALPIN, T. (1993). What is an elementary fact? In *Proceedings of the 1st NIAM-ISDM Conference*. G.M. Nijssen & J. Sharp, URL (May 2013): `http://www.orm.net/pdf/ElemFact.pdf`. 9

[25] HALPIN, T. (1998). Object-Role Modeling (ORM/NIAM). In *Handbook on Architectures of Information Systems*. Springer-Verlag. pp. 81–102. 1, 9

[26] JORGENSEN, J. AND BOSSEN, C. (2004). Executable use cases: requirements for a pervasive health care system. *Software, IEEE* **vol. 21,** pp. 34–41. 13

[27] KERN, H. AND KÜHNE, S. (2007). Model Interchange between ARIS and Eclipse EMF. In *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM'07)*. No. TR-38 in Computer Science and Information System Reports, Technical Reports University of Jyväskylä. pp. 105–114. 27

[28] MNKANDLA, E. AND DWOLATZKY, B. (2004). A survey of agile methodologies. *The transactions of the SA institute of electrical engineers.* pp. 236–247. 15, 16

[29] MONTES, A., PACHECO, H., ESTRADA, H. AND PASTOR, O. (2008). Conceptual model generation from requirements model: A natural language processing approach. In *Proceedings of the 13th international conference on Natural Language and Information Systems: Applications of Natural Language to Information Systems*. NLDB '08. Springer-Verlag. pp. 325–326. 11

[30] MORDECAI, Y. AND DORI, D. (2013). I5: A model-based framework for architecting system-of-systems interoperability, interconnectivity, interfacing, integration, and interaction. In *Proceedings of the 23rd Annual INCOSE International Symposium*. INCOSE'2013. 11

[31] MOUSAVI, M. (2005). *Structuring Structural Operational Semantics*. Department of Computer Science, Eindhoven University of Technology. 33

[32] NIJSSEN, G. (2008). SBVR: N-ary fact types and subtypes- understandable and formal. *Business Rules Journal* **vol. 9,** URL (May 2013): http://www.BRCommunity.com/a2008/b412.html. 12, 13

[33] O'BRIEN, S. (2003). Controlling controlled english – an analysis of several controlled language rule sets. In *Proceedings of EAMT-CLAW 03*. Dublin City University. pp. 105–114. 20

[34] OMG (2013). Semantics Of Business Vocabulary And Rules (SBVR). *OMG Formally Released Versions Of SBVR*. URL (September 2013): http://www.omg.org/spec/SBVR/index.htm. 38

[35] PASTOR, O., GÓMEZ, J., INSFRÁN, E. AND PELECHANO, V. (2001). The OO-Method approach for information systems modeling: from object-oriented conceptual modeling to automated programming. *Inf. Syst.* **vol. 26,** pp. 507–534. 11

[36] PASTOR, O., INSFRÁN, E., PELECHANO, V., ROMERO, J. AND MERSEGUER, J. (1997). OO-Method: An OO software production environment combining conventional and formal methods. In *Proceedings of the 9th International Conference on Advanced Information Systems Engineering*. CAiSE '97. Springer-Verlag. pp. 145–158. 11

[37] PEETERS, F. (2011). Naar een valide objectmodel. *Internal project document* **EQuA,** pp. 1–16. 1, 3

[38] PÉREZ-GONZÁLEZ, H. G. AND KALITA, J. K. (2002). Automatically generating object models from natural language analysis. In *Companion of the 17th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. OOPSLA '02. ACM. pp. 86–87. 18

[39] PÉREZ-GONZÁLEZ, H. G., KALITA, J. K., NÚÑEZ VARELA, A. S. AND WIENER, R. S. (2005). GOOAL: an educational object oriented analysis laboratory. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. OOPSLA '05. ACM. pp. 180–181. 18

[40] PRASANNA, D. R. (2009). *Dependency Injection* 1st ed. Manning Publications Co. 41

[41] REENSKAUG, T. (2003). The Model-View-Controller (MVC) Its Past and Present. In *Proceedings of the JAOO Conference*. pp. 1–16. 2

[42] ROMBLEY, G. (2012). *Towards a valid object model*. CSE Master's thesis, W&I TU/e. 2

[43] ROSENBERG, D. AND STEPHENS, M. (2007). *Use Case Driven Object Modeling with UML Theory and Practice*. Books for professionals by professionals. Apress. v, 2, 15, 17

[44] ROSS, R. G. (2008). The emergence of SBVR and the true meaning of "semantics": Why you should care (a lot!), part 1. *Business Rules Journal* **vol. 9,** URL (May 2013): http://www.BRCommunity.com/a2008/b401.html. 12

[45] SCHMIDT, D. C. (2006). Guest Editor's Introduction: Model-Driven Engineering. *IEEE Computer* **39,** 25–31. 25

[46] SINNIG, D., MIZOUNI, R. AND KHENDEK, F. (2010). Bridging the gap: empowering use cases with task models. In *Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems*. EICS'10. ACM. pp. 291–296. 13

[47] STELLMAN, A. (2006). *Applied software project management*. O'Reilly. 7

[48] VAN HEE, K., SIDOROVA, N., SOMERS, L. AND VOORHOEVE, M. (2006). Consistency in model integration. *Data and Knowledge Engineering* **vol. 56,** pp. 4–22. 12

[49] VANBRABANT, R. (2008). *Google Guice: Agile Lightweight Dependency Injection Framework (Firstpress)*. APress. 41

[50] VONKEN, F., BRUNEKREEF, J., ZAIDMAN, A. AND PEETERS, F. (2012). Software Engineering in The Netherlands: The State of the Practice. *Software Engineering Research Group, TU Delft* **SERG,** pp. 1–44. 1

[51] YANG, D., SU, F. AND ZHOU, T. (2012). Applying robustness analysis to MDA software paradigm. In *Instrumentation Measurement, Sensor Network and Automation (IMSNA), 2012 International Symposium on*. Vol. 2. pp. 419–422. 3

[52] ZIKRA, I., STIRNA, J. AND ZDRAVKOVIC, J. (2011). Analyzing the integration between requirements and models in Model Driven Development. In *Enterprise, Business-Process and Information Systems Modeling*. Vol. 81 of *Lecture Notes in Business Information Processing*. Springer-Verlag. pp. 342–356. 3

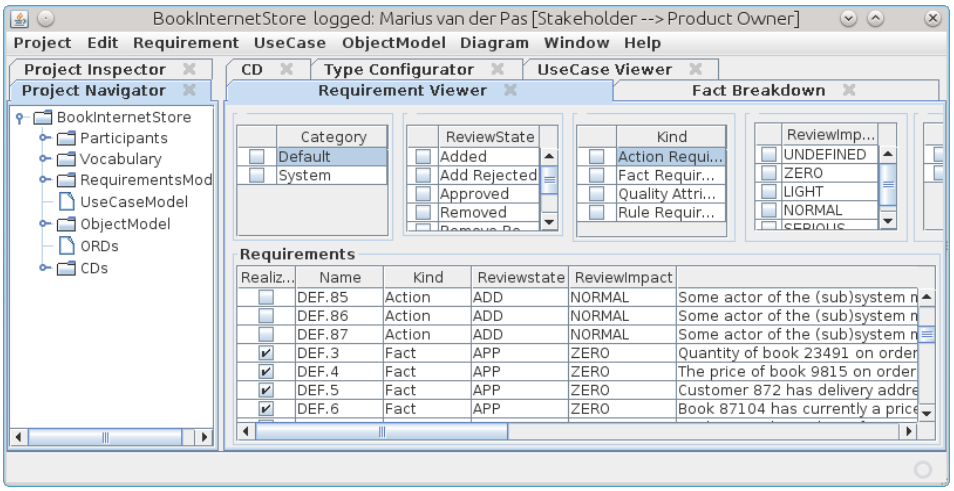
Appendix A

Symbiosis tool

The Symbiosis tool is a standalone application that implements the **EQuA framework**. The following sections include descriptions of the Symbiosis tool within the *BookInternetStore* example project.

A.1 General description

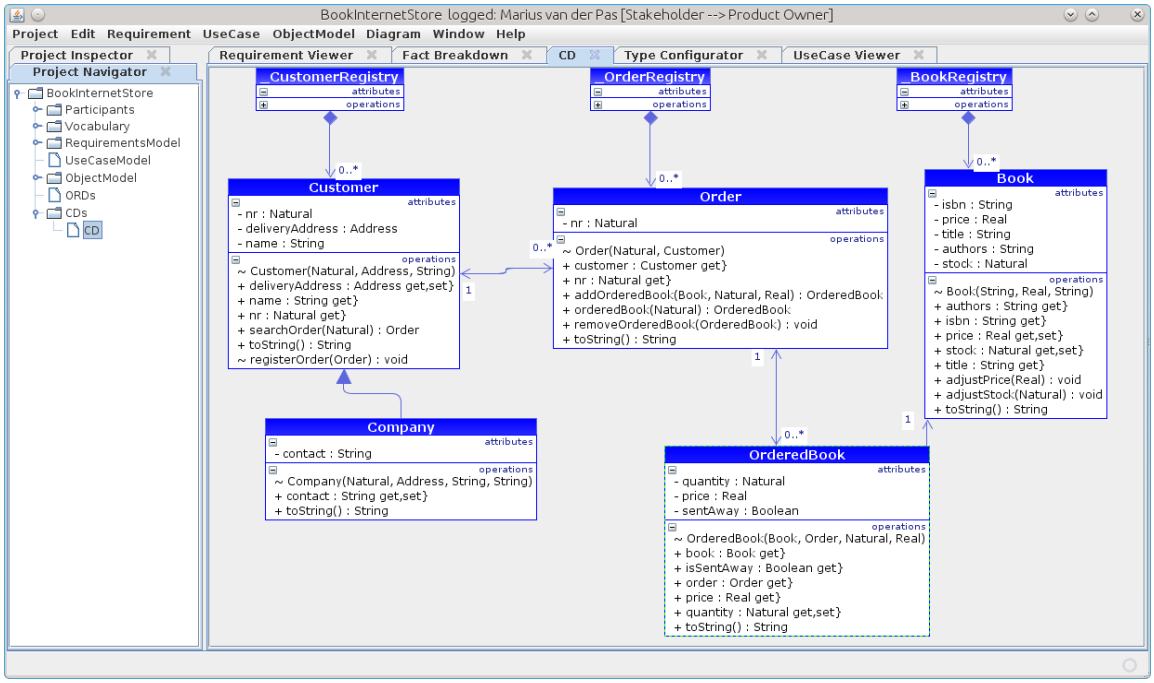
The **Requirements Viewer** tab (Figure A.1) displays a perspective of the Requirements Model. The transformation of requirements into objects is achieved in the **Fact Breakdown** tab. On the other hand, the **Type Configurator** tab shows a perspective of the Object Model and includes tools to adjust this model as needed. The *traceability* of requirements is achieved with the synchronization of the Requirements and Object Models. Symbiosis applies automatic updates to models according to the modifications submitted by users. The **UseCase Viewer** tab is a contribution of this thesis and shows a perspective of the Use Case Model. Finally, the **CD** tab contains the **UML class diagram**, which is a transformation of the Object Model.



The logged on user has the **project role** of 'product owner stakeholder'. A project role is composed by four attributes: the *type* (either *project member* or *stakeholder*), his or her *name*, the *role* (for instance, 'product owner') and *password*. In addition to the Natural Language contents of a requirement, the Requirements Model uses *metadata* to provide **categorization** of requirements, as well as **review states** and **prioritization** for the life-cycle of requirements. Moreover, the Requirements Model defines four **kinds** of requirement elements, which are discussed in Section 2.1.

Figure A.1: The Requirements Viewer tab

The class diagram is automatically generated by Symbiosis upon user request. Figure A.2 depicts the diagram from the *BookInternetStore*. The class diagram shows the objects according to their relations, which depend on the *facts* from the Requirements Model. These *facts* are semi-automatically analysed with a *conceptual modeling* approach that is based on the *Object-Role Modeling* (ORM). The resulting relations are UML standard: association, composition or generalization. Further discussion on the Requirements and Object Models and the ORM approach is available in Sections 2.1 and 2.2.



Symbiosis automatically generates the operations of **static-behavior** of objects that are based on facts, such as the constructors or setters/getters. Furthermore, the relations between objects provide information to generate operations that confirm the relations. For instance, the object *Order* sustains its relation with the object *Customer* via three operations; similarly *Customer* reports two operations with *Order*.

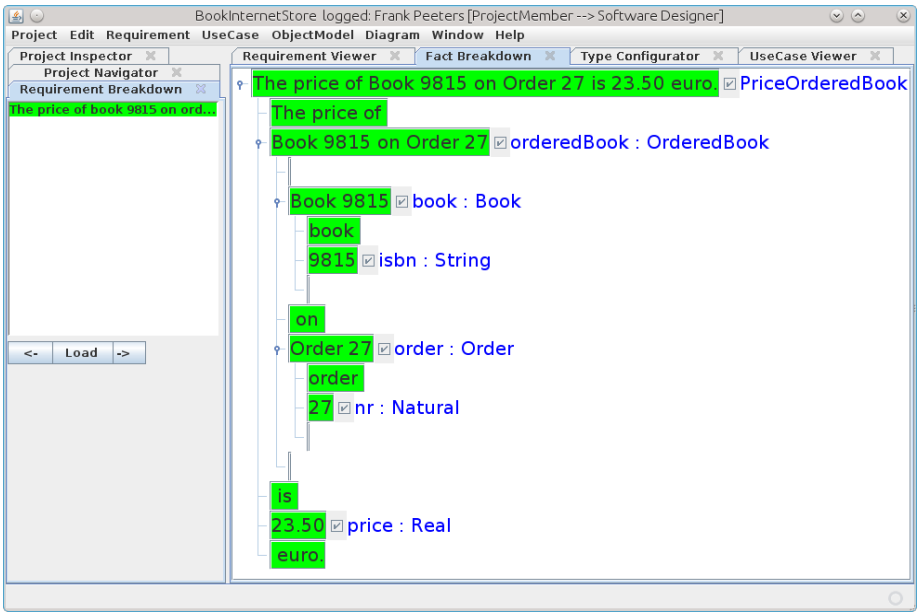
Figure A.2: Class diagram of the *BookInternetStore* example project

The Symbiosis tool requires the Java Platform Standard Edition 7 (JRE 7[1]) for its execution.

[1] http://docs.oracle.com/javase/7/docs/webnotes/install/index.html

A.2 Fact breakdown

From the *BookInternetStore* example project, the *fact* requirement *"The price of Book 9815 on order 27 is 23.50 euro."* becomes transformed into a *fact-type*, a formal element of the Object model. The **breakdown** transformation is **not deterministic**, as it is user-driven (i.e., semi-automatic). Figure A.3 shows the visualization of this breakdown in the **Fact Breakdown** tab of Symbiosis.

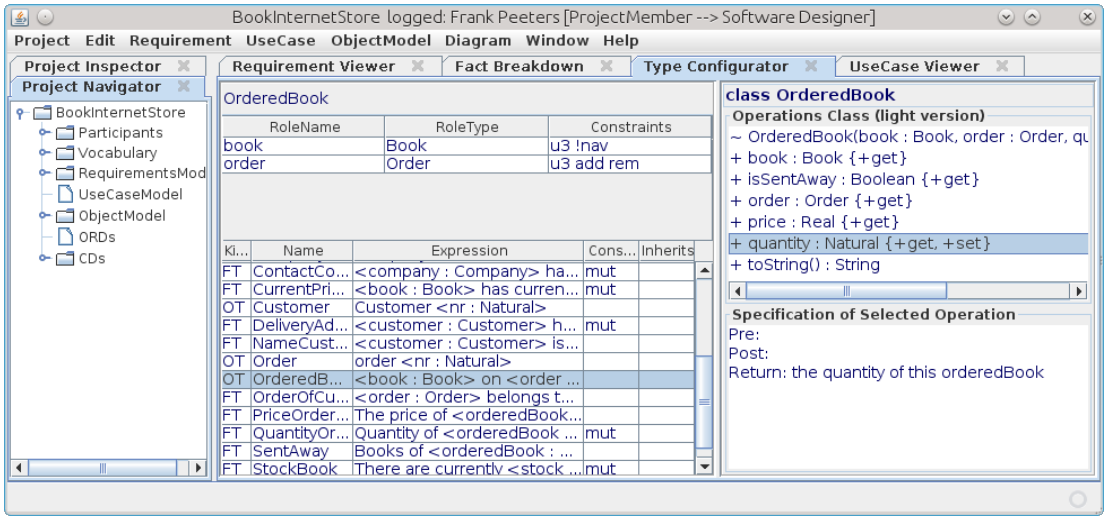


The *project member* analyses the requirement *"The price of Book 9815 on order 27 is 23.50 euro."* and its resulting *fact-type* is `PriceOrderedBook`, which represents a **relationship** between two **roles**: the 'price' and the 'orderedBook'. The 'price' role is played by a `Real` number object, which is a *base-type* element. The 'orderedBook' role is played by an `OrderedBook` object, which is a *fact-type* element. Recursively, these two roles are analysed by the *project member* in order to apply new breakdowns on them. No breakdown is possible in `Real`, because a *base-type* is a **breakdown terminal**. On the other hand, a breakdown is feasible for `OrderedBook`, as it contains more roles. This second breakdown produces two roles, which are the 'book' and 'order'. The players of these roles are *fact-types*, namely, `Book` and `Order`. Repeatedly, the analysis of breakdown is done on these two *fact-types* and only new *base-types* are obtained. Therefore, no more breakdown is achievable.
As an important remark, the *fact-types* `OrderedBook`, `Book` and `Order` are actually *object-types* as they do not represent a relationship between their roles.

Figure A.3: The Fact Breakdown tab

A.3 Type Configurator

Symbiosis offers a **type level** perspective of the Object Model with the **Type Configurator** tab. Furthermore, this tab allows the user-driven (i.e., semi-automatic) **configuration** of the Object Model. This configuration generates formal **constraints**, such as the uniqueness, additivity or removability of *roles* in *fact-types*. Figure A.4 displays an Object Model example.

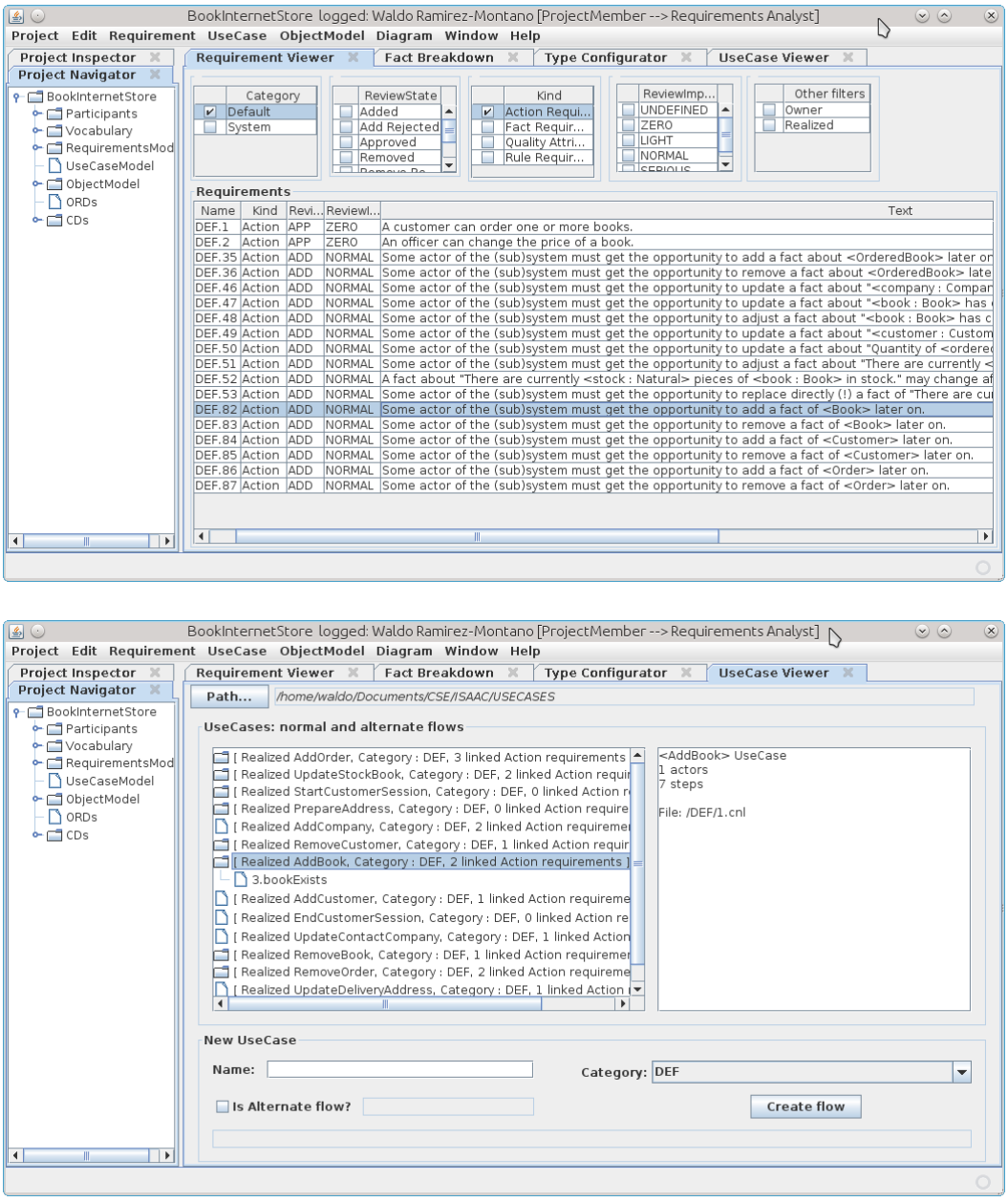


The sets of *fact-types* (FT) or *object-types* (OT) are presented in the lower matrix. The selected FT is `OrderedBook` and its set of *roles* is displayed in the upper matrix. Both matrices include the *constraint codes* of the corresponding FT, OT or *base-type*. The right hand side of this tab shows a class perspective of `OrderedBook`. This perspective includes some *behavioral-features*, such as the constructor and properties in the upper text panel. The lower text panel provides additional information about the selected *behavioral-feature*, such as the 'quantity' property in this example.

Figure A.4: The Type Configurator tab

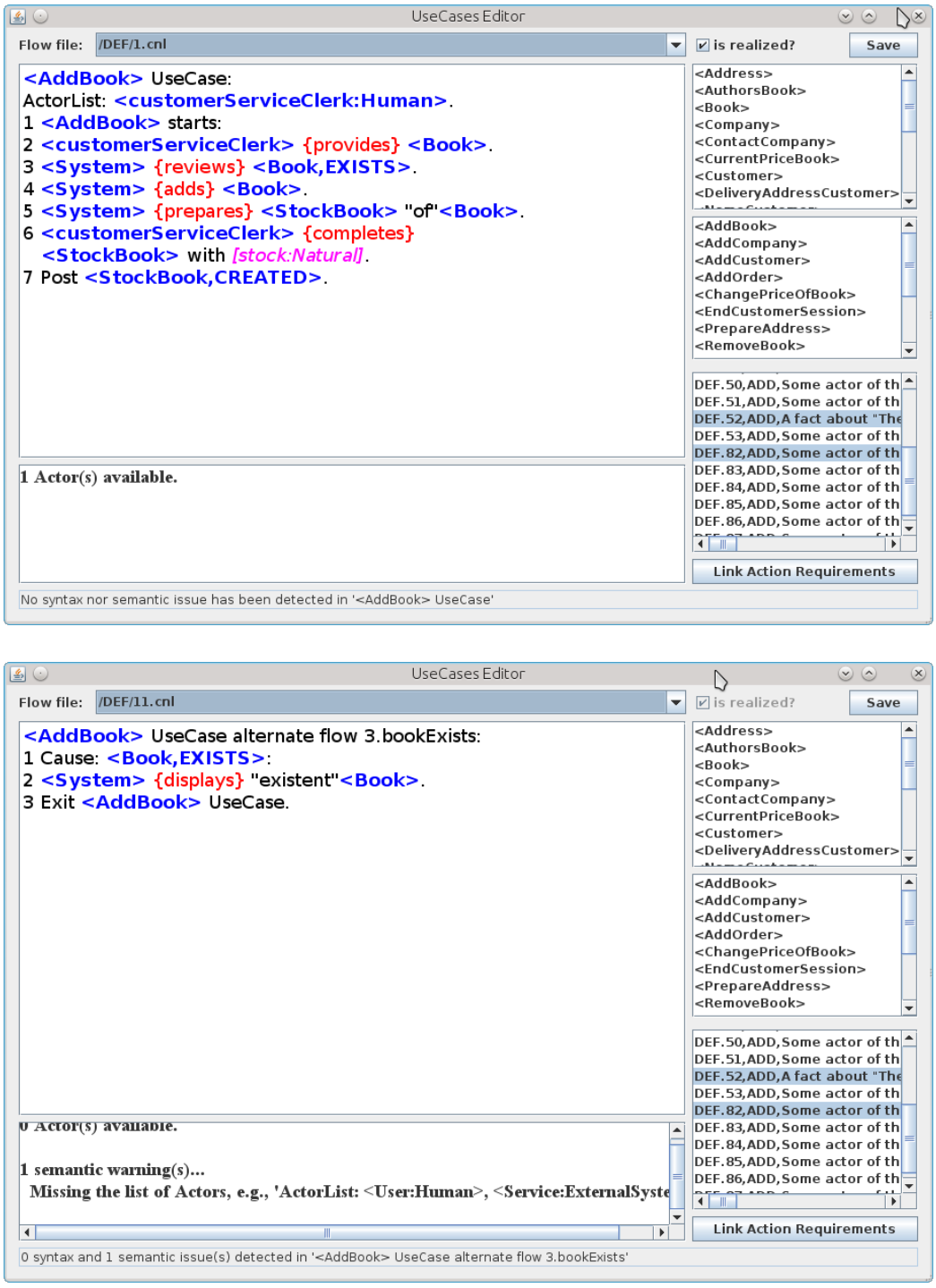
A.4 Categories for organizing use cases

The Requirements Model organizes the requirements with **categories**. A category is defined and owner by a project member who has the responsibility of its administration. The use cases utilize the categories for organizational purposes. Figure A.5 highlights two elements, an *action* requirement and a use case, which belong to the same category. Figure A.6 strengthens the link between these two elements.



The top image shows the **Requirement Viewer** tab that highlights an *action* requirement which refers to the *fact-type* `<Book>` in the category 'DEF'. The bottom image shows the **UseCase Viewer** tab that highlights a use case that belongs to the same category and that is linked with 2 *action* requirements.

Figure A.5: Category example – Requirement and UseCase Viewer



The top image shows the normal flow of the highlighted use case in Figure A.5. The **UseCases Editor** only allows to **link** use cases with *action* requirements of the same category. The highlighted *action* requirement in Figure A.5 is one of the 2 linked *actions*. The bottom image shows the alternate flow of the use case.

Figure A.6: Category example – UseCases Editor

Appendix B

Selected formalisms of the Object model

B.1 Lemmas

1. $n-1$ rule.
   If a fact type, with size of $n > 0$, is elementary then the smallest uniqueness constraint includes at least $n-1$ roles.
2. Unification of fact and object types.
   Every object type is a fact type.
3. Redundant uniqueness constraints.
   Assume a fact type with a set of roles R; consider a uniqueness constraint uc_1 which bounds the set of roles as $R_1 \subset R$ and a uniqueness constraint uc_2 which bounds the set of roles as $R_2 \subset R$ then:
   $R_1 \subset R_2 \Rightarrow uc_2$ is redundant.
4. No fact types without uniqueness.
   An elementary object model does not contain a fact type, with size > 0, without an uc.

B.2 Elementary Object Model

1. Elementary Fact Type.
   A fact type is elementary if it satisfies the $n-1$ rule.
2. Elementary Object Type.
   An object type is elementary if the corresponding fact type is elementary and every role is subject to an uniqueness constraint.
3. An elementary Object Model.
   An elementary object model contains only elementary facts and object types.

B.3 Standard Types

1. $\mathbb{T} \stackrel{\text{def}}{=}$ *the set of (simple and compound) types within an object model*
2. $\mathbb{OT} \stackrel{\text{def}}{=} \{t \mid t \in \mathbb{T} \wedge t \text{ is an object type}\}$
3. $\mathbb{FT} \stackrel{\text{def}}{=} \{t \mid t \in \mathbb{T} \wedge t \text{ is a fact type} \wedge t \notin \mathbb{OT}\}$
4. $\mathbb{F} \stackrel{\text{def}}{=} \mathbb{FT} \cup \mathbb{OT}$
5. $\mathbb{BT} \stackrel{\text{def}}{=} \{\text{String, Character, Natural, Integer, Real, Boolean}\}$
6. $\mathbb{ST} \stackrel{\text{def}}{=} \mathbb{BT} \cup \mathbb{OT}$

B.4 Role related utilities

1. $\mathbb{RL} \stackrel{\text{def}}{=} \{r \mid \exists f \in \mathbb{F} : r \text{ is a role of } f\}$
2. $type \stackrel{\text{def}}{=} \mathbb{RL} \rightarrow \mathbb{ST} : r \mapsto$ *the substitution type used in* r
3. $name \stackrel{\text{def}}{=} \mathbb{RL} \rightarrow \text{String} : r \mapsto$ *role name of* r *except if role name is unknown or empty then the name of* $type(r)$
4. $roles \stackrel{\text{def}}{=} \mathbb{F} \rightarrow \mathscr{P}(\mathbb{RL}) : f \mapsto \{r \mid r \text{ belongs to } f\}$
5. $parent \stackrel{\text{def}}{=} \mathbb{RL} \rightarrow \mathbb{F} : r \mapsto (f \mid f \in \mathbb{F} \wedge r \in roles(f))$
6. $size \stackrel{\text{def}}{=} \mathbb{F} \rightarrow \text{Natural} : f \mapsto |roles(f)|$
7. $nav \stackrel{\text{def}}{=} \mathbb{RL} \rightarrow \text{Boolean} : r \mapsto r$ *is a navigable role*
8. $rolesPlayedBy \stackrel{\text{def}}{=} \mathbb{OT} \rightarrow \mathscr{P}(\mathbb{RL}) : ot \mapsto \{r \mid r \in \mathbb{RL} \wedge type(r) = ot \wedge nav(r)\}$

Appendix C

CNL Design

C.1 CNL Extended-BNF

The non-terminal symbols V are the set of left-hand side symbols of the terminal and production rules R in Tables C.2 and C.3.

Key-symbols = { ; : . , < > [ ] { } " }

Keywords = { 'NonActorList:', 'ActorList:', 'Cause:', 'Continue', 'Post', 'Exit', 'for', 'each', 'every', 'if', 'calls:', 'expects:','UseCase' }

Sets of concrete terminal symbols of the alphabet that are utilized in the production rules.

Table C.1: Key-symbols and Keywords

BASETYPEKIND ::= 'Natural' | 'Integer' | 'Real' | 'Character' | 'Boolean' | 'String'
ACTORKIND ::= 'Human' | 'ExternalSystem'
FACTPREDICATEKIND ::= 'CREATED' | 'UPDATED' | 'REMOVED' | 'EXISTS' | 'EXCEPTION'
WORD ::= ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*
INT ::= ('0'..'9')+
STRING ::= '"' ('.'|','|'a'..'z'|'A'..'Z'|'0'..'9'|' ')* '"'

BASETYPEKIND is equivalent to the type of breakdown terminals specified in the Object Model. ACTORKIND follows the ICONIX guidelines for use cases kind of actors. FACTPREDICATEKIND represents either the CRUD response of a *fact-type* or an unexpected response. The rest of these rules generate groups of terminal symbols to represent words, integer numbers or strings.

Table C.2: Terminal rules

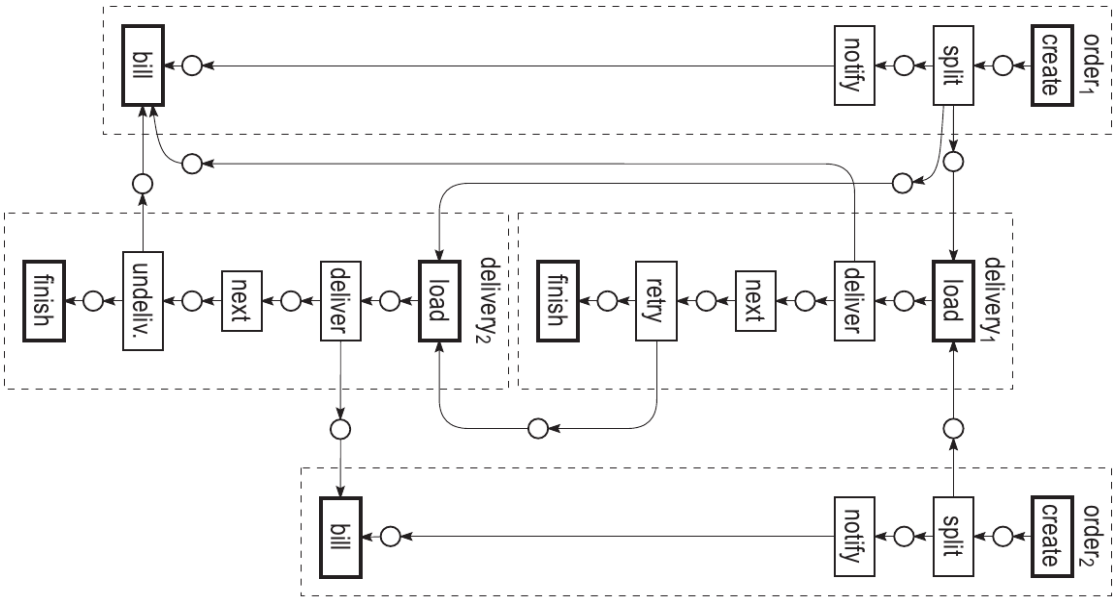
```
Flow ::= FlowType ':' (ActorList '.' )?  (NonActorList '.')?  ActionTypeList

NonActorList ::= 'NonActorList:' (','?  Noun)+
ActorList ::= 'ActorList:' (','?  Noun)+
ActionTypeList ::= TriggerType ':' ActionType* (EndType)?
Noun ::= '<' ((WORD ':' ACTORKIND )|WORD) '>'
BaseType ::= '[' WORD ':' BASETYPEKIND ']'
BaseTypeValue ::= '[' STRING ']'
Verb ::= '{' Keyphrase* WORD '}'
BaseAndValueExp ::= BaseType (Keyphrase+ BaseTypeValue)?
TriggerType ::= Rank (BeginExp | CauseExp)
BeginExp ::= NounPhraseExp WORD (Keyphrase+ NounPhraseExp+)?
CauseExp ::= 'Cause:' NounPhraseExp+
EndType ::= Rank EndExp '.'
EndExp ::= ( 'Continue' FlowType? WORD+ Rank) |
     ( 'Post' NounPhraseExp) | ( 'Exit' FlowType)
SubActionTypeList ::= ((ActionType+ EndType?)|EndType)
ActionType ::= (Rank NounPhraseExp VerbPhraseExp? '.' )  |
     (Rank (ForLoopExp | ConditionalExp) ':'
     SubActionTypeList Keyphrase+ Rank ';' )
ForLoopExp ::= 'for' (( 'each' NounPhraseExp Keyphrase+ NounPhraseExp) |
     ( 'every' NounPhraseExp))
ConditionalExp ::= 'if' NounPhraseExp
NounPhraseExp ::= STRING? '<' Noun ( ',' FACTPREDICATEKIND)? '>' BasePhraseExp?
BasePhraseExp ::= Keyphrase+ ( ',' ?  BaseAndValueExp )+
VerbPhraseExp ::= CallableVerbPhraseExp | InteractionExp
CallableVerbPhraseExp ::= (Verb NounPhraseExp+)|
     (Verb ( ',' ?  BaseAndValueExp)*)
RemoteActionExp ::= NounPhraseExp CallableVerbPhraseExp?
InteractionExp ::= InteractionScope RemoteActionExp
     (OptionPhrase InteractionScope RemoteActionExp)*
InteractionScope ::= 'calls:' | 'expects:'
OptionPhrase ::= Keyphrase+
Rank ::= INT ( '.' INT )*
FlowType ::=  '<' WORD '>' ((WORD WORD)|(WORD WORD Rank '.' WORD)|
     ( 'UseCase' )|( 'UseCase' WORD WORD Rank '.' WORD))
Keyphrase ::= INT | WORD
```

`Flow` is the start symbol of the CNL EBNF. `FlowType` is the non-terminal symbol that is designed to specify if a `Flow` is of either a use case or an object.

Table C.3: Production rules

C.2 Examples of Object life-cycle with CNL EBNF

The example of 'Artifact Choreography' [19] is used to expose an alternative to express object life- cycle with the CNL. The following figure shows a sample execution of the artifacts `Order` and `Delivery`, as well as a general description of the process.



A partially ordered run of the artifact choreography

"The shop splits each order into several packages based on the availability of the ordered items. Several packages from different orders are then delivered in one tour. In case a package cannot be delivered, it is scheduled for another delivery tour or returned to the shop as undeliverable. The order is billed to the customer once all packages are processed." (figure and text taken from [19]).

Figure C.1: `Order` and `Delivery` sample execution

The analogy of the Proclet model (see Figure 2.6) to the CNL EBNF is depicted on Figure C.1 with four example flows developed in the *XText CNL GUI*. This GUI is not designed to have communication with the Object Model, thus, these flows assume an Object Model that contains the *fact-types*. Within this assumption, the CNLV produces warning messages to recall that the validation of *fact-types* cannot be achieved. For instance,

corresponds to the warning message on each `Delivery` noun.

```
<Order> normal flow:
ActorList: <CustomerService:ExternalSystem>.
NonActorList: <Order>,<PackageList>,<Delivery>,<Package>,<OrderDelivery>.
1 <Order> starts:
2 <Order> {sets} <PackageList>.
3 <Order> {finds} <Delivery> with attribute [IsWaiting:Boolean] as ["true"].
4 for each <Package> of <PackageList>:
    4.1 <Order> calls: <Delivery> {loads} <Package>.
Completion of step 4;
5 <Order> {notifies} <CustomerService>.
6 <Order> expects: <Order> {update state} "of"<Delivery> "of"<Package>.
7 if <Order> has [IsCompleted:Boolean] as ["false"]:
    7.1 Continue in step 6.
End of step 7;
8 <Order> {bills} <CustomerService>.
9 Post <OrderDelivery,UPDATED>.
```

```
<Order> alternate flow 4.1.cannotLoadPackage:
NonActorList: <UnavailableDelivery>,<Order>,<Delivery>.
1 Cause: <UnavailableDelivery,EXCEPTION>:
2 <Order> {finds} <Delivery> with attribute [IsWaiting:Boolean] as ["true"].
3 Continue <Order> normal flow in step 4.1.
```

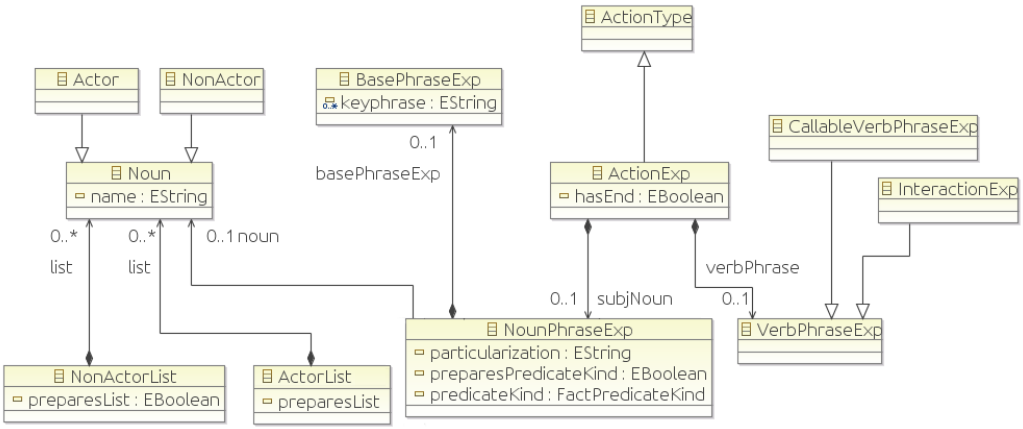
Figure C.2: `Order` life-cycle

```
<Delivery> normal flow:
NonActorList: <OrderDelivery>,<Delivery>,<Package>,<Order>.
1 <Delivery> begins:
2 <Delivery> sets attribute [MorePackagesFit:Boolean] as ["true"].
3 <Delivery> expects: <Delivery> {loads} <Package>
       or expects: <Delivery> {reloads} <Package>.
4 if <Delivery> has attribute [MorePackagesFit:Boolean] as ["true"]:
    4.1 Continue in step 3.
End of 4;
5 for each <Package> of <Delivery>:
    5.1 <Delivery> {distributes} <Package>.
    5.2 <Delivery> calls: <Order> {update state} "of"<Delivery> "of"<Package>.
End of 5;
6 Post <OrderDelivery,UPDATED>.
```

```
<Delivery> alternate flow 5.1.cannotDistributePackage:
ActorList: <CustomerService:ExternalSystem>.
NonActorList: <Package>,<Delivery>,<UndeliveredPackage>,<OrderDelivery>.
1 Cause: <UndeliveredPackage,EXCEPTION>:
2 if <Package> has attribute [State:String] as ["loaded"] :
    2.1 <Delivery> {finds} "Another"<Delivery>
        with attribute [IsWaiting:Boolean] as ["true"].
    2.2 <Delivery> calls: "Another"<Delivery> {reloads} <Package>.
    2.3 Continue <Delivery> normal flow in step 5.2.
End of step 2;
3 if <Package> has attribute [State:String] as ["reloaded"] :
    3.1 <Delivery> {notifies} <CustomerService>
        "about undeliverable"<Package>.
    3.2 Continue <Delivery> normal flow in step 5.
End of step 3;
```

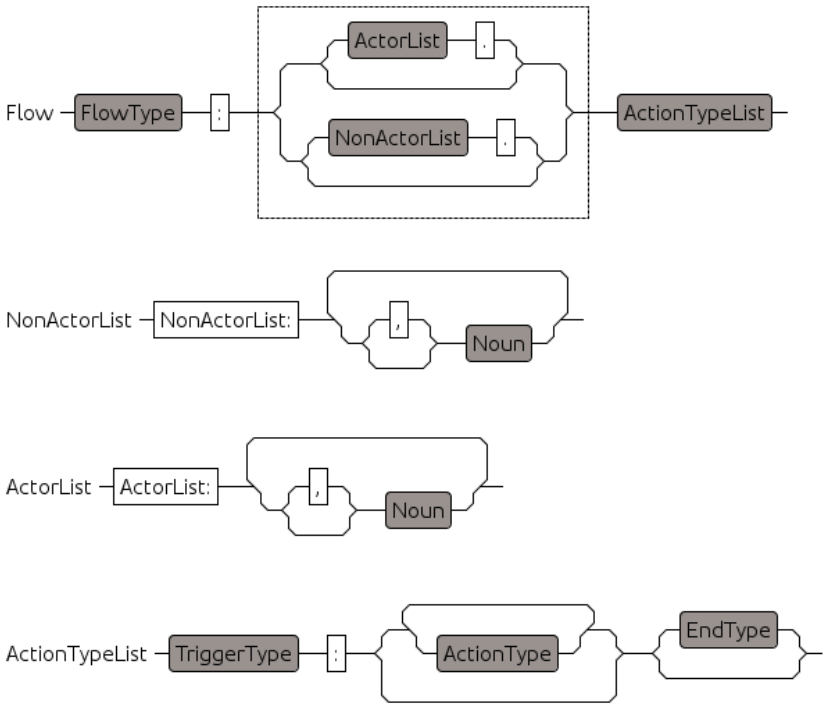
Figure C.3: `Delivery` life-cycle

C.3 CNL with MDE



Fragment of the EMF metamodel transformed into a class diagram that highlights the NP-VP pattern in the CNL.

Figure C.4: NP-VP pattern as a class diagram



Fragment of the Xtext CNL EBNF transformed into graphs. Each graph represents a rule. Along each graph, the shadowed elements are rules and the non-shadowed elements are key-symbols or keywords. The group of elements inside a dashed rectangle can have any sequential order in the textual input, for instance, `ActorList` can happen either before or after `NonActorList`.

Figure C.5: Syntax metamodel as a graph

C.4 Descriptions of Rule Delegates

The tables in this section describe the rule delegate validators χ_o. Some additional observations:

- → The utilization of active voice in sentences indicate that the subject noun is the direct executor of the verb. In this manner, the specification of `BasePhraseExp` (i.e., *base-types*) is not suggested in subject nouns. On the other hand the direct object nouns receive the execution of the verb, which suggests the specification of `BasePhraseExp` in them. Some χ_o return warning messages related to these suggestions.
- → The types returned by each Xtext CNL EBNF rule, as well as the types of features in these rules, can be inspected in Appendix D.1.
- → The reserved verbs and words are available in Appendix D.3.

RULE DELEGATE	DESCRIPTION
$\chi_{ActionType}$	The validation depends on the subtype of `ActionType`. The subtype is validated by calling its corresponding rule delegate: $\chi_{ActionExp}$, $\chi_{SubTriggerType}$, $\chi_{EndType}$ or $\chi_{TriggerType}$.
$\chi_{ActionExp}$	First, the existence of the subject noun in the actor or non-actor list is done with $\chi_{NounPhraseExp}$. Second, if the `VerbPhraseExp` is not present, the subject noun should not be a human actor or should not use CRUD predicate. Also, if this noun uses `BasePhrasExp` a warning is prepared, although its existence in the vocabulary is still validated. Third, if the `VerbPhraseExp` is present, its type defines the delegate to call: $\chi_{OperationExp}$, $\chi_{UnaryOperationExp}$ or $\chi_{InteractionExp}$. It is also checked that if the subject noun is a human actor and the flow is a use case flow, then the type of `VerbPhraseExp` should be `OperationExp`.
$\chi_{SubTriggerType}$	First, the call to either $\chi_{ForLoopExp}$ or $\chi_{ConditionalExp}$ is done according to the type of the attribute 'trigger'. Second, the validation of correct enumeration of the `SubActionTypeList` attribute (i.e., the *sub-action-types* of the for-loop or conditional expression) is done with the call to $\chi_{SubActionTypeList}$. Third, the keyphrase attribute is valid if it is not empty (i.e., its words are of free contents for the user). Fourth, the 'endSubTrigger' cross-reference should make reference to the `Rank` (i.e., step number) of the `SubTriggerType` (i.e., the validation of the closing of the for-loop or conditional expression). Finally, as keyphrase can contain numbers, the validation checks the presence of semicolon right after the endSubTrigger cross-reference.
$\chi_{ForLoopExp}$	This delegate is focused on object flows. The `ForLoopExp` can start with either 'for each' or 'for every'. In the first case, two `NounPhraseExp` and one keyphrase between them are validated. The existence of nouns in the non-actors list is validated with $\chi_{NounPhraseExp}$, afterwards it is checked that they do not have CRUD predicate. If any `NounPhraseExp` includes `BasePhraseExp` a warning message is prepared, although its existence in the vocabulary is validated. The keyphrase should end in one of the following reserved verbs: 'of', 'in' or 'on'. In the second case of `ForLoopExp`, the validation of one `NounPhraseExp` is done in the same manner as in the first case. General example 1: 'for each <noun> of <noun>' General example 2: 'for every <noun>' With further research on the Interaction Model, the validation of noun as *collection type* in its `NounPhraseExp` should be appropriately included.

Table C.4: Semantic rule delegates (I)

RULE DELEGATE	DESCRIPTION
$\chi_{ConditionalExp}$	This delegate is focused on object flows. The existence of the noun of the `NounPhraseExp` as a non actor is validated with $\chi_{NounPhraseExp}$. Next, the `NounPhraseExp` should have no CRUD predicate but should have `BasePhraseExp`. The keyphrase and `BaseAndValueExp` of the `BasePhraseExp` are validated. The keyphrase should begin with the reserved verb 'has/have' and end with the reserved word 'attribute'. The `BaseAndValueExp` should use the name of a *base-type* from the noun, its expected value and a keyword to specify the conditional expression. This keyphrase should end with the reserved word 'as'. General example: 'if <noun> has attribute [base-type] as [value]'
$\chi_{SubActionTypeList}$	This delegate also requires the `Rank` (i.e., step) of the `SubTriggerType` that starts the `SubActionTypeList` in order to validate their enumeration. The list of `ActionTypes` of the `SubActionTypeList` are iteratively validated with $\chi_{ActionType}$. With respect to the enumeration, the first `ActionType` should have the `Rank` $i.1$, where i is the `Rank` of the `SubTriggerType`, the point represents steps of 'sub-`ActionTypes`' and 1 represents the first sub-`ActionType`. Each `ActionType` that follows should have its `Rank` as $i.j+1$, where $i.j$ is the `Rank` of the prior `ActionType`. Furthermore, i and j should be positive integers.
$\chi_{InteractionExp}$	The `InteractionExp` should contain one or more `RemoteActionExps`. Their validation is done with one or more calls to the $\chi_{RemoteActionExp}$ delegate. In each call, the `InteractionScope` of the corresponding `RemoteActionExp` should be included. In the case of more than one `RemoteActionExps`, each `OptionPhrase` is analysed as a keyphrase. This keyphrase should start with the reserved words 'or' or 'and' to define the kind of option between the `RemoteActionExps` that are linked by the `OptionPhrase`. In the scope of use cases, this rule is designed to represent the ICONIX relationships «invokes» and «precedes». In the scope of objects, further experimentation was initiated in this rule, as it could represent (parallel) interaction between *fact-types* (see Appendix C.2). Moreover, the preliminary prototype enables the configuration of this rule. For instance, if the subject noun of each `RemoteActionExp` should be the same or if the all `InteractionScope` should be the same.
$\chi_{RemoteActionExp}$	This rule also receives the `InteractionScope`, which represents the verb of the sentence in which the subject noun is in the `InteractionExp` that calls this delegate. Thus, the `NounPhraseExp` of `RemoteActionExp` is the direct-object noun. First, the $\chi_{NounPhraseExp}$ is called to validate that the direct-object noun is not a human actor. Moreover, this noun should not use CRUD predicate nor `BasePhraseExp`. If the `CallableVerbPhraseExp` is not present, then the `RemoteActionExp` obtains the state that is used to represent the «invokes» and «precedes» use case relationships. That is, if `InteractionScope` is 'calls:' the direct-object noun would be the use case to be invoked. If `InteractionScope` is 'expects:' the direct-object noun would be the use case that precedes to the subject noun. On the other hand, if the `CallableVerbPhraseExp` is present, its validation is done with the delegate that validates the corresponding subtype, that is, with either $\chi_{OperationExp}$ or $\chi_{UnaryOperationExp}$.

Table C.5: Semantic rule delegates (II)

RULE DELEGATE	DESCRIPTION
$\chi_{OperationExp}$	This delegate utilizes χ_{Verb} to validate the operation (i.e., verb), as well as $\chi_{NounPhraseExp}$ for each `NounPhraseExp`. The nouns of these `NounPhraseExp` are the direct object nouns, so they should not be human actors. The CRUD predicate is allowed for *fact-types* but not for external system actors. If these direct object nouns utilize `BasePhraseExp`, the keyphrase in this expression should start/end with the reserved words 'with'/'attribute', otherwise a warning is prepared. The specification of value for the base-type in the `BasePhraseExp` is optional. If the value is included, it should match the type of the base-type and the keyphrase that joins the *base-type* with the value could use the reserved word 'as'. If the keyphrase does not include it, a warning is prepared.
$\chi_{UnaryOperationExp}$	This delegate reviews if one or more *base-types* are specified. If they do, the `UnaryOperationExp` is interpreted as a setter operation. The χ_{Verb} is utilized with the indication that the verb should be the reserved verb 'set' and the value for each *base-type* should be specified. The keyphrase between each *base-type* and its value should include the reserved word 'as', otherwise a warning is prepared. If no *base-type* is specified, the delegate χ_{Verb} is utilized without special indications.
$\chi_{TriggerType}$	`TriggerType` is an extension of `ActionType`, but `ActionType` is also abstract. Thus, the type of its attribute 'expression' determines if this delegate redirects the validation to $\chi_{BeginExp}$ or $\chi_{CauseExp}$.
$\chi_{CauseExp}$	This delegate validates that the noun of each `NounPhraseExp` is a non actor with $\chi_{NounPhraseExp}$. Afterwards, each `NounPhraseExp` should specify a CRUD predicate or a `BasePhraseExp` but not both. In the scope of use cases, the predicate 'EXCEPTION' is proposed to represent the cause of alternate flows that resolve unexpected executions of normal flows. In the scope of object flows, the predicate 'EXCEPTION' is proposed as the analogy to the flow that catches an exception of behavior of an object.
$\chi_{BeginExp}$	The first `NounPhraseExp` is the subject noun and is validated as a non actor with $\chi_{NounPhraseExp}$. Afterwards, the `NounPhraseExp` should not specify CRUD predicate nor `BasePhraseExp`. The 'verb' attribute of `BeginExp` should be one of the following reserved verbs: 'begin', 'starts', 'extends', 'precede'. If no more `NounPhraseExp` are included, the validation is completed. If more `NounPhraseExp`s are included, they are considered direct object nouns. Thus, the verb and direct object nouns should be linked with a keyphrase that should end with the reserved word 'with'. A warning is prepared if this keyword is not utilized. This delegate validates with $\chi_{NounPhraseExp}$ that each direct object noun is a non actor. Additionally, these direct object nouns should not specify CRUD predicate nor `BasePhraseExp`. In the scope of object flows, this expression is proposed as an analogy to a constructor, where the direct object nouns would represent its parameters.
$\chi_{EndType}$	`EndType` is the extension of `ActionType` that contains `EndExp`, which is an abstract *EObject*. Thus, the subtype of `EndExp` defines which delegate continues with the validation, that is, either $\chi_{GoToExp}$, $\chi_{PostExp}$ or $\chi_{EndUseCaseExp}$.

Table C.6: Semantic rule delegates (III)

RULE DELEGATE	DESCRIPTION
$\chi_{EndUseCaseExp}$	This delegate validates that the cross-reference 'particularFlow' corresponds to a use case flow.
$\chi_{PostExp}$	The `NounPhraseExp` of the `PostExp` is validated be a non actor via $\chi_{NounPhraseExp}$. In addition, the `NounPhraseExp` should specify its CRUD predicate and should not contain `BasePhraseExp`.
$\chi_{GoToExp}$	This delegate validates the correspondence of the cross-reference 'particularFlow' to a flow, as well as that the keyphrase should end with the reserved word 'step'. In addition, $\chi_{GoToExp}$ confirms the existence of the `Rank` (i.e., step) that is specified where to go. Furthermore, neither 'particularFlow' nor the `Rank` should refer to itself.
χ_{Rank}	This validator reviews that the enumeration of two `Ranks` (i.e., steps) is valid. As any `Rank` belongs to an `ActionType`, this validator is called by the delegate $\chi_{SubActionTypeList}$ and the root rule $\xi_{ActionTypeList}$.
$\chi_{NounPhraseExp}$	This delegate validates the existence of `NounPhraseExp` in the `NonActorList` and/or `ActorList`. The validation depends on the type of search that is specified to this delegate. For instance, the $\chi_{PostExp}$ calls $\chi_{NounPhraseExp}$ specifying that the noun should be a non actor (i.e., in the `NonActorList`).
χ_{Verb}	This delegate validates that the `Verb` matches reserved verbs or regular expressions. As future work, this validator should include the matching of verbs with operations according to the *fact-types* in the Interaction Model. In addition, a possible extension of the CNL Dictionary could offer a CRUD classification of verbs, which would be utilized by χ_{Verb} to extend the validation of verbs in use cases.

Table C.7: Semantic rule delegates (IV)

Appendix D

Prototype

D.1 Xtext CNL EBNF

The Grammar Language of Xtext has been used to implement the Xtext CNL EBNF. The following notes are related to the construction of *EObjects* from this grammar:

- → The assignation of features is done with the symbols '=', '?=' and '+='. The first symbol assigns an *EDataType* attribute (e.g., *EInt*) or a class relationship. The second symbol assigns an *EBoolean* attribute. The third attribute assigns an *EList* attribute.
- → The generalization/abstraction is achieved directly (i.e., using the 'returns' keyword in the rule declaration or indirectly (i.e., with the "{RULE.reason}" syntax, where the reason might be unneeded).
- → The cross-reference is specified with the syntax "[RULE|FEATURE]", where the specification of a feature is optional. If it is not specified, Xtext assumes the feature 'name' in the RULE. If FEATURE is used, it should represent an *EString* attribute.
- → The data-types cannot have assignation of features (otherwise they become a rule).

```
/*
 * Grammar definition of the Controlled Natural Language (CNL)
 * for the Use Cases and Interaction models. The behavior
 * is perceived as flows of actions.
 * The workspace with xtext consists of 4 Eclipse projects:
 * - equa.project.behavior Core project with the grammar definition
 * - equa.project.behavior.tests JUnit
 * - equa.project.behavior.ui Eclipse editor (i.e., GUI)
 * - equa.project.behavior.sdk Feature that links the three previous projects
 *
 * @author: waldo ramirez-montano
 */
grammar equa.project.behavior.CNL with org.eclipse.xtext.common.Terminals

generate cNL "http://www.project.equa/behavior/CNL"

// ---------------------- MODEL RULE ----------------------
Flow:
name=FlowType hasName?=':' (
(actorList=ActorList hasActorList?='.')?
& (nonActorList=NonActorList hasNonActorList?='.')?
)
flow=ActionTypeList;
```

```
// ---------------------- BEHAVIOR RULES ----------------------
NonActorList:
preparesList?='NonActorList:' (','? list+=Noun)+;
ActorList:
preparesList?='ActorList:' (','? list+=Noun)+;
ActionTypeList:
firstAction=TriggerType hasFirstAction?=':'
(actionList+=ActionType)*
(=> lastAction=EndType)?;
Noun:
'<'( ({Actor} name=ID preparesKind?=':' kind=ActorKind) |
({NonActor} name=ID) ) '>';
BaseType:
'[' name=ID is=':' type=BaseTypeKind ']';
BaseTypeValue:
'[' name=STRING ']';
Verb:
'{' keyphrase+=Keyphrase* name=ID '}';
BaseAndValueExp:
baseType=BaseType ( keyphrase+=Keyphrase+ value=BaseTypeValue)?;

// ---------------------- TRIGGER RULES ----------------------
TriggerType returns ActionType:
{TriggerType} name=Rank (
expression=BeginExp |
expression=CauseExp
);
BeginExp:
subjNoun=NounPhraseExp verb=ID
( keyphrase+=Keyphrase+ objNounList+=NounPhraseExp+ )?;
CauseExp:
hasCause?='Cause:' (objNounList+=NounPhraseExp)+;

// ---------------------- END RULES ----------------------
EndType returns ActionType:
{EndType} name=Rank expression=EndExp hasEnd?='.';
EndExp:
( name='Continue' {GoToExp.expression=current} (particularFlow=[Flow|FlowType])?
keyphrase+=ID+ actionTypeReference=Rank ) |
( name='Post' {PostExp.expression=current} objNoun=NounPhraseExp ) |
( name='Exit' {EndUseCaseExp.expression=current} particularUseCase=[Flow|FlowType] );

// ---------------------- ACTION RULES ----------------------
SubActionTypeList:
( (actionList+=ActionType+ (=> lastAction=EndType)?) | lastAction=EndType );
ActionType:
({ActionExp} name=Rank subjNoun=NounPhraseExp verbPhrase=VerbPhraseExp? hasEnd?='.')
| ({SubTriggerType} name=Rank ( trigger=ForLoopExp | trigger=ConditionalExp )
preparesSubActionList?=':' subActionList=SubActionTypeList
keyphrase+=Keyphrase+ endSubTrigger=[SubTriggerType|Rank]  hasEnd?=';');

// ---------------------- CONDITIONAL AND LOOP RULES ----------------------
ForLoopExp:
hasFor?='for' (
(hasEach?='each' element=NounPhraseExp keyphrase+=Keyphrase+ collection=NounPhraseExp)
| (hasEvery?='every' collection=NounPhraseExp )
);
ConditionalExp:
```

```
hasIf?='if' element=NounPhraseExp;

// ---------------------- NATURAL LANGUAGE RULES ----------------------
NounPhraseExp:
particularization=STRING? '<'noun=[Noun] (preparesPredicateKind?=','
predicateKind=FactPredicateKind)?'>' basePhraseExp=BasePhraseExp?;
BasePhraseExp:
keyphrase+=Keyphrase+ ( ','? baseValueList+=BaseAndValueExp )+;
VerbPhraseExp:
CallableVerbPhraseExp | InteractionExp;
CallableVerbPhraseExp:
({OperationExp} operation=Verb objNounList+=NounPhraseExp+) |
({UnaryOperationExp} operation=Verb (','? conditionList+=BaseAndValueExp)* );
RemoteActionExp:
nounPhrase=NounPhraseExp verbPhrase=CallableVerbPhraseExp?;

InteractionExp:
scope=InteractionScope expression=RemoteActionExp
( optionPhraseList+=OptionPhrase optionScopeList+=InteractionScope
optionExpressionList+=RemoteActionExp )*;
InteractionScope:
( calls?='calls:' | expects?='expects:' );
OptionPhrase:
keyphrase+=Keyphrase+;
// ---------------------- DATA TYPES ----------------------
Rank:
INT ( '.' INT )*;
FlowType:
'<' ID '>' (
(ID ID) |
(ID ID Rank '.' ID) |
('UseCase') |
('UseCase' ID ID Rank '.' ID)
);
Keyphrase:
INT | ID;

// ---------------------- ENUMS ----------------------
enum BaseTypeKind:
NATURAL='Natural' | INTEGER='Integer' | REAL='Real'
| CHARACTER='Character' | BOOLEAN='Boolean' | STRING='String';
enum ActorKind:
HUMAN='Human' | EXTERNAL_SYSTEM='ExternalSystem';
enum FactPredicateKind:
CREATED='CREATED' | UPDATED='UPDATED' | REMOVED='REMOVED'
| EXISTS='EXISTS' | EXCEPTION='EXCEPTION';
```

D.2 Xtext CNL API

The CNL contribution has been developed with the *Eclipse Modeling Tools IDE* Kepler Release under Linux Kubuntu 12.04, 64 bits. The resulting library for Symbiosis is the **CNLStandAloneValidator.jar** file. Table D.1 contains the description of this jar and the list of its API dependencies. The deduction of dependencies has been obtained by the Eclipse IDE. The configuration of dependencies is achieved by Xtext with Guice. The outline of the resulting dependency graph is available in Section D.4.

LIBRARY	DESCRIPTION
ANTLR 3.2 API	org.antlr.generator_3.2.0.v201108011202.jar, org.antlr.runtime_3.2.0.v201101311130.jar,
EMF 2.9 API	org.eclipse.emf.codegen.ecore_2.9.0.v20130610-0406.jar, org.eclipse.emf.codegen_2.9.0.v20130610-0406.jar, org.eclipse.emf.common_2.9.0.v20130528-0742.jar, org.eclipse.emf.ecore.xmi_2.9.0.v20130528-0742.jar, org.eclipse.emf.ecore_2.9.0.v20130528-0742.jar, org.eclipse.emf.mwe.core_1.2.1.v201306110341.jar, org.eclipse.emf.mwe.utils_1.3.0.v201306110341.jar, org.eclipse.emf.mwe2.language_2.4.0.v201306110940.jar, org.eclipse.emf.mwe2.launch_2.4.0.v201306110940.jar, org.eclipse.emf.mwe2.lib_2.4.0.v201306110341.jar, org.eclipse.emf.mwe2.runtime_2.4.0.v201306110341.jar
Guice 3.0 API	aopalliance.jar, com.google.guava_11.0.2.v201303041551.jar, com.google.inject_3.0.0.v201203062045.jar, guice-3.0.jar, guice-assistedinject-3.0.jar, guice-grapher-3.0.jar, guice-jmx-3.0.jar, guice-jndi-3.0.jar, guice-multibindings-3.0.jar, guice-persist-3.0.jar, guice-servlet-3.0.jar, guice-spring-3.0.jar, guice-struts2-plugin-3.0.jar, guice-throwingproviders-3.0.jar, javax.inject.jar, javax.inject_1.0.0.v20091030.jar.
Xtext CNL API	**CNLStandAloneValidator.jar** that contains: the *EObject* classes returned by the EBNF rules, the ANTLR parser, the `CNLStandAloneValidator` class and its utilities, the `CNLJavaValidator` class, the CNL Dictionary classes and the EMF extensions generated by Xtext.
Xtend 1.4 API	org.eclipse.xtend.lib_2.4.2.v201306120542.jar, org.eclipse.xtend.typesystem.emf_1.4.0.v201306110406.jar, org.eclipse.xtend_1.4.0.v201306110406.jar
Xtext 2.4.2 API	com.ibm.icu_50.1.1.v201304230130.jar, de.itemis.xtext.antlr_2.0.0.v201108011202.jar, org.apache.commons.cli_1.2.0.v201105210650.jar, org.apache.commons.logging_1.1.1.v201101211721.jar, org.apache.log4j_1.2.15.v201012070815.jar, org.eclipse.jdt.annotation_1.1.0.v20130513-1648.jar, org.eclipse.xpand_1.4.0.v201306110406.jar, org.eclipse.xtext.common.types_2.4.2.v201306120542.jar, org.eclipse.xtext.generator_2.4.2.v201306120542.jar, org.eclipse.xtext.util_2.4.2.v201306120542.jar, org.eclipse.xtext.xbase.lib_2.4.2.v201306120542.jar, org.eclipse.xtext.xbase_2.4.2.v201306120542.jar, org.eclipse.xtext_2.4.2.v201306120542.jar

Table D.1: CNL API for Symbiosis

The development of the prototype for Symbiosis has been done with NetBeans IDE 7.3.1 under the same operating system. The implementation of the prototype is discussed in Section 5.3. As a remark, the `UseCaseModel` class represents the new model member of Symbiosis and the new Swing GUI components are the `UseCaseViewer` and `UseCaseEditorDialog` classes. This prototype requires the APIs of Table D.1.

D.3 Isolated Vocabulary

Derived from the CNL grammar

LABEL	VALUE	LABEL	VALUE
KW_ACTOR_LIST_COLONS	ActorList:	KW_BOOLEAN	Boolean
KW_CALLS_COLONS	calls:	KW_CAUSE_COLONS	Cause:
KW_CHARACTER	Character	KW_CONTINUE	Continue
KW_CREATED	CREATED	KW_EACH	each
KW_EVERY	every	KW_EXCEPTION	EXCEPTION
KW_EXISTS	EXISTS	KW_EXIT	Exit
KW_EXTERNAL_SYSTEM	ExternalSystem	KW_EXPECTS_COLONS	expects:
KW_FOR	for	KW_HUMAN	Human
KW_IF	if	KW_INTEGER	Integer
KW_NATURAL	Natural	KW_NON_ACTOR_LIST_COLONS	NonActorList:
KW_POST	Post	KW_REAL	Real
KW_REMOVED	REMOVED	KW_STRING	String
KW_USECASE	UseCase	KW_UPDATED	UPDATED

Table D.2: Isolated Vocabulary: Key-words

Used in regular expressions or to define keyphrases

LABEL	VALUE	LABEL	VALUE
RW_ALTERNATE	alternate	RW_AND	and
RW_AS	as	RW_ATTRIBUTE	attribute
RW_CNL_FILE_EXTENSION	.cnl	RW_FALSE	false
RW_FLOW	flow	RW_FROM	from
RW_IN	in	RW_NORMAL	normal
RW_OF	of	RW_ON	on
RW_OR	or	RW_STEP	step
RW_SYSTEM	System	RW_TRUE	true
RW_WITH	with		

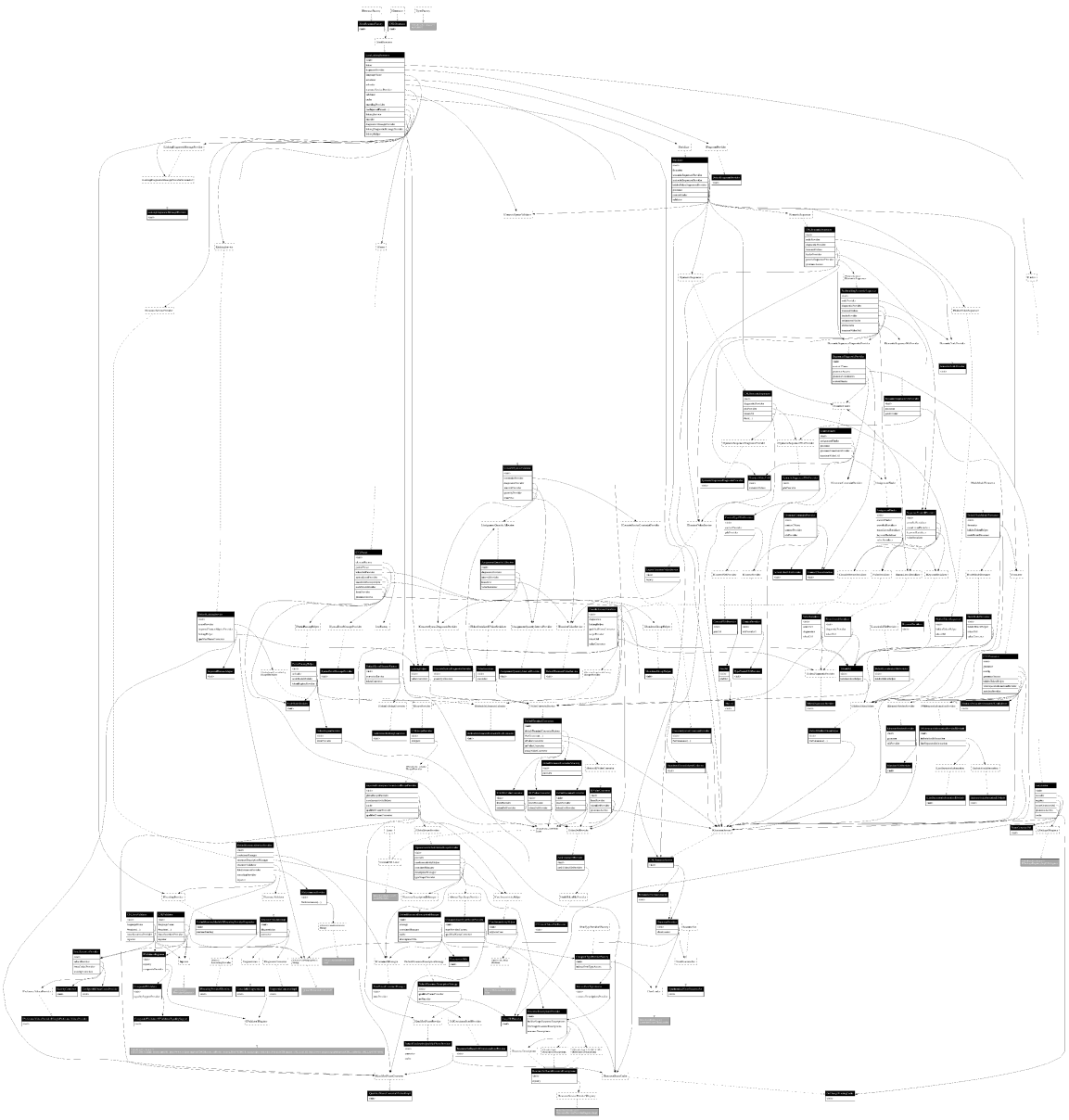
Table D.3: Isolated Vocabulary: Reserved words

'RV' refers to reserved verb

LABEL	VALUE
REGEX_ACTOR_BASE_TYPE	KW_HUMAN+"|"+KW_EXTERNAL_SYSTEM
REGEX_BOOLEAN_VALUE	RW_TRUE+"|"+RW_FALSE
REGEX_CHARACTER_VALUE	"."
REGEX_COLONS	":"
REGEX_FACT_TYPE_NAME	"\\w+"
REGEX_FACT_TYPE_PREDICATES	KW_CREATED+"|"+KW_UPDATED+"|"+KW_REMOVED+"|" +KW_EXISTS+"|"+KW_EXCEPTION
REGEX_FILE_OR_FOLDER_NAME	"\\w+(\\.\\w+)*"
REGEX_MEMBERSHIP	RW_OF+"|"+RW_IN+"|"+RW_ON+"|"+RW_FROM
REGEX_NATURAL_NUMBER	"0|[1-9]\\d*"
REGEX_INTEGER_NUMBER	"(\\+|-)?("+REGEX_NATURAL_NUMBER+")"
REGEX_OPTION_WORD	RW_OR+"|"+RW_AND
REGEX_FORWARD_SLASH	"/"
REGEX_PACKAGE_NAME	"\\w+("+REGEX_FORWARD_SLASH+"\\w+)*"
REGEX_RANK	"\\d+(\\.\\d+)*"
REGEX_LABEL	"\\.([a-zA-Z]+\\w*)"
REGEX_RANK_AND_LABEL	REGEX_RANK+REGEX_LABEL
REGEX_REAL_NUMBER	REGEX_INTEGER_NUMBER+"(\\.\\d+)?"
REGEX_WHITE_SPACES	"\\s+"
RV_BEGIN	"begin|begins|start|starts|extend|extends|precede|precedes"
RV_HAVE	"have|has"
RV_SET	"set|sets"

Table D.4: Isolated Vocabulary: Regular expressions

D.4 Outline of the dependency graph for the CNL



Graph created by the `Injector` of Guice according to the modules of configuration provided by Xtext.

Figure D.1: Outline of the dependency graph for the CNL

Appendix E

Preliminary case Study

E.1 Requirements Model of the BookInternetStore

Name	Kind	Reviewstate	ReviewImpact	Text
DEF.1	Action	APP	ZERO	A customer can order one or more books.
DEF.2	Action	APP	ZERO	An officer can change the price of a book.
DEF.35	Action	ADD	NORMAL	Some actor of the (sub)system must get the opportunity to add a fact about <OrderedBook> later on.
DEF.36	Action	ADD	NORMAL	Some actor of the (sub)system must get the opportunity to remove a fact about <OrderedBook> later on.
DEF.46	Action	ADD	NORMAL	Some actor of the (sub)system must get the opportunity to update a fact about "<company : Company> has contact with name <name : String>." later on.
DEF.47	Action	ADD	NORMAL	Some actor of the (sub)system must get the opportunity to update a fact about "<book : Book> has currently a price of <price : Real> euro." later on.
DEF.48	Action	ADD	NORMAL	Some actor of the (sub)system must get the opportunity to adjust a fact about "<book : Book> has currently a price of <price : Real> euro." later on.
DEF.49	Action	ADD	NORMAL	Some actor of the (sub)system must get the opportunity to update a fact about "<customer : Customer> has delivery address <deliveryAddress : Address>." later on.
DEF.50	Action	ADD	NORMAL	Some actor of the (sub)system must get the opportunity to update a fact about "Quantity of <orderedBook : OrderedBook> is <quantity : Natural>." later on.
DEF.51	Action	ADD	NORMAL	Some actor of the (sub)system must get the opportunity to adjust a fact about "There are currently <stock : Natural> pieces of <book : Book> in stock." later on.
DEF.52	Action	ADD	NORMAL	A fact about "There are currently <stock : Natural> pieces of <book : Book> in stock." may change after input of such a fact.
DEF.53	Action	ADD	NORMAL	Some actor of the (sub)system must get the opportunity to replace directly (!) a fact of "There are currently <stock : Natural> pieces of <book : Book> in stock.".
DEF.82	Action	ADD	NORMAL	Some actor of the (sub)system must get the opportunity to add a fact of <Book> later on.
DEF.83	Action	ADD	NORMAL	Some actor of the (sub)system must get the opportunity to remove a fact of <Book> later on.
DEF.84	Action	ADD	NORMAL	Some actor of the (sub)system must get the opportunity to add a fact of <Customer> later on.
DEF.85	Action	ADD	NORMAL	Some actor of the (sub)system must get the opportunity to remove a fact of <Customer> later on.
DEF.86	Action	ADD	NORMAL	Some actor of the (sub)system must get the opportunity to add a fact of <Order> later on.
DEF.87	Action	ADD	NORMAL	Some actor of the (sub)system must get the opportunity to remove a fact of <Order> later on.
DEF.3	Fact	APP	ZERO	Quantity of book 23491 on order 45 is 1.
DEF.4	Fact	APP	ZERO	The price of book 9815 on order 27 is 23.50 euro.
DEF.5	Fact	APP	ZERO	Customer 872 has delivery address Rachelsmolen 1, 5600BA Eindhoven, The Netherlands.
DEF.6	Fact	APP	ZERO	Book 87104 has currently a price of 22.50 euro.
DEF.7	Fact	APP	ZERO	Book 87104 has title "Software Development with Symbiosis".
DEF.8	Fact	APP	ZERO	Book 87104 is written by Frank Peeters and Waldo Ramirez.
DEF.9	Fact	APP	ZERO	Customer 3461 is called Ion Barosan.
DEF.10	Fact	APP	ZERO	Company 872 has contact with name Rutger Lippits.
DEF.11	Fact	APP	ZERO	Company 872 is called Fontys Hogeschool ICT.
DEF.12	Fact	APP	ZERO	There are currently 45 pieces of book 23491 in stock.
DEF.13	Fact	APP	ZERO	Books of book 23491 on order 45 are sent away for deliverance.
DEF.14	Fact	APP	ZERO	Order 45 belongs to Customer 78.

Table E.1: Fact and Action requirements of the BookInternetStore

Name	Kind	Reviewstate	ReviewImpact	Text
DEF.16	QA	APP	ZERO	Sign up by a customer is protected.
DEF.15	Rule	APP	ZERO	If the ordered quantity of a book is available, sending will be done immediately.
DEF.17	Rule	ADD	NORMAL	Two (or more) facts about <Order> with the same value on <nr : Natural> are not allowed.
DEF.18	Rule	ADD	NORMAL	Two (or more) facts about <Book> with the same value on <isbn : String> are not allowed.
DEF.19	Rule	ADD	NORMAL	Two (or more) facts about <OrderedBook> with the same combination of values on <book : Book,order : Order> are not allowed.
DEF.20	Rule	ADD	NORMAL	Two (or more) facts about "Quantity of <orderedBook : OrderedBook> is <quantity : Natural>." with the same value on <orderedBook : OrderedBook> are not allowed.
DEF.21	Rule	ADD	NORMAL	Every <OrderedBook> cannot exist without a fact about "Quantity of <orderedBook : OrderedBook> is <quantity : Natural>.", without any consideration.
DEF.22	Rule	ADD	NORMAL	Two (or more) facts about <Customer> with the same value on <nr : Natural> are not allowed.
DEF.23	Rule	ADD	NORMAL	Two (or more) facts about <Address> with the same combination of values on <street : String,nr : String,zip : String,city : String,country : String> are not allowed.
DEF.24	Rule	ADD	NORMAL	Two (or more) facts about <Company> with the same value on <nr : Natural> are not allowed.
DEF.25	Rule	APP	ZERO	Two (or more) facts about "Books of <orderedBook : OrderedBook> are sent away for deliverance." with the same value on orderedBook are not allowed.
DEF.26	Rule	ADD	NORMAL	Two (or more) facts about "<book : Book> is written by <authors : String>." with the same value on <book : Book> are not allowed.
DEF.27	Rule	ADD	NORMAL	Two (or more) facts about "<company : Company> has contact with name <name : String>." with the same value on <company : Company> are not allowed.
DEF.28	Rule	ADD	NORMAL	Every <Company> cannot exist without a fact about "<company : Company> has contact with name <name : String>.", without any consideration.
DEF.29	Rule	ADD	NORMAL	Two (or more) facts about "<book : Book> has currently a price of <price : Real> euro." with the same value on <book : Book> are not allowed.
DEF.30	Rule	ADD	NORMAL	Every <Book> cannot exist without a fact about "<book : Book> has currently a price of <price : Real> euro.", without any consideration.
DEF.31	Rule	ADD	NORMAL	Two (or more) facts about "<customer : Customer> has delivery address <deliveryAddress : Address>." with the same value on <customer : Customer> are not allowed.
DEF.32	Rule	ADD	NORMAL	Every <Customer> cannot exist without a fact about "<customer : Customer> has delivery address <deliveryAddress : Address>.", without any consideration.
DEF.33	Rule	ADD	NORMAL	Two (or more) facts about "<customer : Customer> is called <name : String>." with the same value on <customer : Customer> are not allowed.
DEF.34	Rule	ADD	NORMAL	Every <Customer> cannot exist without a fact about "<customer : Customer> is called <name : String>.", without any consideration.
DEF.37	Rule	ADD	NORMAL	Two (or more) facts about "<order : Order> belongs to <customer : Customer>." with the same value on <order : Order> are not allowed.
DEF.38	Rule	ADD	NORMAL	Every <Order> cannot exist without a fact about "<order : Order> belongs to <customer : Customer>.", without any consideration.
DEF.39	Rule	ADD	NORMAL	Two (or more) facts about "The price of <orderedBook : OrderedBook> is <price : Real> euro." with the same value on <orderedBook : OrderedBook> are not allowed.
DEF.40	Rule	ADD	NORMAL	Every <OrderedBook> cannot exist without a fact about "The price of <orderedBook : OrderedBook> is <price : Real> euro.", without any consideration.
DEF.42	Rule	ADD	NORMAL	Two (or more) facts about "There are currently <stock : Natural> pieces of <book : Book> in stock." with the same value on <book : Book> are not allowed.
DEF.43	Rule	ADD	NORMAL	Every <Book> cannot exist without a fact about "There are currently <stock : Natural> pieces of <book : Book> in stock.", without any consideration.
DEF.44	Rule	ADD	NORMAL	Two (or more) facts about "<book : Book> has title "<title : String>"." with the same value on <book : Book> are not allowed.
DEF.45	Rule	ADD	NORMAL	Every <Book> cannot exist without a fact about "<book : Book> has title "<title : String>".", without any consideration.
DEF.54	Rule	ADD	NORMAL	0 is the default value of <stock> in "There are currently <stock : Natural> pieces of <book : Book> in stock."
DEF.68	Rule	ADD	NORMAL	The value of nr in <Customer> is generated by auto increment
DEF.75	Rule	ADD	NORMAL	The value of nr in <Order> is generated by auto increment

Table E.2: Rule and Quality requirements of the BookInternetStore

E.2 Object Model of the BookInternetStore

Kind	Name	Expression	Constraints	Inherits
FT	_BookRegistration	<_BookRegistry> has registered <Book>.		
OT	_BookRegistry	BookRegistry		
FT	_CustomerRegistration	<_CustomerRegistry> has registered <Customer>.		
OT	_CustomerRegistry	CustomerRegistry		
FT	_OrderRegistration	<_OrderRegistry> has registered <Order>.		
OT	_OrderRegistry	OrderRegistry		
VT	Address	<street : String> <nr : String>, <zip : String> <city : String>, <country : String>		
FT	AuthorsBook	<book : Book> is written by <authors : String>.		
OT	Book	book <isbn : String>		
OT	Company	Company <nr : Natural>		Customer
FT	ContactCompany	<company : Company> has contact with name <contact : String>.	mut	
FT	CurrentPriceBook	<book : Book> has currently a price of <price : Real> euro.	mut	
OT	Customer	Customer <nr : Natural>		
FT	DeliveryAddressCust...	<customer : Customer> has delivery address <deliveryAddress : Address>.	mut	
FT	NameCustomer	<customer : Customer> is called <name : String>.		
OT	Order	order <nr : Natural>		
OT	OrderedBook	<book : Book> on <order : Order>		
FT	OrderOfCustomer	<order : Order> belongs to <customer : Customer>.		
FT	PriceOrderedBook	The price of <orderedBook : OrderedBook> is <price : Real> euro.		
FT	QuantityOrderedBook	Quantity of <orderedBook : OrderedBook> is <quantity : Natural>.	mut	
FT	SentAway	Books of <orderedBook : OrderedBook> are sent away for deliverance.		
FT	StockBook	There are currently <stock : Natural> pieces of <book : Book> in stock.	mut	
FT	TitleBook	<book : Book> has title "<title : String>".		

The class diagram perspective of this Object Model is available in Appendix A.1, Figure A.2.

Table E.3: Fact-types of the BookInternetStore

E.3 Use cases of the BookInternetStore

`<AddCustomer>` and `<AddCompany>` use cases

<AddCustomer> UseCase: ActorList: <client:Human>. 1 <AddCustomer> starts: 2 <AddCustomer> expects: <PrepareAddress>. 3 <System> {creates} <Customer>. 4 <System> {creates} <DeliveryAddressCustomer> "with given"<Address>. 5 <client> {provides} <NameCustomer>. 6 Post <Customer,CREATED>.	<AddCompany> UseCase: ActorList: <client:Human>. 1 <AddCompany> starts: 2 <AddCompany> expects: <PrepareAddress>. 3 <System> {creates} <Company>. 4 <System> {creates} <DeliveryAddressCustomer> "with given"<Address>. 5 <client> {provides} <ContactCompany>. 6 Post <Company,CREATED>.

`<PrepareAddress>` use case

<PrepareAddress> UseCase: ActorList: <client:Human>, <locatorService:ExternalSystem>. 1 <PrepareAddress> starts: 2 <client> {provides} <Address> with [street:String], [nr:String], [zip:String], [city:String], [country:String]. 3 <System> {checks} <Address,EXISTS> "with the"<locatorService>.	<PrepareAddress> UseCase alternate flow 3.invalidAddress: 1 Cause: <Address,EXCEPTION>: 2 <System> {notifies} "invalid"<Address>. 3 Continue <PrepareAddress> UseCase in step 2.
	<PrepareAddress> UseCase alternate flow 2.cancelPreparation: ActorList: <client:Human>. 1 Cause: <PrepareAddress,EXCEPTION>: 2 <client> {cancels} <PrepareAddress>. 3 Exit <PrepareAddress> UseCase.

`<UpdateContactCompany>` and `<UpdateDeliveryAddress>` use cases

<UpdateContactCompany> UseCase: ActorList: <client:Human>. 1 <UpdateContactCompany> starts: 2 <client> {updates} <ContactCompany> with new [contact:String]. 3 Post <ContactCompany,UPDATED>.	<UpdateDeliveryAddress> UseCase: ActorList: <client:Human>. 1 <UpdateDeliveryAddress> begins: 2 <UpdateDeliveryAddress> expects: <PrepareAddress>. 3 <System> {updates} <DeliveryAddressCustomer> "with given"<Address>. 4 Post <DeliveryAddressCustomer,UPDATED>.

`<AddOrder>` use case

<AddOrder> UseCase: ActorList: <client:Human>. 1 <AddOrder> starts: 2 <client> chooses <Book>. 3 <System> reviews <StockBook>. 4 if <StockBook> has [stock:Natural] as ["0"]: 4.1 <System> {notifies} "unavailable"<Book>. 4.2 Continue in step 2. End of step 4; 5 <client> {adds} <OrderedBook>. 6 <System> {creates} <OrderOfCustomer>. 7 <client> {updates} <QuantityOrderedBook>. 8 Post <Order,CREATED>.	<AddOrder> UseCase alternate flow 5.refusedPrice: ActorList: <client:Human>. 1 Cause: <CurrentPriceBook,EXCEPTION>: 2 <client> {refuses} <CurrentPriceBook>. 3 Continue <AddOrder> UseCase in step 2.
	<AddOrder> UseCase alternate flow 2.cancelAddOrder: ActorList: <client:Human>. 1 Cause: <AddOrder,EXCEPTION>: 2 <client> {cancels} <AddOrder>. 3 Exit <AddOrder> UseCase.

Table E.4: Use cases for the actor `<client>` (I)

<RemoveOrder> use case

<RemoveOrder> UseCase: ActorList: <client:Human>. 1 <RemoveOrder> starts: 2 <client> {selects} <OrderOfCustomer>. 3 <System> {returns} <Order>. 4 <client> {cancels} <Order>. 5 <System> {removes} <OrderedBook>. 6 Post <Order,REMOVED>.	<RemoveOrder> UseCase alternate flow 4.orderSent: 1 Cause: <SentAway,EXISTS>: 2 <System> {notifies} <SentAway>. 3 Exit <RemoveOrder> UseCase.

<StartCustomerSession> and EndCustomerSession use cases

<StartCustomerSession> UseCase: ActorList: <client:Human>. 1 <StartCustomerSession> starts: 2 <client> {provides} "credentials of"<Customer>. 3 <System> {acknowledges} <Customer>. 4 <StartCustomerSession> calls: <AddOrder> or calls: <RemoveOrder> or calls: <EndCustomerSession> or calls: <UpdateContactCompany> or calls: <UpdateDeliveryAddress>. 5 Continue in step 4.	<StartCustomerSession> UseCase alternate flow 3.invalidCustomer: 1 Cause: <Customer,EXCEPTION>: 2 <System> {replies} "invalid"<Customer>. 3 Exit <StartCustomerSession> UseCase.
	<EndCustomerSession> UseCase: ActorList: <client:Human>. 1 <EndCustomerSession> starts: 2 <client> {terminates} <StartCustomerSession>. 3 Exit <StartCustomerSession> UseCase.

Table E.5: Use cases for the actor <client> (II)

<ChangePriceOfBook> use case

<ChangePriceOfBook> UseCase: ActorList: <officer:Human>. 1 <ChangePriceOfBook> starts: 2 <officer> {provides} <Book> with [isbn:String]. 3 <System> {searches} <Book>. 4 <System> {displays} <CurrentPriceBook>. 5 <officer> {updates} <CurrentPriceBook> with [price:Real]. 6 Post <CurrentPriceBook,UPDATED>.	<ChangePriceOfBook> UseCase alternate flow 3.invalidBook: ActorList: <officer:Human>. 1 Cause: <Book,EXCEPTION>: 2 <System> {notifies} "nonexistent"<Book>. 3 Continue <ChangePriceOfBook> UseCase in step 2.
	<ChangePriceOfBook> UseCase alternate flow 2.changeCancellation: ActorList: <officer:Human>. 1 Cause: <ChangePriceOfBook,EXCEPTION>: 2 <officer> {cancels} <ChangePriceOfBook>. 3 Exit <ChangePriceOfBook> UseCase.

<UpdateStockBook> use case

<UpdateStockBook> UseCase: ActorList: <officer:Human>. 1 <UpdateStockBook> begins: 2 <officer> {provides} <Book> with [isbn:String]. 3 <System> {searches} <Book>. 4 <System> {displays} <StockBook>. 5 <officer> {updates} <StockBook> with [stock:Natural]. 6 Post <StockBook,UPDATED>.	<UpdateStockBook> UseCase alternate flow 3.invalidBook: ActorList: <officer:Human>. 1 Cause: <Book,EXCEPTION>: 2 <System> {notifies} "nonexistent"<Book>. 3 Continue <UpdateStockBook> UseCase in step 2.
	<UpdateStockBook> UseCase alternate flow 2.updateCancellation: ActorList: <officer:Human>. 1 Cause: <UpdateStockBook,EXCEPTION>: 2 <officer> {cancels} <UpdateStockBook>. 3 Exit <UpdateStockBook> UseCase.

Table E.6: Use cases for the actor <officer>

<RemoveCustomer> use case

<RemoveCustomer> UseCase: ActorList: <customerServiceClerk:Human>. 1 <RemoveCustomer> starts: 2 <customerServiceClerk> {specifies} <Customer> with its attribute [nr:Natural]. 3 <System> {searches} <Customer>. 4 <customerServiceClerk> {removes} <Customer>. 5 Post <Customer,REMOVED>.	<RemoveCustomer> UseCase alternate flow 3.invalidCustomerAccount: 1 Cause: <Customer,EXCEPTION>: 2 <System> {notifies} "nonexistent"<Customer>. 3 Continue <RemoveCustomer> UseCase in step 2. <RemoveCustomer> UseCase alternate flow 4.orderExists: 1 Cause: <OrderOfCustomer,EXISTS>: 2 <System> {informs} "existent"<OrderOfCustomer>. 3 Exit <RemoveCustomer> UseCase.

<AddBook> use case

<AddBook> UseCase: ActorList: <customerServiceClerk:Human>. 1 <AddBook> starts: 2 <customerServiceClerk> {provides} <Book>. 3 <System> {reviews} <Book,EXISTS>. 4 <System> {adds} <Book>. 5 <System> {prepares} <StockBook> "of"<Book>. 6 <customerServiceClerk> {completes} <StockBook> with [stock:Natural]. 7 Post <StockBook,CREATED>.	<AddBook> UseCase alternate flow 3.bookExists: 1 Cause: <Book,EXISTS>: 2 <System> {displays} "existent"<Book>. 3 Exit <AddBook> UseCase.

<RemoveBook> use case

<RemoveBook> UseCase: ActorList: <customerServiceClerk:Human>. 1 <RemoveBook> starts: 2 <customerServiceClerk> {requests} <Book> with [isbn:String]. 3 <System> {searches} <Book>. 4 <customerServiceClerk> {deletes} <Book>. 5 Post <Book,REMOVED>.	<RemoveBook> UseCase alternate flow 3.nonexistentBook: 1 Cause: <Book,EXCEPTION>: 2 <System> {informs} "nonexistent"<Book>. 3 Continue <RemoveBook> UseCase in step 2. <UpdateStockBook> UseCase alternate flow 2.updateCancellation: ActorList: <officer:Human>. 1 Cause: <UpdateStockBook,EXCEPTION>: 2 <officer> {cancels} <UpdateStockBook>. 3 Exit <UpdateStockBook> UseCase.

Table E.7: Use cases for the actor <customerServiceClerk>

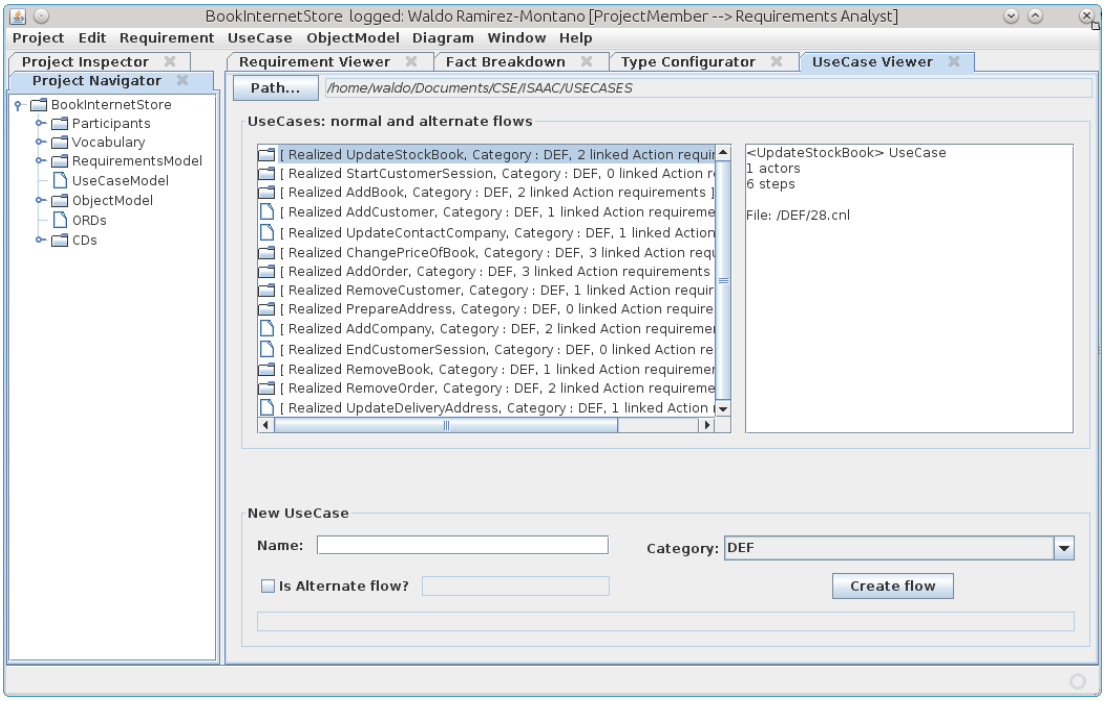


Figure E.1: UseCaseViewer with the case study