

Masterarbeit

CDL - An Extensible Framework to Create Constraints in Model-Based Development

Markus Krallinger

Institut für Technische Informatik
Technische Universität Graz



Begutachter & Betreuer: Dipl.-Ing. Dr. techn. Christian Kreiner

Graz, im Mai 2012

Kurzfassung

Immer häufiger wird modelbasierte Entwicklung in industriellen Anwendungen eingesetzt. Um die Komplexität von modernen Softwaresystemen zu beherrschen, werden diese oft top-down entwickelt. Ausgehend von einer abstrakten Beschreibung des Systems, dem Modell, wird dieses so lange mit Details angereichert, bis das Zielsystem, z.B. Quellcode, Dokumentation oder ein anderes Modell, generiert werden kann.

Modelle sind in diesem Ansatz eine zentrale Datenstruktur und können entweder über eine universelle Modellierungssprache wie z.B. UML oder SysML, oder eine domänenspezifische Sprache (DSL) beschrieben werden. Eine domänenspezifische Sprache ist eine Sprache die speziell für eine bestimmte Domäne entwickelt wurde und durch ihren abgegrenzten Sprachumfang Vorteile gegenüber universellen Sprachen hat. Bei der Entwicklung einer DSL sind üblicherweise zwei Rollen involviert: Der Anwendungsentwickler, verantwortlich für die Entwicklung der Sprache und die Editoren und der Domänenexperte, der seine Expertise dem Entwickler zur Verfügung stellt.

Instanzen dieser Modelle müssen spezielle Anforderungen erfüllen, die z.B. durch den Entwicklungsprozess, gesetzliche Bestimmungen oder den Zweck des Modells gegeben sind. Technische Einschränkungen, die solche Anforderungen sicherstellen, sind typischerweise statisch in das Modellierungswerkzeug implementiert. Nachteile dieser Vorgehensweise sind unter anderem notwendige Kenntnisse über die Struktur des Modells und der verwendeten Programmiersprache, sowie die beschränkte Wiederverwendbarkeit.

Den Inhalt dieser Masterarbeit stellt ein Framework dar, das die Erstellung von Einschränkungen auf Modelle vereinfachen soll. Einerseits soll es dem Anwendungsentwickler ermöglichen, Einschränkungen wiederzuverwenden, andererseits sollten diese vom Domänenexperten ohne Programmierkenntnisse auf die Modelle angewendet werden. Zusätzlich wird der Modellierer bei der Korrektur unterstützt, sollte das Modell nicht konform zu den Anforderungen sein.

Das Framework wurde speziell im Hinblick auf Erweiterbarkeit entwickelt. Der Kern des Frameworks ist die domänenspezifische Sprache "Constraint Definition Language" (CDL), die verwendet wird, um graphisch oder textuell die Einschränkungen zu festzulegen und anzuwenden. Der Anwendungsentwickler kann bestehende Konzepte zur Einschränkungdefinition im Framework wiederverwenden. In dem entwickelten Prototyp können Einschränkungen in OCL, EOL und JAVA formuliert werden.

Das Ergebnis dieser Masterarbeit ist ein erweiterbares Framework, das verwendet werden kann, um Einschränkungen in einem modelbasierten Entwicklungsprozesses zu definieren. Eine prototypische Entwicklung des Frameworks als Eclipse-Plugin und ein Fallbeispiel aus der Automotive-Domäne unter Verwendung der Modellierungssprache EAST-ADL2 zeigt die Durchführbarkeit des Ansatzes.

Schlüsselwörter: Modelbasierte Entwicklung, OCL, EOL, Eclipse, Konsistenz, Einschränkungen, EAST-ADL2

Abstract

The model-based development paradigm has progressed from scientific research to industrial use. To handle the complexity of modern software systems, the system under development is created by refining artefacts along the development process until the main artefact, the model, can be transformed into the required form, e.g. source code documentation or another model.

Models along this development process are usually expressed in a general purpose language, such as UML or SysML, or in a domain-specific language (DSL) that has been developed to capture the concepts of the domain. The development of such a domain-specific language is usually carried out by two roles: the Tool Smith who creates the tooling environment, needed to manipulate the model formulated in the DSL, and the Domain Expert who provides the expertise of the domain to the Tool Smith.

Instances of these models, expressed in a DSL or in a general purpose language, have to fulfill certain requirements stemming from different sources such as the development process, normative regulations or purpose of the model. Constraints to ensure that such requirements are met, are usually statically implemented using a textual programming language. These textual constraint languages have certain drawbacks, e.g. in-depth knowledge of the language's structure to constrain is needed or the re-usability of the defined constraints is limited.

In this thesis, a framework is presented, that eases the development of constraints for both a DSL and a general purpose language, supporting the model-based development process. Using the framework, the Tool Smith can implement re-usable constraints and provide them to the Domain Expert, who can assign these constraints without deep knowledge of the textual constraint language or the structure of the language. The constraints can be augmented with repair actions that support the Modeler in the process of fixing the model, if constraints are not fulfilled.

The framework is designed to be extensive, allowing the Tool Smith to integrate existing approaches to constraint management into the framework. In the prototypical implementation presented in this thesis, OCL, EOL, and JAVA can all be used as constraint languages. The core of the framework is the domain-specific language "Constraint Definition Language" (CDL), which is used to express the constraints in a graphical and a textual notation.

The result of this thesis is an extensible framework that can be used to define constraints along the model-based development process. The feasibility of this approach is shown using a prototypical implementation of the framework as an Eclipse plugin, and a case study from the automotive domain using the domain-specific language EAST-ADL2.

Keywords: Model-Based Development, OCL, EOL, Eclipse, Consistency, Constraints, EAST-ADL2

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

.....
date

.....
(signature)

Danksagung

Diese Diplomarbeit wurde im Studienjahr 2011/12 am Institut für Technische Informatik an der Technischen Universität Graz durchgeführt.

Ich möchte mich besonders bei meinem Betreuer Dr. Christian Kreiner bedanken, der durch intensive Diskussionen und hilfreiche Anregungen besonders zu dieser Diplomarbeit beigetragen hat.

Weiteren Dank gebührt meinen Eltern, Josef und Gertraud, die mir mein gesamtes Studium hindurch finanziell den Rücken gestärkt haben und es mir dadurch ermöglicht haben mich voll und ganz auf meine Ausbildung zu konzentrieren. Danke, ohne euch wäre diese Arbeit in diesem Umfang nicht möglich gewesen.

Besonderer Dank gilt meinen Studienkollegen und Freunden, Philipp, Matthias und den Georgs. Die zahllosen Stunden, die wir mit Diskussionen zu verschiedenen technischen und nichttechnischen Themen verbracht haben, werde ich nie vergessen.

Zu guter Letzt möchte ich auch Kjersti danken, die mir in dieser schwierigen Zeit durch ihre aufbauende Art weitergeholfen hat. Danke.

Graz, im Mai 2012

Markus Krallinger

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction and Motivation | 19 |
| 1.1 | Background | 19 |
| 1.2 | Motivation & Goals | 20 |
| 1.3 | Outline | 20 |
| 2 | Related Work | 23 |
| 2.1 | Model-Driven Engineering | 23 |
| 2.1.1 | Meta-Modeling | 23 |
| 2.2 | Domain-Specific Languages (DSL) | 24 |
| 2.2.1 | Roles within the DSL Development Process | 24 |
| 2.3 | Technologies | 25 |
| 2.3.1 | Eclipse Modelling Framework | 25 |
| 2.3.2 | Textual Constraint Languages | 26 |
| 2.4 | Constraint Management | 28 |
| 2.4.1 | Consistency | 28 |
| 2.4.2 | Completeness | 29 |
| 2.4.3 | Dealing with Incompleteness / Inconsistency | 29 |
| 2.5 | Related Approaches to Constraint Management | 29 |
| 2.5.1 | Logic-Based Approaches | 30 |
| 2.5.2 | Model Checking | 31 |
| 2.5.3 | Special Forms of Analysis | 31 |
| 2.5.4 | Human-based Collaborative Exploration | 32 |
| 2.5.5 | Summary | 32 |
| 2.6 | Requirements for the CDL Framework | 32 |
| 2.7 | Case Study "Functional Safety" | 33 |
| 2.7.1 | Constraints Used throughout the Thesis | 33 |
| 2.8 | Definitions | 34 |
| 3 | Defining Constraints Using CDL | 35 |
| 3.1 | Introduction | 35 |
| 3.2 | Overview CDL Framework Design | 35 |
| 3.3 | Tool Smith View | 37 |
| 3.3.1 | Model Import | 37 |
| 3.3.2 | Classifier | 37 |
| 3.3.3 | Macro Engine | 41 |

| | | |
|----------|--|-----------|
| 3.4 | Domain Expert View | 41 |
| 3.4.1 | Constraint Groups | 42 |
| 3.4.2 | Assignments | 44 |
| 3.5 | Modeller View | 45 |
| 3.5.1 | Execution of the CDL Constraints | 45 |
| 3.6 | Constraint Definition Language (CDL) | 46 |
| 4 | CDL Framework Implementation | 49 |
| 4.1 | Introduction | 49 |
| 4.2 | Syntax & Semantics | 49 |
| 4.3 | The CDL Execution Engine | 50 |
| 4.3.1 | GUI Elements | 50 |
| 4.3.2 | Structural Overview | 51 |
| 4.3.3 | Constraints Execution | 53 |
| 4.3.4 | Constraint Language Plugins | 55 |
| 4.4 | CDL Textual Representation | 56 |
| 4.4.1 | Overview Xtext Framework | 56 |
| 4.4.2 | CDL Grammar | 57 |
| 4.4.3 | Validation | 60 |
| 4.5 | Graphical Representation of the CDL | 60 |
| 4.5.1 | Graphiti Framework Overview | 60 |
| 4.5.2 | Auto Layout | 63 |
| 4.6 | Synchronisation | 64 |
| 5 | Case Study | 67 |
| 5.1 | Modelling in EAST-ADL2 | 67 |
| 5.2 | Implemented Properties | 68 |
| 5.3 | In-Depth Example: "A Requirement is satisfied" on VehicleFeature | 72 |
| 5.4 | Results | 74 |
| 5.4.1 | Lesson Learned | 75 |
| 6 | Conclusion and Future Work | 77 |
| 6.1 | Evaluation with Respect to the Requirements | 77 |
| 6.2 | Summary | 78 |
| 6.3 | Future Work | 78 |
| A | Grammar for the Textual Representation | 81 |
| | Bibliography | 83 |

List of Figures

| | | |
|------|---|----|
| 2.1 | Overview meta-modelling on the example of the UML, based on the OMG modelling stack. | 24 |
| 2.2 | Top-Level overview of the Ecore meta-meta-model | 26 |
| 2.3 | Workflow developing a DSL with a graphical representation using EMF | 27 |
| 2.4 | Overview EAST-ADL2 package structure [EAS10] | 33 |
| 2.5 | All examples throughout this thesis use the element "HazardousEvent" from the EAST-ADL2 meta-model | 34 |
| 3.1 | Relationship of constraints and the domain meta- and instance models | 35 |
| 3.2 | Overview of the CDL framework; activities and artefacts are annotated with the different roles | 36 |
| 3.3 | Conceptual Overview "Classifier", a Classifier consists of the three parts: an optional Precondition, a Criterion and an arbitrary number of Actions. Every part contains a textual constraint expression, called "LanguageEntity". | 38 |
| 3.4 | Schematic overview of the Macro Engine, the three different listings show the information added by the Macro Engine. | 41 |
| 3.5 | Simplified structural overview of Constraint Groups and Assignments | 42 |
| 3.6 | Graphical representation of a Constraint Group | 43 |
| 3.7 | Graphical representation of the Assignment to the meta-class "HazardousEvent", domain meta-model EAST-ADL2 (property 12, Table 2.2) | 44 |
| 3.8 | Screenshot of the CDL framework Problem View | 46 |
| 3.9 | Simplified meta-model of the Constraint Definition Language (CDL), annotated with the different roles | 47 |
| 4.1 | Workflow of defining and executing constraints | 50 |
| 4.2 | Structural overview of CDL Execution Engine | 52 |
| 4.3 | Exemplary execution of constraints with artefacts. For this example, properties 10, 11a and 12 from Table 2.2, are combined in one Constraint Group and assigned to the EClass "HazardousEvent". | 53 |
| 4.4 | Binary Object Tree | 54 |
| 4.5 | Overview Xtext Framework | 56 |
| 4.6 | Top-level elements of the CDL grammar | 57 |
| 4.7 | Syntax elements to import a given Ecore based meta-model | 58 |
| 4.8 | Syntax elements used for macro definitions | 58 |
| 4.9 | Syntax elements used to define a Classifier | 58 |
| 4.10 | Syntax elements used to define a Constraint Group | 59 |

| | | |
|------|--|----|
| 4.11 | Syntax elements used to define assignments | 59 |
| 4.12 | Example indication of a syntax error | 60 |
| 4.13 | Overview of Graphiti data structure | 61 |
| 4.14 | Overview of the Graphiti implementation structure (elements for feature "Assignment" shown, others omitted) | 62 |
| 4.15 | Tooling palette with constraints for the EAST-ADL2 meta-model | 63 |
| 4.16 | Auto-layout of diagram elements | 63 |
| 4.17 | Autolayout of a Constraint Group | 64 |
| 4.18 | Example of a graphical element that needs to be updated | 64 |
| 5.1 | Top-Level structure of EAST-ADL2 | 67 |
| 5.2 | Relation of the class <i>VehicleFeature</i> with the class <i>Requirement</i> | 72 |
| 5.3 | Screenshot of the CDL framework plugins, parts of the CDL framework are annotated | 76 |
| A.1 | Syntax tree for the CDL EBNF grammar | 82 |

List of Tables

| | | |
|-----|--|----|
| 2.1 | Summary of Different Approaches to the Detection of Inconsistencies (Table initially from [SZ01] and confirmed in [LMT09]) | 30 |
| 2.2 | Example properties, used throughout the thesis [MGL ⁺ 11] | 34 |
| 3.1 | Example replacements for the EAST-ADL2 element SafetyGoal in the different constraint languages | 42 |
| 3.2 | Elements of the CDL meta-model mapped to roles and functionalities | 48 |
| 4.1 | Top-Level mapping from CDL syntax elements to their semantic counterparts | 51 |
| 5.1 | Implemented properties and proposed Actions in CDL, properties from [MGL ⁺ 11] and [MAL ⁺ 11] | 71 |
| 5.2 | Comparison of the different prototypes to check constraints on the EAST-ADL2 meta-model | 74 |

List of CDL Source Codes

| | | |
|-----|---|----|
| 3.1 | Import of EAST-ADL2 models | 38 |
| 3.2 | Example implementation of property 12 in Table 2.2 with Precondition, Criterion and Action | 39 |
| 3.3 | Sample replacement definition | 41 |
| 3.4 | Textual representation of a Constraint Group | 43 |
| 3.5 | Textual representation of the Assignment for the element "HazardousEvent" (property 12, Table 2.2) | 45 |
| 5.1 | Implementation of property 23 from Table 5.1: declaration of the Classifier and Criterion | 72 |
| 5.2 | Action for property 20 from Table 5.1: query the user for an existing requirement and assign it to VehicleFeature. | 73 |

Chapter 1

Introduction and Motivation

1.1 Background

The model-based development paradigm has progressed from scientific research to industrial use. The main objective of this approach, is to use the model of the developed system as the main development artefact; the real system (source code, documentation or a process description) is derived, generated or transformed from the model.

Inherent to the model-based development paradigm is the concept of meta-modelling. Meta-modelling states, that a model has to conform to its language specification, the meta-model. Based on a general meta-meta-model, the Meta Object Facility (MOF), the language engineer can create a specialized meta-model that provides all the concepts for the model of the developed system.

The foundation of meta-modelling provides a powerful mechanism to create domain-specific languages to capture the specific concepts within different application domains, instead of using general purpose languages such as SysML or UML. Language workbenches, such as the popular Eclipse Modelling Framework, facilitates the creation of such languages and their editors.

A strong focus within the model-based development paradigm lies on the division of the development process into several abstraction layers. The Object Management Group (OMG) proposes a four-layered development process, the domain-specific language EAST-ADL2 is partitioned into four, connected layers. This separation into the different abstraction levels is essential to tackle the complexity of the developed system, but can be a source of certain inconsistencies stemming from contradicting descriptions of the elements within the model.

Besides structural constraints introduced through the development process, the developed model has to fulfill certain semantical constraints that are derived from the purpose of the model. Certain domain-specific languages require the alignment to norms, qualitative regulations or organisational regulations that restrict the content of the model. In the automotive domain, the model has to conform, among others, to the functional safety norm ISO26262.

These constraints, both syntactic and semantic, are nowadays often described using textual programming languages such as the Object Constraint Language (OCL), Eclipse Epsilon Language (EOL), JAVA, or certain tool-proprietary scripting languages. However,

all of these approaches require a deep knowledge of the programming languages and the meta-models involved. Besides the industrial approaches, there exists several academic approaches that facilitate techniques such as model-checking or logical constraint solving.

1.2 Motivation & Goals

The motivation to write this thesis, was to create the Constraint Definition Language (CDL), a framework that allows the different end-users (Tool Smith, Domain Expert and Modeller) to create semantic and syntactic constraints in order to ensure the developed model fulfills its requirements. Furthermore, the framework should support the Modeller, when repairing the identified defects of the model.

Consequently, the goals of the CDL framework are the following:

- *Provide support for the Modeller to repair the model:*
The framework shall support the Modeller in repairing the defects on the model, identified by the framework. This repair mechanism should facilitate rich GUI elements in order to query the content of the model. Furthermore, all repair-actions need to be revertable.
- *Consider the role of the end-user:*
The framework shall provide different views for the different roles of the language-development process. In that sense, the complexity of the meta-model should be hidden from the Domain Expert and the Modeller because they may not have in-depth knowledge of meta-modelling.
- *Allow the integration of other constraint-definition approaches:*
The framework shall be extensible to integrate other approaches of constraint definition.
- *Enable the creation of reusable constraints:*
The framework shall allow the creation of constraints that can be reused for different models and different elements.

1.3 Outline

This thesis is structured as follows:

Chapter 2 gives an overview of the relevant background information: **Section 2.1** briefly introduces the concept of model-driven development, **Section 2.2** describes the process of developing a domain-specific language. In **Section 2.3**, the concepts introduced in the first two sections are mapped to the technology that was used to implement them. **Section 2.4** introduces the term "consistency" and discusses the different concepts connected with consistency. In **Section 2.5**, related approaches to consistency management are presented and discussed. These lead to the requirements of the CDL framework, which are stated in **Section 2.6**. **Section 2.7** gives an overview on the case study that is used throughout the thesis to illustrate the capabilities of the framework.

In **Chapter 3**, the Constraint Definition Language (CDL) and the CDL framework is presented. **Section 3.1** and **Section 3.2** introduce the ideas of the framework and give an overview. The subsequent sections describe the activities for every role in detail: Tool Smith in **Section 3.3**, Domain Expert in **Section 3.4**, and Modeller in **Section 3.5**. **Section 3.6** concludes with a description of the Constraint Definition Language and its meta-model.

While Chapter 3 shows the functional capabilities of the CDL framework, **Chapter 4** shows how those concepts are implemented: **Section 4.1** gives an overview of the different aspects of the implementation, **Section 4.2** shows the mapping from the CDL syntax to their semantic counterparts. The subsequent sections present the three different plugins that were implemented to provide the functionality described in Chapter 3: **Section 4.3** describes the part of the framework that executes the constraints, **Section 4.4** describes the textual editor and the grammar of the textual representation, and **Section 4.5** describes the graphical representation. In **Section 4.6**, a mechanism to synchronise both the textual and the graphical representation is presented.

Chapter 5 presents a case study where the CDL framework was used to define constraints on the EAST-ADL2 meta-model. **Section 5.1** introduces the domain-specific language EAST-ADL2, **Section 5.2** lists the properties implemented in the case study. In **Section 5.3** one Classifier is exemplary presented, **Section 5.4** concludes the case study by comparing the CDL framework to a property checker with similar functionality.

Chapter 6 concludes this thesis: in **Section 6.1**, the CDL framework is evaluated with respect to the requirements defined in Section 2.5, **Section 6.2** presents an short summary of the CDL framework and **Section 6.3** discusses three conceptual possibilities to improve the Constraint Definition Language framework.

Chapter 2

Related Work

2.1 Model-Driven Engineering

“Model-driven development is simply the notion that we can construct a model of a system that we can then transform into the real thing” [MCF03].

The complexity of today’s software systems is overwhelming. Software systems are distributed, heterogeneous and highly elaborate. One way to handle this complexity is to raise the abstraction level of the system under development. There are several examples where software systems are no longer developed using source code, but are generated. The data describing such systems is captured in a model, a data structure that specifies the layout of the included data. The model is the key artefact. Depending on the tool, it is usually manipulated using graphical or textual editors.

The Object Management Group (OMG) [Gro12b] provides a *conceptual* process framework to support this approach, the model-driven architecture (MDA). In MDA, as specified by the OMG, the system under development starts as a Computational Independent Model (CIM), and is transformed into the Platform Independent Model (PIM) and further into the Platform Specific Model (PSM). Transforming the PSM further, leads to the actual system, the result of the engineering process.

2.1.1 Meta-Modeling

In the model-driven architecture, the model itself is defined using several abstraction layers. The OMG standard describes a meta-meta-model called Model Object Facility (MOF) which serves as the meta-model for the (instance-) meta-model. The MOF specification describes two variants: the Complete Meta Object Facility (CMOF) and the Essential Meta Object Facility (EMOF). The CMOF is a complete description for all meta-modelling facilities standardized by the OMG and thus, is quite extensive. EMOF on the other hand is a standalone subset of the CMOF, providing all concepts necessary to model object-oriented systems [Gro11].

This approach leads to a four-level structure: the lowest level (M0) represents the real system that is created/derived/generated using the model under development. This model is created on the second level (M1) and conforms to its meta-model. In the example illustrated in Figure 2.1, the real system is (generated) source code, whereas the model is

an UML model. The UML model itself conforms to its UML meta-model defined in the third layer (M3). The third layer describes the UML meta-model that is defined using MOF. In the fourth layer (M3) MOF is defined, it is itself described in MOF.

Another way to look at meta-modeling is, that a model has to conform to its language specification, the meta-model. When raising the abstraction layer, the meta-model becomes the model and therefore, has to conform again to its meta-model and so on. In that sense the meta-model defines which language elements are available for the given model.

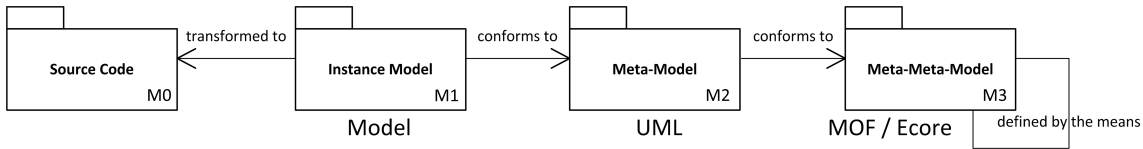


Figure 2.1: Overview meta-modelling on the example of the UML, based on the OMG modelling stack.

Besides MDA, the term "Model-Driven Development" (MDD) and "Model-Driven Engineering" are used. Both denote a development process that focuses on models as the primary artifacts. These models should then be used as an input for automated processing steps, such as code generation or verification. An example for such a process is the development of embedded systems in the automotive domain, where Matlab/Simulink¹ are widely used.

2.2 Domain-Specific Languages (DSL)

Based on the foundation of meta-modelling, it is possible to create specialized languages that cover a (sub-) set of concepts given by a domain. This approach aims to create a specialized language, that fulfills all the needs of a certain domain instead of using a general purpose language such as UML or SysML². Domain-Specific Languages are described by using an abstract syntax that defines the internal storage of the data and the concrete syntax that presents the data in a human readable notation [Kle08]. The abstract syntax can be specified using a meta-model (level M2) or the profiling mechanism of UML³. The concrete syntax is usually implemented using a textual or graphical notation.

2.2.1 Roles within the DSL Development Process

In the process of creating domain-specific languages, Kleppe [Kle08] and Gronback [Gro09] describe different roles: the language engineer or Tool Smith and the language user or Practitioner. In this work the language engineer is called Tool Smith, and the role of the language user is separated into the Modeller and the Domain Expert.

The Modeller is responsible for creating the instance model using the tools and concepts provided by the Tool Smith. The Domain Expert is responsible for providing the domain

¹<http://www.mathworks.de/products/simulink/>

²<http://www.omgsysml.org/>

³http://www.omg.org/technology/documents/profile_catalog.htm

knowledge to the Tool Smith in order to capture the necessary concepts. In the constraint definition framework, the Domain Expert is responsible for creating a constraint set to ensure that the constraints of the domain are obeyed. In this work it is assumed, that the Domain Expert does not have any technical background in the field of creating model-based tooling environments. The Tool Smith on the other hand, is an expert in the field of meta-modelling and responsible for creating the tooling environment for the Modeller to be able to create the model in the domain-specific language. Within the CDL framework, the role of the Tool Smith is to create constraints, formulated in a constraint language.

2.3 Technologies

In this section the model-based development concepts are mapped to existing technologies within the Eclipse Modelling Framework⁴ (EMF). Eclipse was chosen because it provides a highly customizable, open source framework to create a model-based tooling environment. Furthermore, due to the powerful plugin-mechanism, a lot of plugins already ease the development of a model-driven tooling environment.

2.3.1 Eclipse Modelling Framework

The CDL framework is built on top of the Eclipse Modelling Framework (EMF). EMF provides several facilities to support the creation of model-based development tools. The technological foundation is provided by the EMF core framework that includes an implementation of the EMOF meta-meta-model, as specified by the OMG, called Ecore. Figure 2.2 shows the hierarchical overview of the core components.

The EMF framework is quite elaborate, therefore, this section can only provide the necessary background information of the framework.

Tooling Facilities

As depicted in Figure 2.3, the traditional way of developing a DSL using EMF is first to specify the abstract syntax (meta-model, M2) using the Ecore meta-meta model and derive a generator model (.genmodel file) that serves as input for the Java Emitter Templates⁵ (JETs). These generate JAVA files or other artefacts such as serialization/deserialization facilities based on the meta-model. In Figure 2.3, this step is denoted as "Develop Domain Model". These JAVA files serve as input for the concrete syntax (editor), which is then developed either as textual (using a Textual Modelling Framework, TMF, usually Xtext) or graphical editor (for example with Graphiti or other sub-projects of the Graphical Modelling Project⁶, GMP or its synonym Graphical Modelling Framework, GMF).

The instance model (M1) created with these editors serves as the input for either a Model-To-Text (M2T) creating e.g. JAVA files or a Model-To-Model (M2M) transformation. Both types can transform the instance model vertically or horizontally.

The concrete syntax, as well as components for persisting the instance model are provided as Eclipse plugins and can be used by the Modeller.

⁴<http://www.eclipse.org/modeling/emf/>

⁵<http://www.eclipse.org/modeling/m2t/?project=jet#jet>

⁶<http://www.eclipse.org/modeling/gmp/>

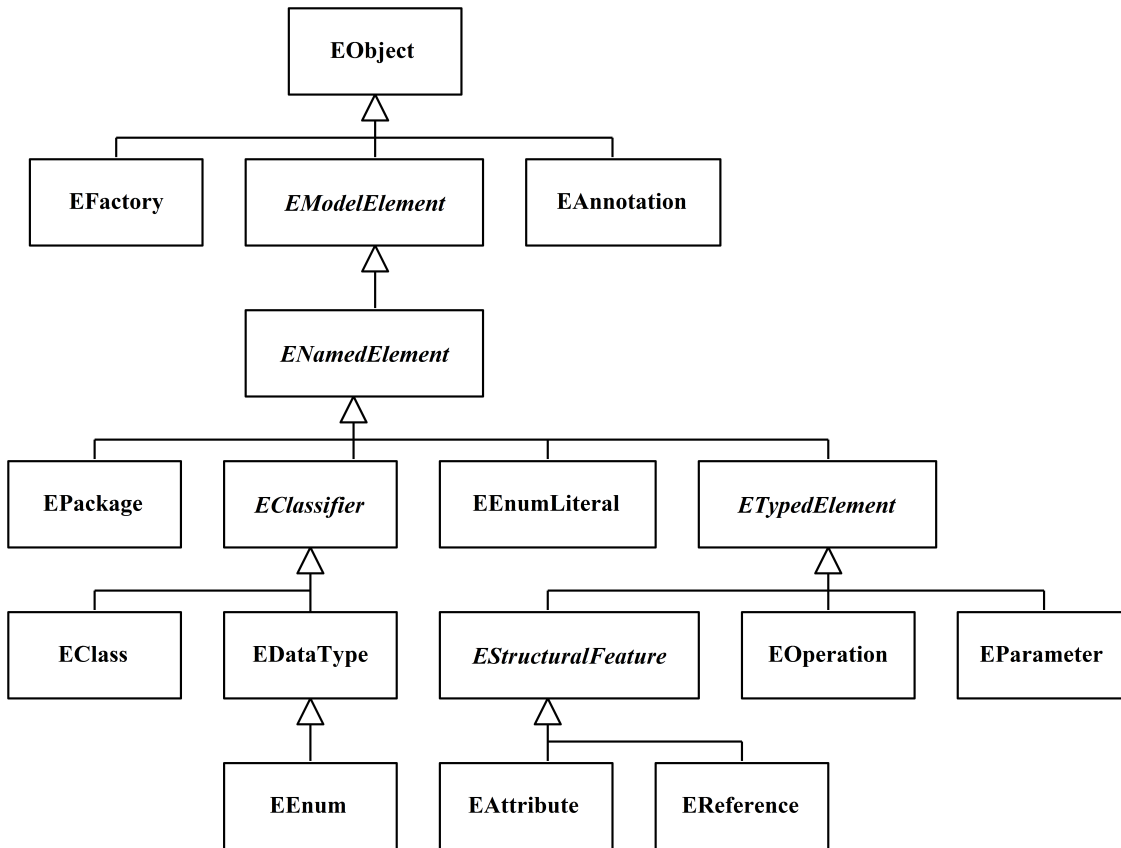


Figure 2.2: Top-Level overview of the Ecore meta-meta-model

Note: There are several frameworks that ease certain steps in this process; an overview can be found on the EMF homepage⁷.

Dynamic EMF

In case the facilities to deserialize the instance model, as described in Section 2.3.1 are not loaded in the current Eclipse instance, EMF provides a dynamic mechanism that relies on reflection, to access the elements in the model. That mechanism enables the CDL framework to process domain models even if the creating editor is not loaded or the meta-model is unknown in the current Eclipse instance.

2.3.2 Textual Constraint Languages

The EMF framework only checks if a model conforms to its meta-model up to a certain extent. The Cardinality of connectors e.g. is not checked. Furthermore, the conformance relationship cannot make any statements about the contents of the attributes of the elements (e.g. that a name is set) or structural characteristics of the instance model.

⁷<http://www.eclipse.org/modeling/>

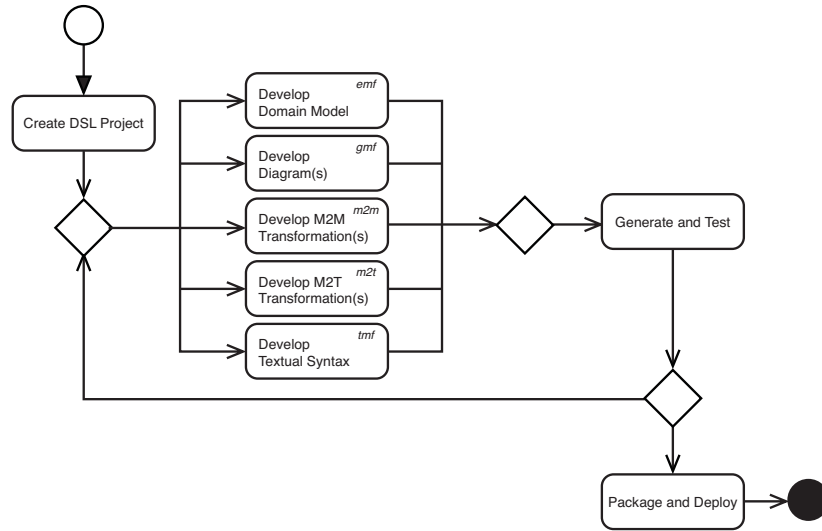


Figure 2.3: Workflow developing a DSL with a graphical representation using EMF [Gro09], see Section 2.3

To be able to formulate such statements, certain textual constraint languages were developed. In the prototypical implementation of the CDL framework the following textual constraint languages are considered: Object Constraint Language (OCL) [Gro12a], Epsilon Object Language (EOL) [KPP06] or JAVA (used as constraint language).

Object Constraint Language (OCL)

The Object Constraint Language is a textual specification language standardized by the OMG. The language was developed 1995 as Integrated Business Engineering Language (IBEL) by IBM and was merged into the UML 1.1 standard in 1997. The current specification is version 2.3.1 from January 2012.

Conceptually, OCL is a pure specification language [Gro12a] and side-effect free. Furthermore, OCL follows a descriptive language paradigm. In the first version, the OCL standard targeted the UML meta-model, whereas the current version is adapted to support the model-driven architecture paradigm.

OCL's initial purpose was to add formal specifications to ambiguous, semi-formal UML models. According to the official language description, OCL can be used as a (1) query language, (2) to define invariants on classes, types and stereotypes, (3) to define pre- and postconditions on operations and methods, (4) to describe guards, (5) to specify targets for messages and actions, (6) to specify constraints on operations and (7) to specify derivation rules for attributes for any expression over an UML model [Gro12a]. In order to support querying the model, OCL provides methods from set-theory such as select subset from a set or collection, iterate over a set or collection or the *exists* quantifier. The Eclipse Modeling Framework provides several OCL interpreters.

OCL in Ecore A rather new approach called OCLinEcore⁸, presented at the Eclipse Summit Europe 2010, combines meta-modeling with Ecore and the constraint-capabilities of OCL in a textual editor, using its own domain specific language. In the textual editor the Tool Smith creates the meta-model in a textual way and embeds the OCL constraints with the elements of the model in the same editor. The result is an annotated meta-model. OCL expression are validated constantly when the Modeller creates the instance-model.

Eclipse Epsilon

The shortcomings of OCL, such as lack a of model-modification, motivated Kolovos *et al.* [KPP06] to create a family of textual languages, the Epsilon Object Language (EOL). A sub-language of EOL, the Epsilon Validation Language (EVL) has the capabilities to define constraints, as well as fixing actions when the constraint does not hold. The EOL languages follow an imperative paradigm, in contrast to the declarative OCL. EOL is extensible, allowing the Tool Smith to embed JAVA classes into the constraint code. Furthermore, EOL provides easy access to GUI elements such as input-boxes, lists or message boxes in order to provide visual feedback or interact with the Modeller.

JAVA as a Constraint Language

The use of Eclipse as a language workbench makes it possible to use JAVA as a constraint language. EMF provides highly elaborate mechanisms to serialize/unserialize, query and alter any supported model. The only limitation is, that the current Eclipse instance has to be able to load the persisted model, e.g. the plugin used to create the model has to be loaded, or the model has to be persisted in XMI⁹ notation.

2.4 Constraint Management

2.4.1 Consistency

In [HKRS05], consistency is described as lack of contradiction in a system of properly related artifacts (of the UML model). Lange *et al.* [LCM⁺03] defines consistency as *''soundness of a design''*.

Consistency can be refined into intra-consistency (horizontal consistency) and inter-consistency (vertical consistency). Intra-consistency is understood to be the consistency within the model, meaning that elements created with different views have to be consistent with each other. Inter-consistency describes the consistency along the levels of abstraction.

In [Str05], the term consistency is refined into syntactical and semantic consistency: Syntactical consistency is understood to be the conformance to the abstract syntax of the meta-model, whereas semantic consistency is defined as *''violations that cannot be defined as conformance violations to the abstract syntax''* [Str05]. Semantic consistency, therefore, ensures that the model fulfills its purpose as well as it conforms to the requirements of the given domain.

⁸<http://wiki.eclipse.org/MDT/OCLinEcore>

⁹<http://www.omg.org/spec/XMI/2.1/>

In [LCM⁺03] the term "well-formedness" is introduced as a set of "soundness restrictions" that ensure the correct usage of class diagrams (e.g. that every object has to have a name or that attributes have to be set private). Since the two terms "well-formedness" and "semantic consistency" greatly overlap and describe the same concepts, "semantic consistency" is used in the remainder of this work.

Using the model-based development paradigm, the conformance relation between the instance model and its meta-model can be seen as syntactical consistency, where as the compliance to constraints of the domain can be seen as semantical consistency (Table 2.2 shows such a list of constraints for the application of a Hazard & Risk analysis using the EAST-ADL2 meta-model).

2.4.2 Completeness

Lange et al. [LCM⁺03] defines completeness (with respect to the model) as "*Completeness of a design is concerned with the fact that the presence of information in some diagrams requires the presence of other information in another part of the design*". In that sense completeness is constantly infringed upon during the development process since not all elements and their relations can be created at the same time. This is known as *temporal incompleteness* [WGN03].

2.4.3 Dealing with Incompleteness / Inconsistency

Nuseibeh *et al.* [NER00] present a managing framework to deal with inconsistencies in general artefacts in software development: They claim that constraints defined in the framework lead to a model / software program that is consistent with respect to this constraints, but not inconsistency-free (without unreasonable, additional work and expenses). Furthermore, they state that the constraints often are not stated explicitly. They emerge from several sources such as the model language, development methods or processes, local constraints, and application domains.

2.5 Related Approaches to Constraint Management

The evolution of UML as an industrial standard for modelling led to certain efforts trying to ensure consistency between the different artefacts and abstraction levels. A survey by Spanoudakis and Zisman [SZ01] in 2001 identified four major approaches to consistency management, which are presented in Table 2.1. Another survey, an exhaustive literature review carried out in 2009 [LMT09], confirmed the categories that had been identified. Lucas *et al.* analyzed 42 papers on the topic of "consistency management" and categorized the different approaches into the four major approaches. Furthermore, the different papers were evaluated for their practical applicability, extendability and the possibility of generalising the approach for model-based development.

The authors concluded, that a method for the widespread use of constraint management, has to fulfill the following three requirements [LMT09]:

1. *Include a mechanism to extend the proposal in order to facilitate the managing of new models and inconsistency problems.*

2. Tackle inconsistency problems related to vertical consistency, since they have been less frequently studied.
3. Fully integrate all of the features stated in the proposal within a CASE tool. This will thus permit its use and validation outside the academic scope.

| Approach | Main Assumptions | Positive Features | Limitations |
|---------------------------------------|---|---|---|
| Logic-based | Models expressed in some formal language | <ul style="list-style-type: none"> • Well-defined inconsistency detection procedures with sound semantics • Applicable to arbitrary consistency rules | <ul style="list-style-type: none"> • First-order logic is semi-decidable • Theorem proving is computationally inefficient |
| Model checking | It must be possible to express or translate models in the particular state-oriented language used by the model checker | <ul style="list-style-type: none"> • Well-defined inconsistency detection procedures with sound semantics | <ul style="list-style-type: none"> • Not efficient due to explosion of states • Only specific kinds of consistency rules (e.g. reachability of states) can be checked |
| Special Forms of Analysis | Models need to be expressed in a specific common language (e.g. conceptual graphs, UML, Petri Nets, XML) or need to be translated into it | <ul style="list-style-type: none"> • Well-defined inconsistency detection procedures | <ul style="list-style-type: none"> • Only specific kinds of consistency rules can be checked |
| Human-based collaborative exploration | Models (or parts of models) expressed in informal modelling languages | <ul style="list-style-type: none"> • Only method for informal models | <ul style="list-style-type: none"> • Labour intensive and difficult to use with large models |

Table 2.1: Summary of Different Approaches to the Detection of Inconsistencies (Table initially from [SZ01] and confirmed in [LMT09])

The remainder of this section presents selected representative work for the all categories, presented in Table 2.1, and discusses the feasibility of the approaches.

2.5.1 Logic-Based Approaches

Van der Straeten *et al.* [Str05] present a method where the UML meta-model is formalised using Descriptive Logic and reasoned upon with a rule-based system. Inconsistencies are detected automatically, and a set of possible inconsistency resolutions is presented to the user that then executes the most appropriate of them.

Ossami *et al.* [OJS05] present a method where a specification in the formal language B is created next to the UML model, and consistency rules are defined on the two representations, allowing reasoning on the B specification.

Malgouyres *et al.* [MM06] describes an approach where the UML model, as well as the UML meta-model (based on MOF) are modelled using Constraint Logic Programming (CLP), enabling the formalisation of identified consistency rules.

Formal approaches, however, rely strongly on a representation of the domain meta-model in the given formal environment, reducing the practical application for domain-specific languages. Furthermore, the extendability of such approaches is questionable: none of the works in the category "logic-based approaches" was considered extensible in the literature review.

2.5.2 Model Checking

Model checking approaches, such as [ZLQ06] or [DH04], are suitable to check the consistency between behavioural views of the model. Both approaches aim to keep UML state diagrams consistent to their specification, modelled as a UML sequence diagram. To achieve that, both behavioural representations are transferred to a formal specification, which serves as input for a model-checker. The model-checker then executes the specification (usually the sequence diagram) on the state-machine and creates an error-trace in case the two representations are not consistent.

The model checking approach seems beneficial for checking the consistency between state- and sequence diagrams. However, similar to the logical-based approaches, the mapping between the input model and the formal specification is rather unpractical.

2.5.3 Special Forms of Analysis

This category mainly contains non-formal approaches. Most of them use a specific programming language in order to express the constraints they define on the model. The textual constraint languages presented in Section 2.3.2 also fall into this category.

Nentwich *et al.* [NEFE03] present an approach to manage consistency between certain UML modelling artefacts using the tool *xlinkit* and a XML-based constraint language. In a subsequent paper [NEF03], identified defects in the model can be repaired using predefined repair actions (add, delete, and change). One interesting aspect of their approach, is extensibility: it is possible to constrain generic artefacts (such as JAVA source code) with their tool. To accomplish this, *xlinkit* provides the possibility of creating generic adapters allowing the integration of information from outside the model. Unfortunately, the approach was never generalised for model-based development and the tool *xlinkit* is not available anymore.

Another example for this category is provided by Wagner *et al.* [WGN03]. They present a framework that tackles inconsistency using a graph-based notation, incremental checking and an extensible mechanism to provide (user-defined) repair actions. Furthermore, the different roles in the development process are taken into account; they separate between the Administrator and the Modeller. However, using a graph-grammar implies that only horizontal consistency can be checked, since the elements to define the rules have to come from the domain meta-model.

As presented in Section 2.7, Mader *et al.* also follow this approach: in their work certain constraints that *indicate* the correct application of a Hazard & Risk Analysis are identified. In contrast to other works presented in this section, Mader *et al.* do not ensure consistency with other artefacts, they solely ensure that the regulative requirements of the ISO26262 are met. However, they implemented the rules hard coded into their editor, and for this particular method, not being able to adapt the constraints throughout the development cycle or use the rules for a different method.

2.5.4 Human-based Collaborative Exploration

In the original work from Spanoudakis and Zisman [SZ01] several approaches were presented that facilitate structured inspection that allows the identification of inconsistencies on the model. However, since models nowadays are more complex and distributed, the applicability of this approach is questionable.

The human-based collaboration on the other hand, has a high relevance when it comes to choose actions to repair the model when a defect is identified. In [NEF03], [Str05] and [WGN03] repair actions are presented to the Modeller where he needs to evaluate the most relevant solution to the defect.

2.5.5 Summary

All the investigated works solve subparts of the challenges connected with consistency and completeness issues in model-based development. Certain approaches just work on the UML model, others require a transformation into a formal environment to be processed further. Others cannot be generalised for use in a model-based environment, apart from the UML model.

2.6 Requirements for the CDL Framework

Based on the literature research from Section 2.5, a mechanism should be implemented that fulfills the following requirements:

- RQ1** The constraint mechanism shall be based on MOF in order to be applicable to all models, no assumption should be made on the editor creating the model.
- RQ2** The constraint mechanism shall be extensible to support models from different abstraction levels in order to support vertical consistency checking.
- RQ3** The constraint mechanism shall be able to integrate existing consistency approaches, such as those presented in Section 2.5 in order to reuse existing effort.
- RQ4** The constraint mechanism shall support the processing of information from *outside* the model in order to allow the definition of semantic constraints.
- RQ5** The constraint mechanism shall be implemented using an open-source modeling tool in order to be validated by others as well.
- RQ6** The constraint mechanism shall support the Modeller in repairing his instance model.

The realization of these requirements as well as an evaluation of the CDL framework based on these requirements is described in the subsequent chapters.

2.7 Case Study "Functional Safety"

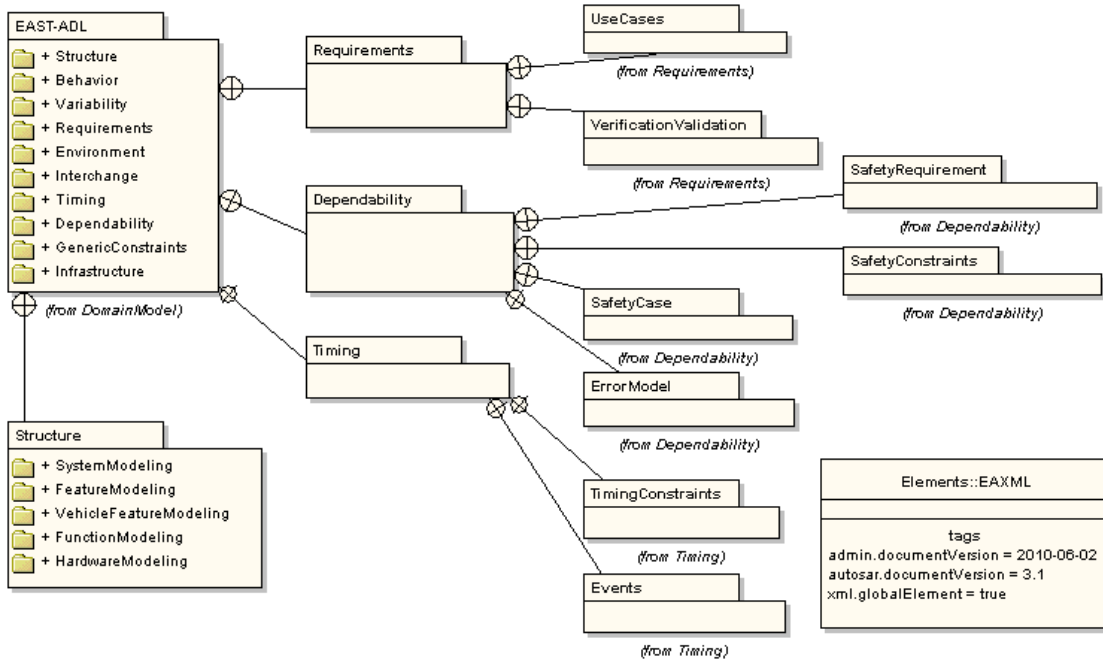


Figure 2.4: Overview EAST-ADL2 package structure [EAS10]

To show the capabilities of the framework described throughout this thesis, constraints from Mader *et al.* [MGL⁺11] and [MAL⁺11] were implemented in the Constraint Definition Language. In this work the authors describe a tool supported method for Hazard Analysis according to ISO26262 [ISO09], a functional safety regulation from the automotive industry. The authors define certain rules that *indicate* the correct application of the method and present a tool prototype that enforces these rules and offers repair actions to the Modeller once a rule is violated.

The domain meta-model used for the examples in this thesis is EAST-ADL2 (top-level package structure illustrated in Figure 2.4), provided as a plug-in to the Papyrus editor¹⁰.

In Chapter 5 the complete case study is presented. All properties from Mader *et al.* are implemented, showing the capabilities of the framework in a real-world example.

2.7.1 Constraints Used throughout the Thesis

The examples, that illustrate the capabilities of the CDL framework throughout this thesis, use one of the properties presented in Table 2.2. The element constrained by

¹⁰www.papyrusuml.org/

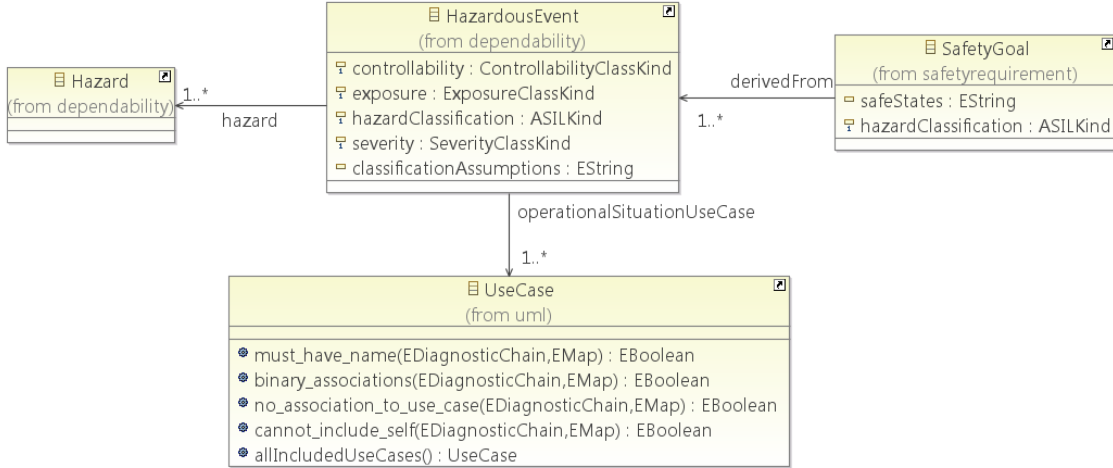


Figure 2.5: All examples throughout this thesis use the element "HazardousEvent" from the EAST-ADL2 meta-model

these properties, is "HazardousEvent" from the package *Dependability* in the EAST-ADL2 meta-model (see Figure 2.5).

| ID | Model Element | Property |
|-----|----------------|---|
| 10 | HazardousEvent | Every HazardousEvent is associated with a Hazard |
| 11a | HazardousEvent | Every HazardousEvent is associated with an Use Case |
| 12 | HazardousEvent | Every HazardousEvent that has an ASIL greater than QM is associated with a SafetyGoal |

Table 2.2: Example properties, used throughout the thesis [MGL⁺11]

2.8 Definitions

In order to avoid confusion with the different instance- and meta-models the meta-model of the domain (EAST-ADL2) is called **domain meta-model** and the instance model is called **domain instance model** throughout this work.

In the remainder of this thesis, the following terms denote a concept from the Constraint Definition Language, and therefore, start with a capital letter: *Classifier*, *Precondition*, *Action*, *Criterion*, *Constraint Group* and *Assignment*.

Chapter 3

Defining Constraints Using CDL

3.1 Introduction

In this chapter, the Constraint Definition Language Framework (CDL) is presented from each of the different user group's perspectives. The main purpose of this chapter is to show *what* can be done with the framework and not how it is implemented.

The remainder of this chapter is structured as follows: in Section 3.2, the overall framework is described conceptually, the subsequent sections Tool Smith Role, Domain Expert Role and Modeller Role illustrate the concepts and activities for the different end users of the CDL framework. Section 3.6 discusses the language structure of the CDL language, as perceived by the end user.

3.2 Overview CDL Framework Design

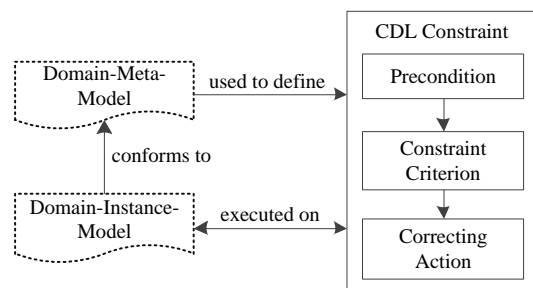


Figure 3.1: Relationship of constraints and the domain meta- and instance models

The CDL framework's objective is to provide an extensible platform to create constraints in order to ensure consistency (horizontal and vertical constraints as well as semantical and syntactical constraints, c.f. Section 2.4). It is technically based on the Eclipse Modelling Framework (EMF). The CDL framework connects an Ecore based domain meta-model and its domain instance model with different interpreters of the constraint languages.

The framework does not make any assumptions on the editor (textual or graphical) that the Modeller or the Domain Expert work with. The CDL framework uses the domain

meta-model to define constraints, that are executed on the domain instance model. The relationship between the elements of the framework and the imported models is depicted in Figure 3.1.

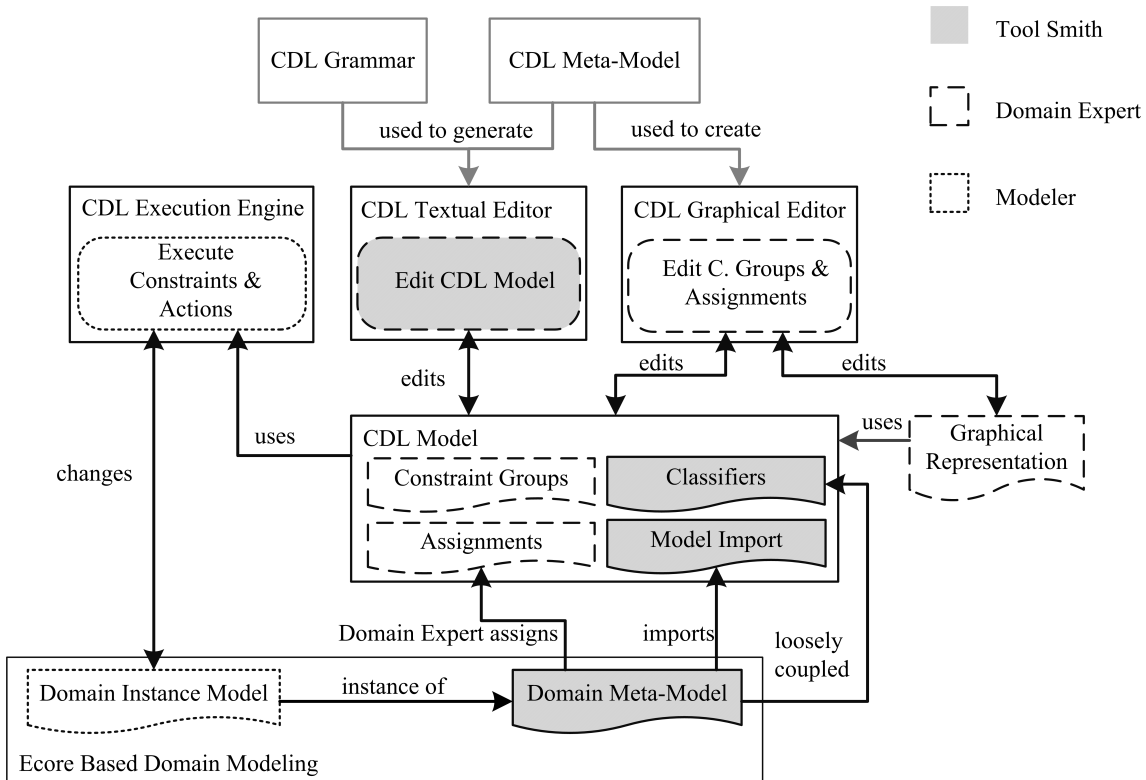


Figure 3.2: Overview of the CDL framework; activities and artefacts are annotated with the different roles

One key-aspect novel with the CDL framework is the separation of the different roles in the development process of a domain-specific language, as depicted in Figure 3.2: The Tool Smith imports the required models (domain meta-model and domain instance model) and creates reusable Classifiers containing three parts, namely Precondition, Criterion and Actions (c.f. Section 3.3.1 and 3.3.2). Each different part can be written in a different textual constraint language. A Classifier representing an atomic constraint concept, is combined logically by the Domain Expert and assigned to elements from the domain meta-model (c.f. Section 3.4.1 and 3.4.2). Finally, the Modeller executes the constraints that identify defects on his domain instance model, which are repaired using the Actions included in the Classifier.

Throughout, there is the option to use JAVA or EOL to create Actions. The Tool Smith can use rich GUI elements to provide feedback or link to documentation that helps the Modeller to repair the model. Moreover, the Tool Smith can use data from outside the model in his constraints. It's possible to use a database or web service as input, as well as using values from outside the model to repair the model once one Criterion does not hold.

Furthermore, the framework aims to enable reuse of the defined constraint code. To

achieve this goal, the framework provides certain mechanisms to automatically add the context / enclosing packages to constraint expression enabling the expression to be defined without a context. Additionally, the option to define macros lets the Tool Smith write shorter constraint code and enables reuse of the code in different models.

3.3 Tool Smith View

The Tool Smith is responsible for building the foundation for the activities of the Domain Expert and the Modeller. In that role, the Tool Smith has to import the required domain models and implement the *textual constraint expressions* inside the different Classifiers, c.f. Section 3.3.2.

In the remainder of this work, a textual constraint expression is a boolean constraint, implemented using one of the supported constraint languages (in the prototypical implementation: EOL, OCL or JAVA).

3.3.1 Model Import

The first activity in the Tool Smith role is to import the domain instance model and the corresponding domain meta-model. The only limitation is that the domain-meta model has to be Ecore-based and the EMF subsystem has to be able to load the domain instance model.

When importing models, the Tool Smith has two possibilities: (1) importing the model using an Ecore file, or (2) importing a meta-model using an URI. In the first case the domain instance model has to be provided in the XMI format since the EMF subsystem cannot otherwise resolve the model. In the more common second case, the domain meta-model is already loaded or known in the current Eclipse instance (for example in the case of UML), and therefore the domain instance model can be persisted in the way the plugin intends.

To illustrate the concept consider the following example: A plugin created by the textual modeling framework Xtext¹ persists its model in the textual form, specified by the grammar of the textual language. If the Xtext plugin is loaded inside the current Eclipse instance, the Tool Smith can import the domain meta-model using the URI of the Xtext model and imports the textual representation of the domain instance model. If the plugin is not loaded, the Tool Smith has to import the domain meta-model using the Xtext-generated Ecore file and convert the textual notion of the model into its XMI representation. An example for a model import for the EAST-ADL2 meta-model is shown in Listing 3.1.

3.3.2 Classifier

The Tool Smith is responsible for providing the elements that the Domain Expert uses to constrain their domain meta-model. In the CDL framework these elements are called "Classifiers". A Classifier represents an atomic constraint concept, e.g. ensures that a given attribute is set to a specific value. Classifiers are logically grouped and assigned to the corresponding elements of the domain meta-model by the Domain Expert. When executed, all instances of the given meta-class are read and serve as an input to the Classifier.

¹<http://www.eclipse.org/Xtext/>

```

1 classifier model: RecuperationModel {
2   is file based
3   package: eastadl
4   model file :
5     'platform:/resource/RecuperationUseCase/model/recuperation_use_case.uml'
6   metamodel file :
7     'platform:/resource/RecuperationUseCase/include/eastadl.ecore'
8 }

```

Listing 3.1: Import of EAST-ADL2 models

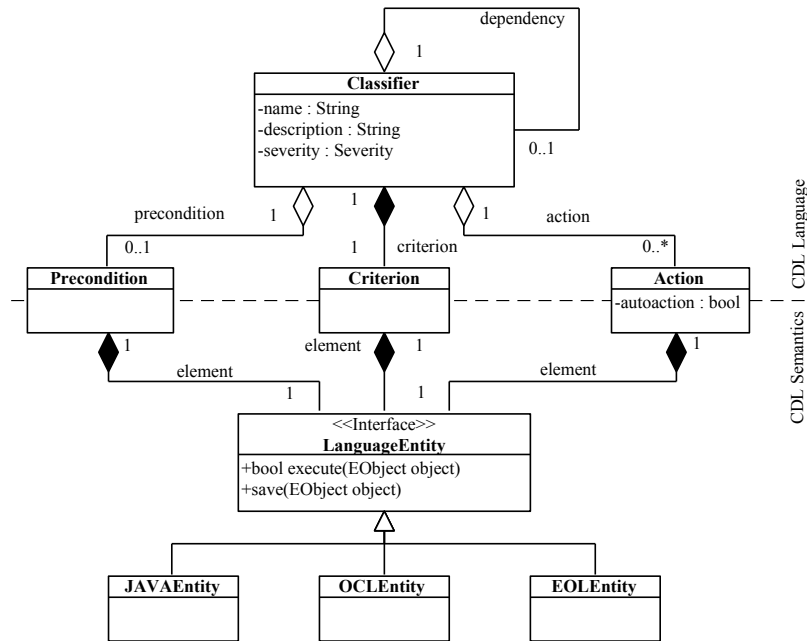


Figure 3.3: Conceptual Overview "Classifier", a Classifier consists of the three parts: an optional Precondition, a Criterion and an arbitrary number of Actions. Every part contains a textual constraint expression, called "LanguageEntity".

As illustrated in Figure 3.3, Classifiers consist of three parts: (1) An optional Precondition, (2) a mandatory Criterion and (3) zero or more Actions. Every one of these parts contain one LanguageEntity that represents a textual constraint expression. Furthermore, the Tool Smith can provide a Classifier as a dependency, that needs to evaluate to true prior to executing Actions. This dependency can be used to ensure that the current object is in the right state to execute the Action. The Precondition and the Criterion can be written in EOL, OCL or JAVA, the Action can be written in EOL or JAVA in the current prototypical implementation.

In the CDL framework the textual constraint expression is either a statement implemented inline as a text, or the expression links to a file written in the corresponding constraint language and the name of the invariant. In that way existing source-code files can be reused. An exemplary implementation for property 12 of Table 2.2, can be found in

Listing 3.2.

In the following subsection the different subparts, Precondition, Criterion and Action of the Classifier are explained.

```

1 classifier : AssociatedWithSafetyGoal{
2   model: RecuperationModel
3   description: 'Checks if the HazardousEvent is associated with a
4     SafetyGoal'
5   severity: ERROR
6
7   precondition: {
8     language: OCL
9     file: 'platform:/resource/RecuperationUseCase/ocl/constraints.ocl'
10    invariant: 'AssociatedPrecondition'
11  }
12  criterion: {
13    language: OCL
14    condition:
15      'inv: dependability::safetyrequirement::SafetyGoal.allInstances()
16        ->exists(goal : $PACKAGE_REQUIREMENTS::SafetyGoal | goal.derivedFrom
17          ->exists(event : dependability::HazardousEvent | event = self))'
18  }
19  action: {
20    name: 'Select SafetyGoal'
21    description: 'Selects a SafetyGoal from the list of possible
22      SafetyGoals' {
23      language: EOL
24      condition: 'do {
25        var safety_goals = SafetyGoal.allInstances().collect(
26          goal : SafetyGoal | goal.name);
27
28        var name = System.user.choose(
29          "Which SafetyGoal should be associated?", safety_goals);
30        var goal = SafetyGoal.allInstances().select(
31          goal : SafetyGoal | goal.name = name).first();
32        goal.derivedFrom.add(self);
33      }'
34  }
35  }
36  }

```

Listing 3.2: Example implementation of property 12 in Table 2.2 with Precondition, Criterion and Action

Preconditions

When applied to an element in the domain instance model, a Classifier can yield to three different results: (1) true, (2) false and (3) inconclusive. True and false are the results of the Criterion, inconclusive indicates that the Precondition did not hold. Due to that mechanism, the Tool Smith can use the Criterion to express the core constraint of the Classifier, ensuring that the element is already in the correct state. Furthermore, the usage of "inconclusive" adds the expressiveness to state that the element is in the wrong state instead of stating that the constraint does not hold. Thus, the usage of Preconditions enables the framework to deal with temporal inconsistencies correctly.

Consider the previously mentioned example: in one instance of "HazardousEvent" the attribute "ASIL" is not set. There are two potential outcomes: the Classifier could evaluate to false or the Classifier could evaluate to inconclusive since the element is not in the correct state. Through the use of the Precondition, the Tool Smith can implement the outcome that fits best in situations like these. As a result, the use of Preconditions results in cleaner Criterion code and less Action code.

Additionally, the Precondition plays an important role when grouping Classifiers logically into Constraint Groups. If the Classifiers are assigned using the implies- or or-conjunction the statement of the Constraint Group would be impaired if the Criterion also has to check the state of the element. With the use of Preconditions the Constraint Group is not evaluated any further and the information is presented to the Modeller, once an inconclusive result is detected.

Constraint Criterion

The Criterion is the determining entity of the Classifier. In the Criterion, the constraint the Classifier expresses, should be implemented. The Criterion should not check if the state of the element is correct, this should be done in the Precondition. It should also be noted, that the Criterion is the logical expression that will be evaluated when the Classifier is logically assigned in a Constraint Group, the Precondition is a needed prerequisite, but does not influence the *logical* outcome of a Constraint Group.

Within a Constraint Group, the Criterion is evaluated using short-circuit evaluation, resulting in less computational steps, once a Criterion does not hold. That means that Actions are only displayed for the *first* Classifier within a Constraint Group that contributes to a false result.

Actions

Once the Criterion (in a single Classifier or inside a Constraint Group) evaluates to false, the Modeller can choose from a list of given Actions. These Actions are provided by the Tool Smith and should support the Modeller in changing the domain instance model in such a way, that the Criterion holds again. Furthermore, the framework provides the possibility of defining Actions as "autoaction", meaning that the expression embedded in the Action is executed right after the Criterion evaluates to false (either as single Classifier or when the Criterion contributes to a Constraint Group evaluating to false).

Due to the use of the extensible languages JAVA and EOL, the framework provides flexibility to the Tool Smith. Eligible Actions can facilitate GUI-elements, information from outside the model, or elements from within the tooling environment (for example opening help to support the Modeller).

Actions are executed within their own EditingDomain inside the CDL framework. The purpose of an EditingDomain² is to track changes on the model, allowing the Modeller to undo Actions. The EditingDomain is destroyed and newly created every time the Modeller applies the constraints to the domain instance model. That means, that the Modeller can undo all Actions until he checks the domain instance model again.

²<http://download.eclipse.org/modeling/emf/emf/javadoc/2.4.3/org/eclipse/emf/edit/domain/EditingDomain.html>

3.3.3 Macro Engine

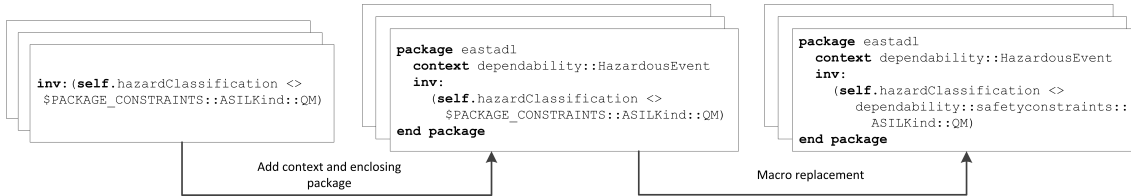


Figure 3.4: Schematic overview of the Macro Engine, the three different listings show the information added by the Macro Engine.

The Macro Engine serves two purposes, as illustrated in Figure 3.4: (1) append the enclosing context and package to constraint expression and (2) replace macros with their given values.

Certain constraint languages, for example OCL and EOL, need the enclosing package and context to define the constraints. Therefore, when defining constraints they are always valid for just one given context. In order to let the same constraint be reused in other contexts, it has to be context-independent formulated. The example provided in Figure 3.4 shows that the first expression is defined context-independent. When executed, this expression is enhanced with the assigned context and enclosing package in order to be a valid expression for the corresponding constraint language interpreter.

Furthermore, the framework provides the possibility of creating textual macros. Macros are textual labels starting with a `$`-sign and are replaced with the corresponding value before the constraint expression is compiled in the interpreter. They are defined by the Tool Smith in the replacement section. Listing 3.3 shows such a definition of a macro.

```

1 replacement :
2 {
3   variable : $PACKAGE_CONSTRAINTS text : 'dependability::safetyconstraints'
4   variable : $PACKAGE_REQUIREMENTS text : 'dependability::safetyrequirement'
5 }

```

Listing 3.3: Sample replacement definition

Moreover, the framework provides two predefined macros, namely `$CONTEXT$` and `$PACKAGE$`. `$CONTEXT$` provides the name of the current EClass (as string) the Classifier is assigned to, `$PACKAGE$` is replaced by the enclosing package of the assigned EClass (encoded in the representation of the given constraint language). Table 3.1 shows exemplary replacements in different constraint languages .

3.4 Domain Expert View

A Domain Expert is responsible for creating a constraint set based on the Classifiers provided by the Tool Smith. The Classifiers can be combined into Constraint Groups using propositional logic. Finally, the Constraint Groups or single Classifiers are assigned to certain elements in the domain meta-model.

| Language | \$CONTEXT\$ | \$PACKAGE\$ |
|----------|-------------|----------------------------------|
| EOL | SafetyGoal | dependability::safetyrequirement |
| OCL | SafetyGoal | dependability::safetyrequirement |
| JAVA | SafetyGoal | dependability.safetyrequirement |

Table 3.1: Example replacements for the EAST-ADL2 element SafetyGoal in the different constraint languages

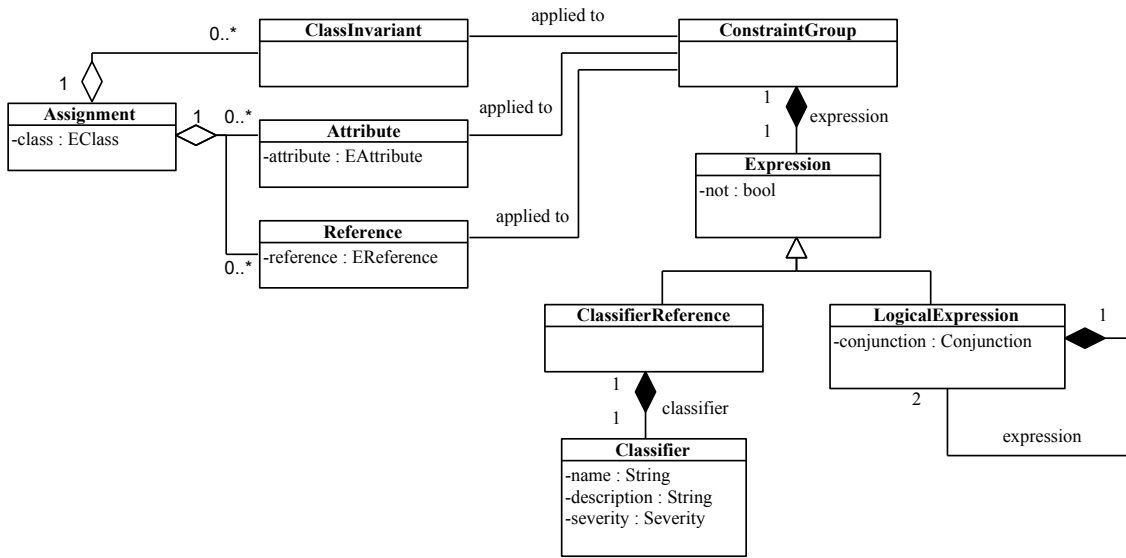


Figure 3.5: Simplified structural overview of Constraint Groups and Assignments

In the following section Constraint Groups and the assignments from Constraint Groups and / or single Classifier to elements from the domain meta-model are discussed.

3.4.1 Constraint Groups

A Constraint Group is a set of logically connected Classifiers. The Classifiers can be connected using conjunctions from propositional calculus (AND, OR, NOT, IMPLIES). A structural overview of Constraint Groups is illustrated in the right part of Figure 4.10. The concept of Constraint Groups supplements and enables reusable Classifiers. Classifiers should therefore be formulated in such a way, that they only cover the atomic information given by the constraint. These atomic Classifiers are then combined into Constraint Groups, allowing the construction of more complex constraints using Classifiers.

Another aspect novel to the concept of Constraint Groups is the possibility of utilising the advantages of the different constraint languages inside the Classifiers. Constraint Groups combine the different textual constraint expressions, potentially even when written in different languages together into one boolean expression.

The usage of Constraint Groups is also beneficial when when considering the usage of Preconditions in order to establish that the element is in the correct state. Using plain

OCL or EOL, an elaborate constraint is the conjunction of different boolean expressions resulting either in true or false. The use of Preconditions within the conjuncted Classifiers however also allows the Constraint Group to evaluate to inconclusive, resulting in less false negative messages for the Modeller. If one Classifier within a Constraint Group evaluated to inconclusive, a message is presented to the Modeller and the rest of the group is not evaluated anymore.

In order to demonstrate the concept of Constraint Groups, recall property 12 in Table 2.2: If the Classifier was implemented using plain OCL, the expression could be formulated as one single invariant. This invariant would only be applicable to one particular element, the HazardousEvent. However, if the Criterion is split into two Classifiers "ASILGreaterThenQM", "AssociatedWithSafetyGoal" that are connected using an implies-conjunction, both Classifiers can be reused in other contexts without duplicating constraint code. Furthermore, the constraint code in both Classifiers is easier to read and understand, since each Classifier contains just the atomic constraint concept.

Constraint Group Structure

As illustrated in Figure 4.10, CDL Constraint Groups are built in a composite pattern [GHJV95] fashion using the meta-class Expression and the derived ClassifierReference and LogicalExpression. A LogicalExpression contains two Expressions and a logical conjunction. The depth of the composite logical expression is not limited, allowing the Domain Expert to create elaborate Constraint Groups. The ClassifierReference represents a link to an existing Classifier.

In the textual representation, Constraint Groups are displayed using brackets to denote the coherence, in the graphical representation the Classifiers are represented by a rectangle and the conjunctions are represented by circles. Both representations have the same meaning. Figure 3.6 shows an example for the graphical representation, Listing 3.4 shows the equivalent textual representation.



Figure 3.6: Graphical representation of a Constraint Group

```

1 constraint group: CG_HazardousEvent
2 {
3   (ASILGreaterThanQM IMPLIES AssociatedWithSafetyGoal)
4 }

```

Listing 3.4: Textual representation of a Constraint Group

Evaluation of the Constraint Groups

The Constraint Groups are logically evaluated in a short-circuit fashion. That means that once the result of a Constraint Groups is known, the rest of the group is not evaluated anymore. Short-circuit evaluation is used for performance reasons since the rest of the expression does not contribute to the result of a logical expression and therefore does not need to be evaluated. One shortcoming of this approach is that it is possible for a Classifier to evaluate to inconclusive but this result would not be identified.

3.4.2 Assignments

The last activity in the role of the Domain Expert is to assign the defined Constraint Groups or Classifiers to elements of the domain meta-model. The constraints can be assigned to every EClass in the domain meta-model. Syntactically, there are three ways to assign them: (1) as an invariant (constraint covers the whole EClass), (2) to an attribute of the meta-class (Meta-element: EAttribute) and (3) to a reference of a meta-class (Meta-element: EReference). As illustrated in Figure 4.10, every category can have zero or more assignments to Classifiers or Constraint Groups.

This distinction is of an informal nature, the Tool Smith always creates Classifiers for an instance of a domain meta-class. The distinction should be used to assign the constraints to the element of the domain meta-class they actually constrain, in order to provide a better overview of which elements are constrained by which Constraint Group or Classifier. An Assignment represents a reference between a Classifier / Constraint Group and the assigned EClass.

Listing 3.5 shows the textual representation of an Assignment (element HazardousEvent from property 12, Table 2.2), whereas Figure 3.7 once again shows the graphical representation.

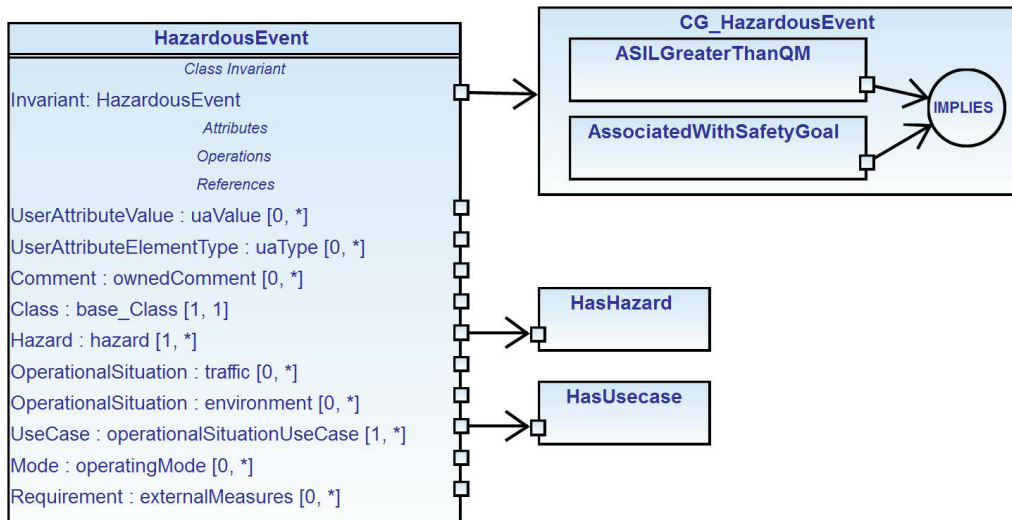


Figure 3.7: Graphical representation of the Assignment to the meta-class "HazardousEvent", domain meta-model EAST-ADL2 (property 12, Table 2.2)

```

1 assignment element: eastadl.dependability.HazardousEvent
2 {
3   invariant group: CG_HazardousEvent
4   reference: hazard classifier: HasHazard
5   reference: operationalSituationUseCase classifier: HasUsecase
6 }

```

Listing 3.5: Textual representation of the Assignment for the element "HazardousEvent" (property 12, Table 2.2)

3.5 Modeller View

The Modeller is responsible for creating a consistent domain instance model. With respect to the CDL framework, the main activity is to execute the constraints provided by the Domain Expert on his domain instance model. Once the constraints have been executed the CDL framework provides a view where all the elements are listed where the Criterion did not hold. The Modeller can now choose one of the Actions to repair the defects on his domain instance model.

The remainder of the section presents in detail the activity of executing constraints on the domain instance model.

3.5.1 Execution of the CDL Constraints

1. Read Assignments

In the first step, the CDL Execution Engine iterates through all defined Assignments in order to determine which Classifiers and elements from the domain instance model are needed to execute the constraint. The instances of the assigned EClasses are temporarily buffered.

2. Create Logical Expressions

In the second step, the composite structure of the Constraint Groups is created.

3. Compile textual constraint expressions with their Classifiers

In the third step all interpreters needed for the constraint languages are created. Moreover, all Classifiers are compiled and stored within their position in the logical expressions.

4. Execute Constraints Instance Objects

The next step is to iterate through all elements and execute the Constraint Groups and the Classifiers. If the Classifiers are combined using a Constraint Group the evaluation for the given element stops once the result of the Constraint Group is determined (short-circuit evaluation).

5. Present Result

If the Constraint Group or Classifier evaluates to false, the element and the corresponding constraint are presented to the Modeller in an Eclipse view (Problem View). An example of this view is show in Figure 3.8.

6. Choose Action

Once a constraint evaluates to false and the Tool Smith provides an Action for the Classifier, the Modeller can choose this Action from the Problem View. This Action is then executed and should either repair the defect or provide help to the Modeller to repair the defect on his own. An example selection of the Action provided is illustrated in Figure 3.8.

7. Undo Action

If the Modeller realises that the executed Action altered the model in an unwanted way, the CDL framework provides a dedicated view where all Actions are presented, allowing the Modeller to undo changes to the model.

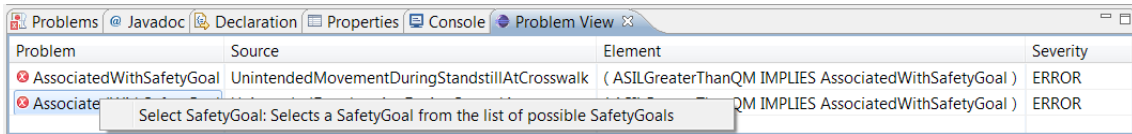


Figure 3.8: Screenshot of the CDL framework Problem View

All Actions are executed within an EditingDomain that is valid until the constraints are executed again on the domain instance model. That means that Actions can simply be undone until the Modeller checks his domain instance model again.

Whereas this section presents a conceptual overview of the activities needed to create and execute CDL constraints, Section 4.3 further describes their technical details.

3.6 Constraint Definition Language (CDL)

The core of the framework is a domain-specific language called "Constraint Definition Language" (CDL). The CDL serves as bridge between different constraint languages and the elements in the domain meta-model. The abstract syntax of CDL is illustrated in Figure 3.9. It is closely connected to the textual syntax since all elements from the meta-model map to the corresponding element in the textual representation (c.f. Section 4.4 for an overview of the CDL grammar and Appendix A for the complete CDL grammar).

The graphical representation covers a smaller subset of functions provided by the CDL meta-model, mainly the functionality needed by the Domain Expert (right hand side in Figure 3.9). This decision was made because the Tool Smith provides his functionality using the textual constraint languages. Thus, there is no need to create a graphical representation for that. The Domain Expert on the other hand should preferably just work with the graphical representation, since it is closer to the craft of modelling.

Table 3.2 shows the mapping between the functionality, roles and the elements in the CDL meta-model. The column "Graphical Notation" denotes whether there is a graphical representation for the given element.

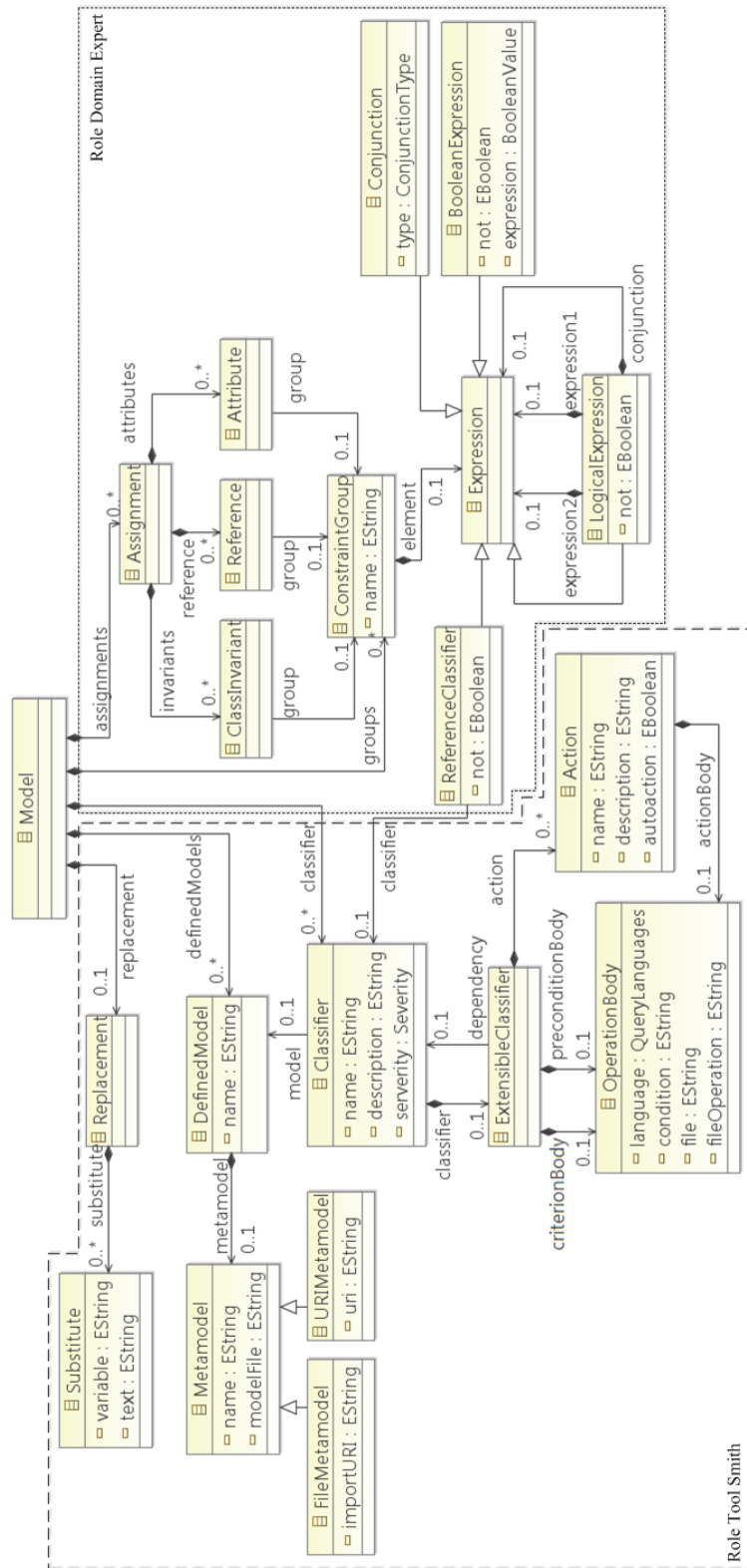


Figure 3.9: Simplified meta-model of the Constraint Definition Language (CDL), annotated with the different roles

| Functionality | Role | Elements in CDL Meta-Model | Graphical Notation |
|--------------------------|---------------|---|---------------------------|
| Import Models | Tool Smith | DefinedModel, FileMetamodel, URIMetamodel | No |
| Create Classifiers | Tool Smith | Classifier, ExtensibleClassifier, OperationBody, Action | No |
| Create Macros | Tool Smith | Substitute, Replacement | No |
| Create Constraint Groups | Domain Expert | ConstraintGroup, ReferenceClassifier, LogicalExpression, Conjunction, BooleanExpression | Yes |
| Assign Constraints | Domain Expert | Assignment, ClassInvariant, Reference, Attribute, ReferenceClassifier, ConstraintGroup | Yes |

Table 3.2: Elements of the CDL meta-model mapped to roles and functionalities

Chapter 4

CDL Framework Implementation

4.1 Introduction

This chapter describes the technical realization of the CDL framework. It describes *how* the functionality presented in Chapter 3 is implemented in the CDL framework. The CDL framework is implemented using three Eclipse plugins: (1) CDL Execution Engine to execute the constraint on the model, (2) Textual editor or Xtext editor to create the CDL instance model textually, and (3) Graphical editor or Graphiti editor to edit the CDL instance model graphically.

As depicted in Figure 4.1, the Xtext and the Graphiti editor take the domain meta-model as input to define the constraints for the domain instance model. The two synchronised editors manipulate one and the same CDL instance model. The CDL instance model and the domain instance model then serves as input to the CDL Execution Engine. The execution engine delegates the textual constraint expression to the corresponding interpreters, collects the result and presents it to the user. The user can execute assigned Actions and alter the domain instance model in this way.

The remainder of this chapter is structured as follows: Section 4.2 provides an overview of the different aspects, syntax and semantics of the Constraint Definition Language. Section 4.3 describes in detail the sequence of constraint execution on the domain instance model, Section 4.4 explains the role of the textual modelling framework Xtext and the grammar of the textual representation. Section 4.5 gives an overview of the graphical editor framework Graphiti and the implementation of the graphical representation using Graphiti and in Section 4.6 the synchronisation between the two representations is described.

4.2 Syntax & Semantics

When developing a domain-specific language, the model is the main artefact. However, the result of the development process is usually not the model, but an artefact derived from it (source code, documentation, etc.). In the case of the CDL framework, the purpose of the model is to make a statement on how well the domain-instance model obeys the specified constraints.

The distinction between the syntax and the semantics of the CDL in the framework is the following: the textual and the graphical editor manipulate the CDL instance model.

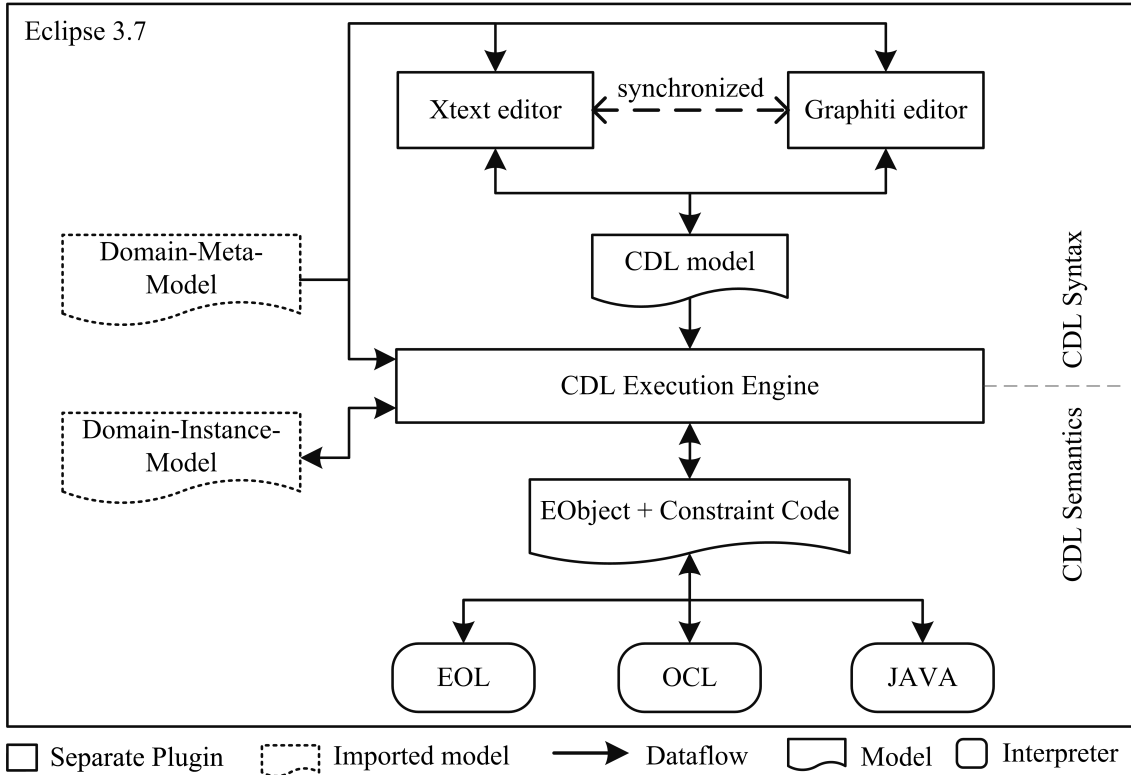


Figure 4.1: Workflow of defining and executing constraints

The CDL Execution Engine uses the CDL instance model as input and executes it on the domain instance model (semantics). In order to provide a better overview on how the different parts of the framework are conceptually connected, Table 4.1 provides an overview of the mapping between the syntax elements and their corresponding elements from the execution engine.

4.3 The CDL Execution Engine

The CDL Execution Engine is responsible for executing the CDL instance model on the domain instance model, and shows which elements do not fulfill the specified constraints. Furthermore, the execution engine executes the Actions in every case a constraint does not hold. It is the only plugin inside the CDL framework that alters the domain instance model.

4.3.1 GUI Elements

The CDL Execution Engine contributes a toolbar entry, as well as two views: the "Problem View" and the "Action Log View" to the Eclipse GUI. The toolbar entry contains the button to start the constraint execution on the domain instance model. In the "Problem View" all error-, warning- and information messages are presented. In case there is an

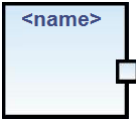
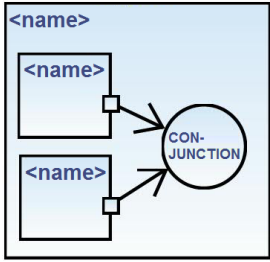
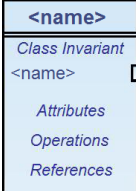
| Syntax Elements | | Semantics Elements |
|---|--------------------------------------|--|
| <i>Graphical Element</i> | <i>Textual Rule (Top-Level Rule)</i> | <i>Elements of the CDL Execution Engine</i> |
| | <u>DefinedModel</u> | <i>ResourceLoader</i> , responsible for reading the elements from the domain instance model |
| | <u>Replacement</u> | No direct counterpart, is used before creating the <i>ExecutableEntity</i> from the textual constraint expression |
|  | <u>Classifier</u> | <i>ClassifierExpressionExecutor</i> containing instances of <i>ExecutableEntity</i> (in the given constraint language) for every subpart (Precondition, Criterion and Actions) |
|  | <u>ConstraintGroup</u> | <i>LogicalExpressionExecutor</i> , arranged in a composite fashion |
|  | <u>Assignment</u> | No direct counterpart, in the class <i>Container</i> , entry class of the execution engine, all objects from the domain instance model are read, stored and the constraints are executed |

Table 4.1: Top-Level mapping from CDL syntax elements to their semantic counterparts

Action assigned in the Classifier, the Modeller can execute it from the context menu of the corresponding message. The "Action Log View" contains all Actions that were executed on the domain instance model since the constraints were executed the last time. To undo an Action, the Modeller calls the corresponding entry from the context menu in the "Action Log View".

4.3.2 Structural Overview

The application structure of the CDL Execution Engine, as illustrated in Figure 4.2, revolves around five main classes and interfaces: (1) *ExecutableEntity*, (2) *AbstractEntityFactory*, (3) *LanguageRegistry*, (4) *ExecuableExpression*, and (5) *ExpressionFactory*.

The abstract class *ExecutableEntity* provides the necessary methods to execute, save and undo one *LanguageEntity* (constraint code inside sub-elements of the Classifier) on the domain instance model. Furthermore, the class ensures that the concrete execute method

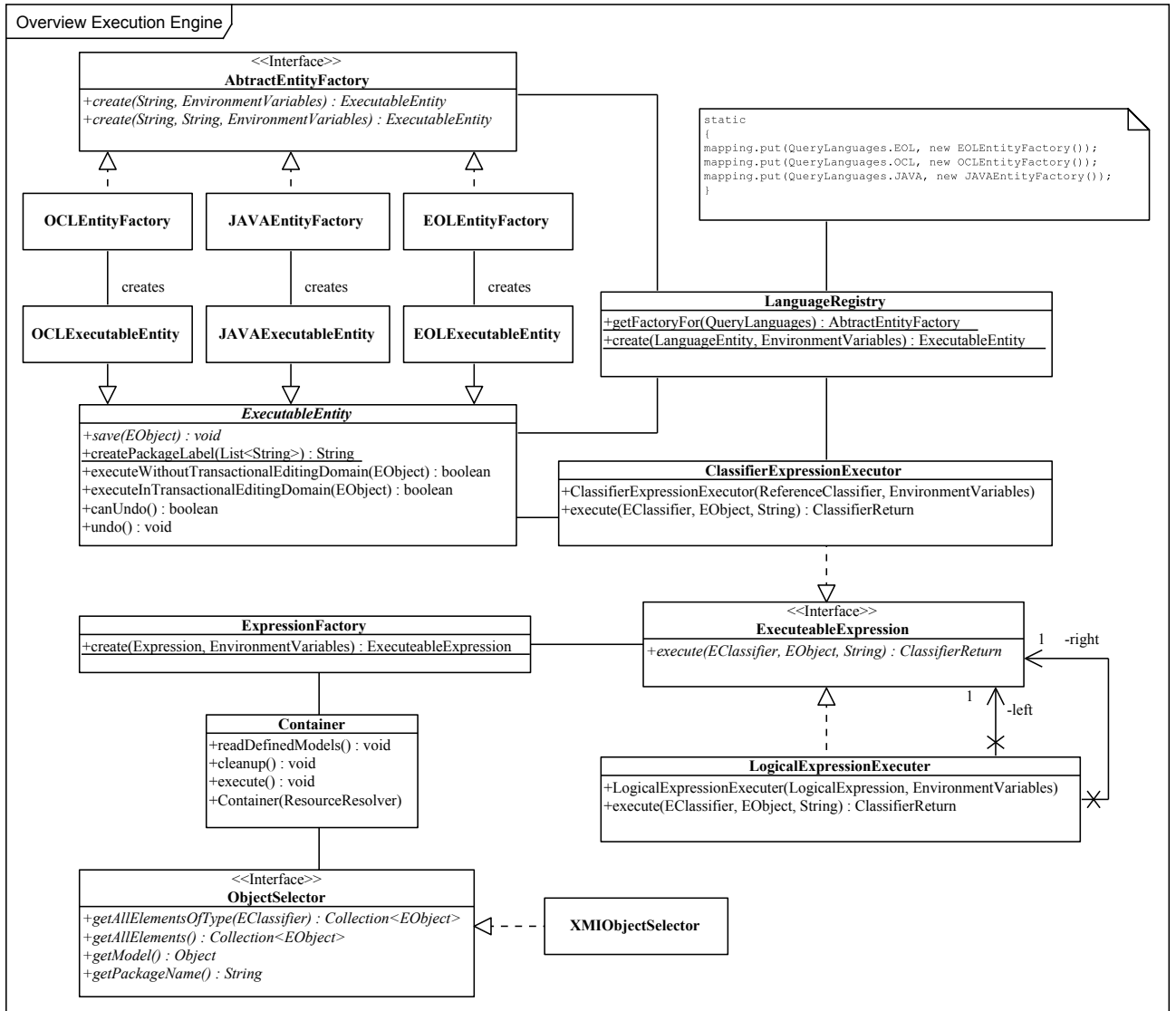


Figure 4.2: Structural overview of CDL Execution Engine

(implemented in the different constraint languages interpreter) is always called from within an *EditingDomain*. The creation of an *ExecutableEntity* is implemented in the concrete constraint language factories, derived from *AbstractEntityFactory*. Within these factories, all language-specific create functions are encapsulated. Moreover, these factories ensure the application of the replacement rules.

The remaining two classes, *ExecuableExpression* and *ExpressionFactory*, are used to create and execute the composite object tree (an example is illustrated in Figure 4.4) that represents the Constraint Groups.

4.3.3 Constraints Execution

The entry point of the execution engine is the class *Container* and its execute method. It is called when the button in the Eclipse toolbar is pressed. The process of executing the constraints is implemented as follows (refer to Section 3.5.1 for a conceptual overview and Figure 4.3 for an exemplary execution).

Note: In the following sequence a Constraint Groups is assigned. In case a single Classifier is assigned (step 5), the combination of the results is omitted.

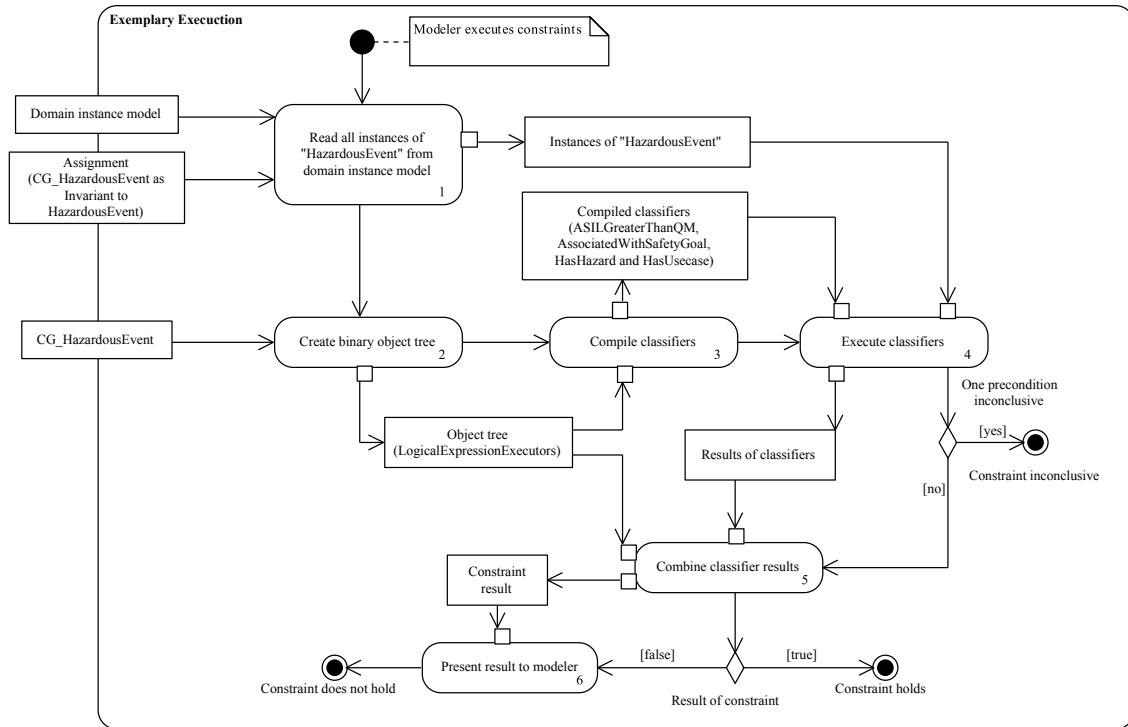


Figure 4.3: Exemplary execution of constraints with artefacts. For this example, properties 10, 11a and 12 from Table 2.2, are combined in one Constraint Group and assigned to the EClass "HazardousEvent".

1. Gather elements from the domain instance model:

To gather the objects from the domain instance model, first the type of the model elements (EClasses from the domain meta-model) have to be determined. Therefore, the Execution Engine iterates through all Assignments and gathers the instances of the associated types using the abstraction provided by *ObjectSelector*.

The result of this step is a map that associates the meta-class from the Assignment to its instances from the domain instance model.

2. Create Composite Object Structure:

The Classifiers are assigned either as a single Classifier, or as a Constraint Group. The logical expression inside the Constraint Group is syntactically represented as a binary

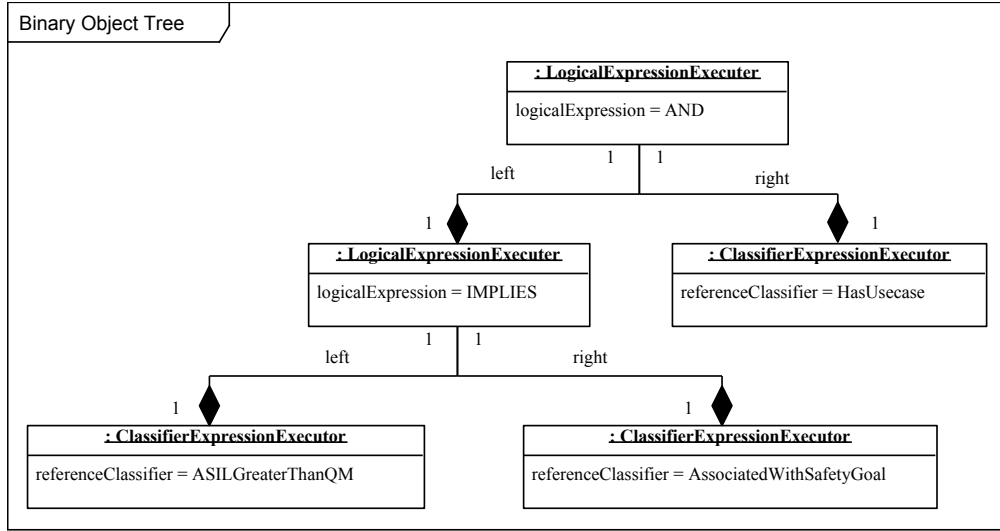


Figure 4.4: Composite object structure for the Constraint Group ((ASILGreaterThanQM IMPLIES AssociatedWithSafetyGoal) AND HasHazard) AND HasUsecase)

tree: the Classifiers are the leaves and the conjunctions are the inner nodes. Based on this composite tree, the corresponding object tree (illustrated in Figure 4.4) with instances of the abstract class *ExecutableExpression* is created for every Constraint Group. The creation of the actual Classifier is described in step three. Figure 4.2 shows the relationship between the classes, that are involved in the creation of the executable expressions.

The result of this step is a binary tree representing the arrangement of the Classifiers inside the Constraint Group.

3. Create ClassifierExpressionExecutors:

A Classifier is a concept from the CDL language syntax, the corresponding semantics element is an instance of the class *ClassifierExpressionExecutor*. To create it, the different instances of *LanguageEntity* from the sub-elements (Precondition, Criterion and Actions) of the *ClassifierExpressionExecutor*, and the surrounding environment is passed to the *LanguageRegistry*. The returned objects are stored within the *ClassifierExpressionExecutor*.

The result of this step are instances of the class *ClassifierExpressionExecutor* for every Classifier in the CDL instance model.

4. Execute the Classifiers on the elements:

Once the object object tree is created, the execute method of the root executor (*LogicalExpressionExecutor* or *ClassifierExpressionExecutor*) is called and propagated through the binary tree. Within the executors, first the execute-method of *ExecutableEntity* for the Precondition is called and, if true, the *ExecutableEntity* for the Criterion is evaluated. This is done for each instance of the given meta-class in the domain instance model (is in the example the meta-class "HazardousEvent", Figure 4.2). Every executor returns an object of the class *ClassifierReturn* that contains the result of the executor

(true, false or inconclusive) and holds references to the instances of *ExecutableEntity* that represent the Actions, if the Criterion in *ClassifierExpressionExecutor* evaluates to false.

The result of this step is an object of the class *ClassifierReturn* for every *ClassifierExpressionExecutor* involved in the Constraint Group.

5. **Combine results:**

The results of the executors are logically assigned, starting from the leaves (*ClassifierExpressionExecutor*) using the enclosing logical executors until the root element is reached. Thus, the *ClassifierReturn* for the root element holds the result of the whole Constraint Group for the given object of the domain instance model, as well as references to the instances of *ExecutableEntity* for the Actions if there are any.

The result of this step is the evaluation outcome for every Constraint Group.

6. **Present result to Modeller:**

If the root element executes to false (or inconclusive within a Constraint Group), a message containing the name of the Classifier, the name of the Constraint Group, the domain instance element and the severity is displayed in the "Problem View". Furthermore, the context menu is filled with instances of *ExecutableEntity* for every message.

7. **Apply Actions:**

Actions are accessible from the context menu in the "Problem View" (Figure 3.8). The corresponding Action is created when the Criterion inside the *ClassifierExpressionExecutor* evaluates to false. When executed, every Action is wrapped into a *RecordingCommand*¹, an element from the EMF Model Transaction Workbench. The *RecordingCommand* combined with an *EditingDomain* provides the functionality to undo changes to the model. Once an Action is applied, a corresponding entry is created in the "Action Log View" enabling the Modeller to undo the Action. The log view is cleared every time the Modeller executes the constraints, because the model is loaded again and otherwise the references inside the log view would no longer be valid.

4.3.4 Constraint Language Plugins

The CDL framework connects the domain models with the corresponding constraint code. The different plugins for the constraint languages are therefore responsible for compiling and executing the constraints on the domain instance model. In the prototypical implementation of the framework, three constraint languages are supported: EOL, OCL and JAVA. OCL and the EOL are both supported by open source project plugins, the JAVA compiler is part of the JAVA Software Developer Kit (SDK).

The OCL plugin (*org.eclipse.ocl*) provides an easy-to-use interface for compiling the constraint code and executes it directly in the given EMF representation of the domain instance model. The EOL plugin on the other hand, uses a different internal representation of the data, and therefore needs a mapping between the EMF data and the interfaces used by the EOL interpreter. EOL provides such a mapping (*InMemoryEmfModel*) that

¹<http://download.eclipse.org/modeling/emf/transAction/javadoc/workspace/1.5.0/org/eclipse/emf/transAction/RecordingCommand.html>

was used in the implementation of the CDL framework. Constraints written in JAVA are compiled using the the JAVA SDK compiler and operate directly on the EMF data structures. The Tool Smith has to ensure that all required libraries are on the classpath.

The parsing capabilities of the different interpreters are further used in the textual editor to check the syntax of the textual constraint expressions. The integration of the syntax checks into the Xtext framework is explained in Section 4.4.3.

4.4 CDL Textual Representation

Xtext is a framework that supports textual modelling: based on an Extended Backus-Naur Form (EBNF) grammar, a text editor is generated that supports certain comfort features, such as auto-completion or validation of the syntax. Editors generated by Xtext are deployed using an Eclipse plugin. Compared to other editor frameworks, Xtext provides the unique feature to generate an Ecore meta-model based from the EBNF grammar, the instance model is created using the abstract syntax tree (AST) of the text. The input text is transformed to an instance of this meta-model. Thus, the framework allows the developer to use the source code created by such an editor as an instance of the created meta-model.

4.4.1 Overview Xtext Framework

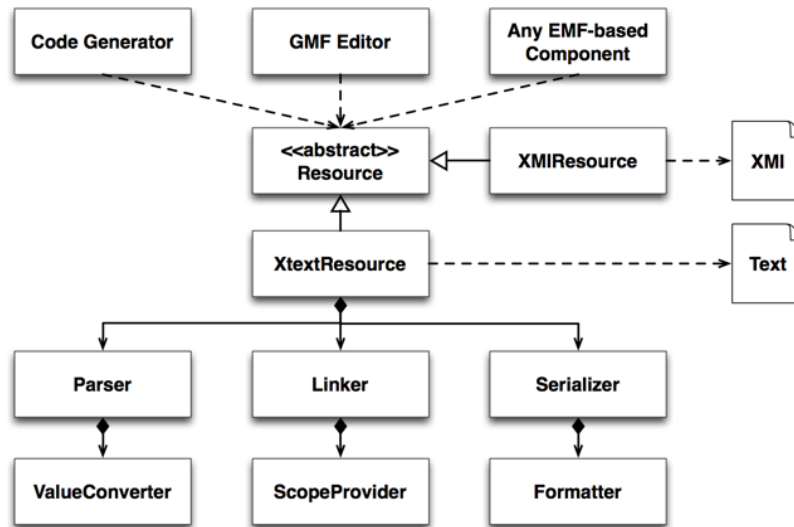


Figure 4.5: Overview Xtext of Framework²

Editors created by Xtext are highly customisable: the framework allows the importation of other Ecore based meta-models and processes them as part of the language. Furthermore, it is possible to adapt the validation of the input text, as used to check the syntax of textual constraint expressions that are declared inline (refer to Section 4.4.3). Moreover,

²<http://www.eclipse.org/Xtext/documentation/>

the framework provides the functionality to limit the scope of elements in the language, as well as the possibility of customising content assists. An example where that is necessary, is the importation of built-in Eclipse meta-models (such as Ecore or UML), that are referred to with their corresponding URIs. The contents of such meta-models are not included automatically by the Xtext linker, therefore the content is provided using the content assists.

The Xtext framework is built using several tools from the Eclipse Modelling Framework, such as the Eclipse Workflow Engine (MWE) or a built in parser generator (though the usage of antlr³ is recommended but due to incompatible licences not included). The usage of EMF technologies enables the integration of other projects, such as EMF Validation or the Graphical Modelling Framework (GMF).

4.4.2 CDL Grammar

Defining the EBNF grammar is the starting point for creating a textual editor using Xtext. The complete CDL language is provided in Appendix A, in this section an overview of the most important rules is given. The CDL meta-model is generated from the EBNF grammar. Thus, all rules can be directly mapped to elements of the CDL meta-model.

Note: The following chapter and the appendix use a syntax tree notation created by Xtext. On the left side, the name of the rule is given, literals are denoted with a white background, whereas rules are denoted using a grey background. Within the running text rules are underlined.

Top-Level Grammar Elements

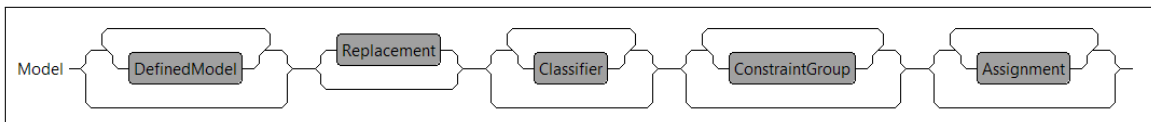


Figure 4.6: Top-level elements of the CDL grammar

The top-level element in the grammar is the rule Model, as illustrated in Figure 4.6, and contains five main rules: (1) DefinedModel to import the domain meta- and instance models, (2) Replacement to define macros, (3) Classifier to define Classifiers and embed or refer constraint code, (4) ConstraintGroup to logically assign the Classifiers and (5) Assignment to connect the Constraint Groups or Classifiers to the domain meta-model.

Import Models Grammar Elements

In order to import an existing meta-model, the following rules are needed: DefinedModel, MetaModel and the derived rules FileMetaModel and UriMetamodel. In Figure 4.7 the rules to import a Ecore file are shown. In the rule FileMetaModel, the content of "metamodel file" is assigned to a special variable in the Xtext framework, namely importURI. This allocation causes the framework to import the content of the file (supposed to be an Ecore

³<http://www.antlr.org/>

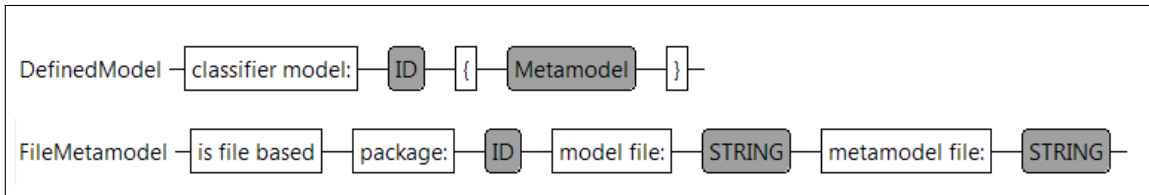


Figure 4.7: Syntax elements to import a given Ecore based meta-model

file) and provide the elements as references. In the case of an URI-based meta-model this resolution is done using an implemented routine in the content assist functionality.

Macro Definitions Grammar Elements

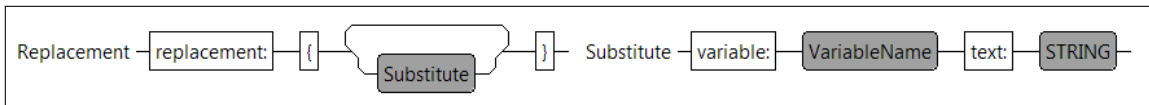


Figure 4.8: Syntax elements used for macro definitions

To provide textual macros three rules are needed, as illustrated in Figure 4.8: Replacement, Substitute and VariableName. Replacement holds a list of Substitute rules containing a mapping from a VariableName (a string starting with \$) to an arbitrary text.

Classifiers Grammar Elements

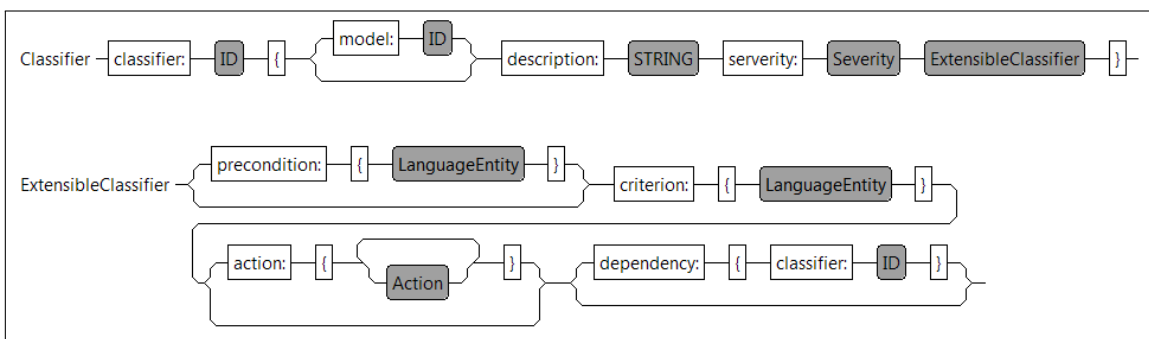


Figure 4.9: Syntax elements used to define a Classifier

Figure 4.9 shows an overview on the syntactical elements needed to define a Classifier. Besides an unique name, a Classifier can hold a reference to a defined model in order to provide syntactical support for the Domain Expert. As illustrated in the syntax graph, the rule ExtensibleClassifier shows that both the Precondition and the Criterion, contain a LanguageEntity that represents a constraint code implemented inline or a reference to

a file with the constraint code. The rule Action, on the other hand, defines a name, a description, and the field "autoaction" in addition to the LanguageEntity, therefore it is implemented using a different rule (c.f. Appendix A for all syntax elements).

Constraint Groups Grammar Elements

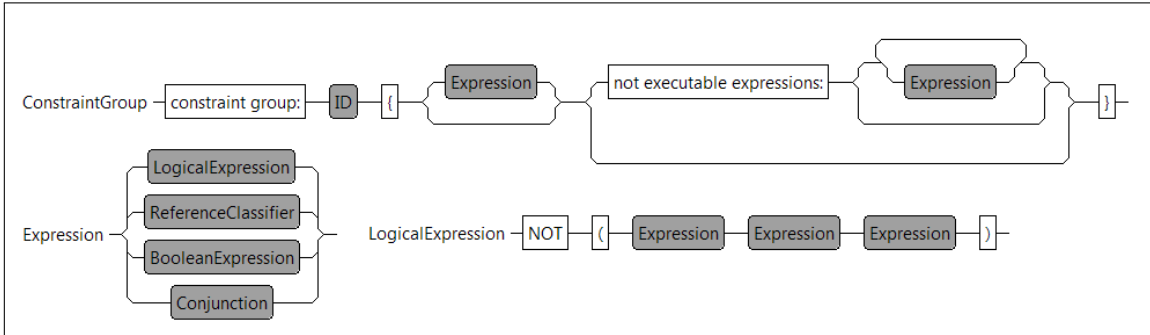


Figure 4.10: Syntax elements used to define a Constraint Group

As illustrated in Figure 4.10, the syntactical element ConstraintGroup is the combination of three Expressions, whereas a Conjunction is also an expression. The check as to whether the Expression is a Conjunction is done in the validation routine. Moreover, since LogicalExpression follows Expression, the syntax allows the Domain Expert to nest an arbitrary number of logical expression within an Expression.

The list titled "not executable expressions" allows the storage of Classifiers from the graphical representation that are not yet connected and are neither needed in the meta-model nor the textual representation. This enables the domain-expert to store a draft (unconnected), and therefore not executable, Constraint Group in the CDL instance model.

Assignments Grammar Elements

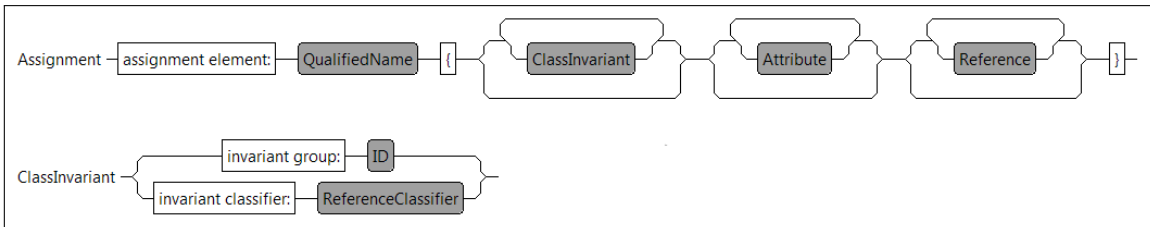


Figure 4.11: Syntax elements used to define assignments

The last element in this syntax description is the rule Assignment (illustrated in Figure 4.11). The Assignment connects an element from the domain meta-model (referred to as a QualifiedName in "assignment element") through one of its sub elements (ClassInvariant, Attribute or Reference) to a Classifier or a Constraint Group, shown in the example for a ClassInvariant in the syntax tree.

4.4.3 Validation

Validation within the Xtext framework is understood to be a syntactical check of the text inside the editor. Checks ensuring that the input text conforms to the CDL meta-model come as a side product to the generated EBNF parser. Furthermore, the framework allows the customisation of the validation routine enabling the developer to create checks that go further than the conformance relationship. The interface provided by Xtext is easy to use and only requires the addition of the annotation "@Check" to a method that receives an element of the CDL meta-model as a parameter.

Within the CDL textual editor, these checks are used to syntactically check the inline implemented constraint code. The implemented constraint code is passed to the corresponding interpreters and is parsed inside the interpreters. In the prototypical implementation the checks are hard coded for each constraint language, in a further release the check should be a member of the class *ExecutableEntity*. If the syntax check fails, a message and a marker is presented to the Tool Smith, as shown in Figure 4.12. If the Tool Smith uses context-dependent macros that cannot be resolved because the Classifier has not been assigned yet, the syntax checker shows a warning indicating that the constraint code cannot be checked.

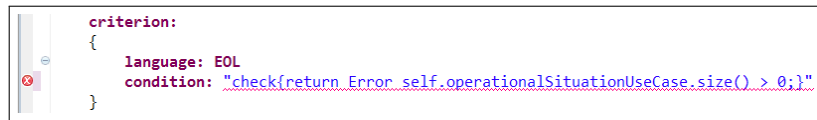


Figure 4.12: Example indication of a syntax error

4.5 Graphical Representation of the CDL

In the CDL framework, Graphiti is used to implement the graphical representation of the Constraint Definition Language. Eclipse Graphiti is a framework that enables the creation of graphical editors based on EMF models. Graphiti is built on top of the Graphical Editing Framework⁴ (GEF) and its subproject Draw2D⁵. The main objective of Graphiti is to ease the development of graphical editors due to the high complexity of GEF. In order to achieve its objective, Graphiti provides several helper-classes that ease the creation of elements and interfaces to add functionality to the editor.

4.5.1 Graphiti Framework Overview

Graphiti Data Concept

As illustrated in Figure 4.13, Graphiti connects the elements from the CDL instance model to a graphical representation using a two-layered structure: Pictogram Elements serve as hierarchical connectors between the business objects and their graphical representation,

⁴<http://www.eclipse.org/gef/>

⁵<http://www.eclipse.org/gef/draw2d/>

⁷<http://www.slideshare.net/michaelwenz/short-talk-on-graphiti-at-eclipsecon-2010>

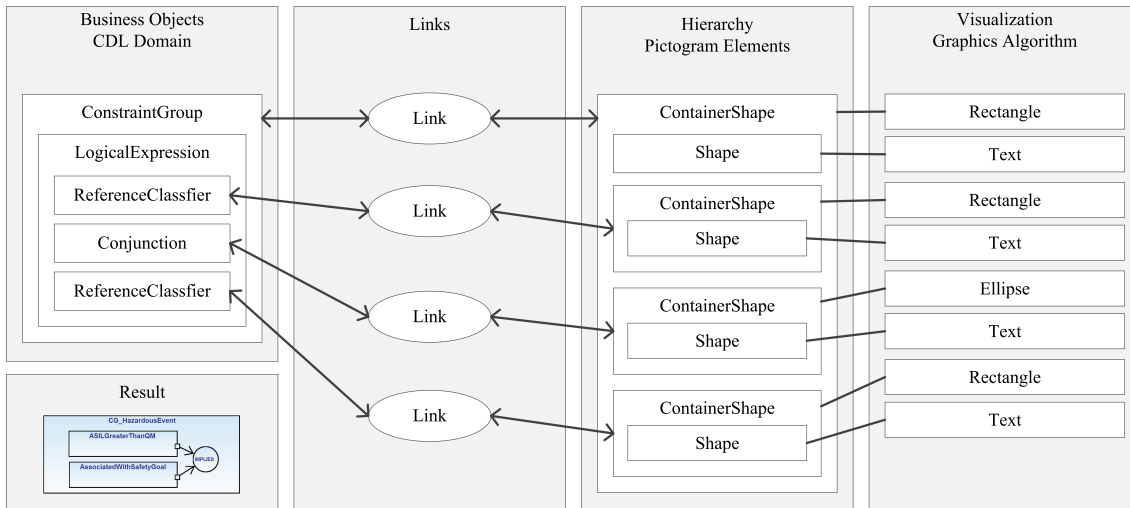


Figure 4.13: Overview of Graphiti data structure, an example of the business object "Constraint Group" (image based on Graphiti documentation⁷).

called Graphical Algorithms in Graphiti. Hierarchical means in that context, that there is an element of the type *ContainerShape* that holds a list of all its sub-elements (Shapes for graphical elements, Anchors for connectors or another *ContainerShapes* for subgroups).

The framework persists the diagrams as instances of the Graphiti meta-model, business objects are implemented as references to the CDL instance model.

Structural Overview

The implementation of the CDL graphical editor follows the example implementation in the tutorial⁸. Every feature of a graphical element (create, add, delete, remove, update, move, layout, resize, direct edit, custom feature, and create connection) is implemented in a separate class. In the framework, a custom feature is one that does not fall into the other categories, in the CDL framework a custom feature is used to, for example auto-layout certain elements in the graphical editor. As illustrated in Figure 4.14, the features are managed by the class *CDLFeatureProvider*. The class has a method for every possible feature of the graphical editor and returns the corresponding instance of the feature, based on the business object provided in the parameter (member of the different contexts). In case there is no implementation of the feature, the provider returns a default implementation.

Furthermore, Graphiti allows to customise the tool palette and the context menus that are provided for the graphical elements. The functionality is implemented in the class *CDLToolingBehaviourProvider* and is also aligned with the exemplary implementation from the Graphiti tutorial. The CDL framework customises the default behaviour by adding all EClasses from the domain meta-model in order to add them as Assignments. Moreover, the tooling palette contains all Classifiers and Constraint Groups from the CDL instance model in order to define Constraints Groups graphically. Figure 4.15 shows an example of the tooling palette using the EAST-ADL2 meta-model.

⁸<http://www.eclipse.org/graphiti/documentation/>

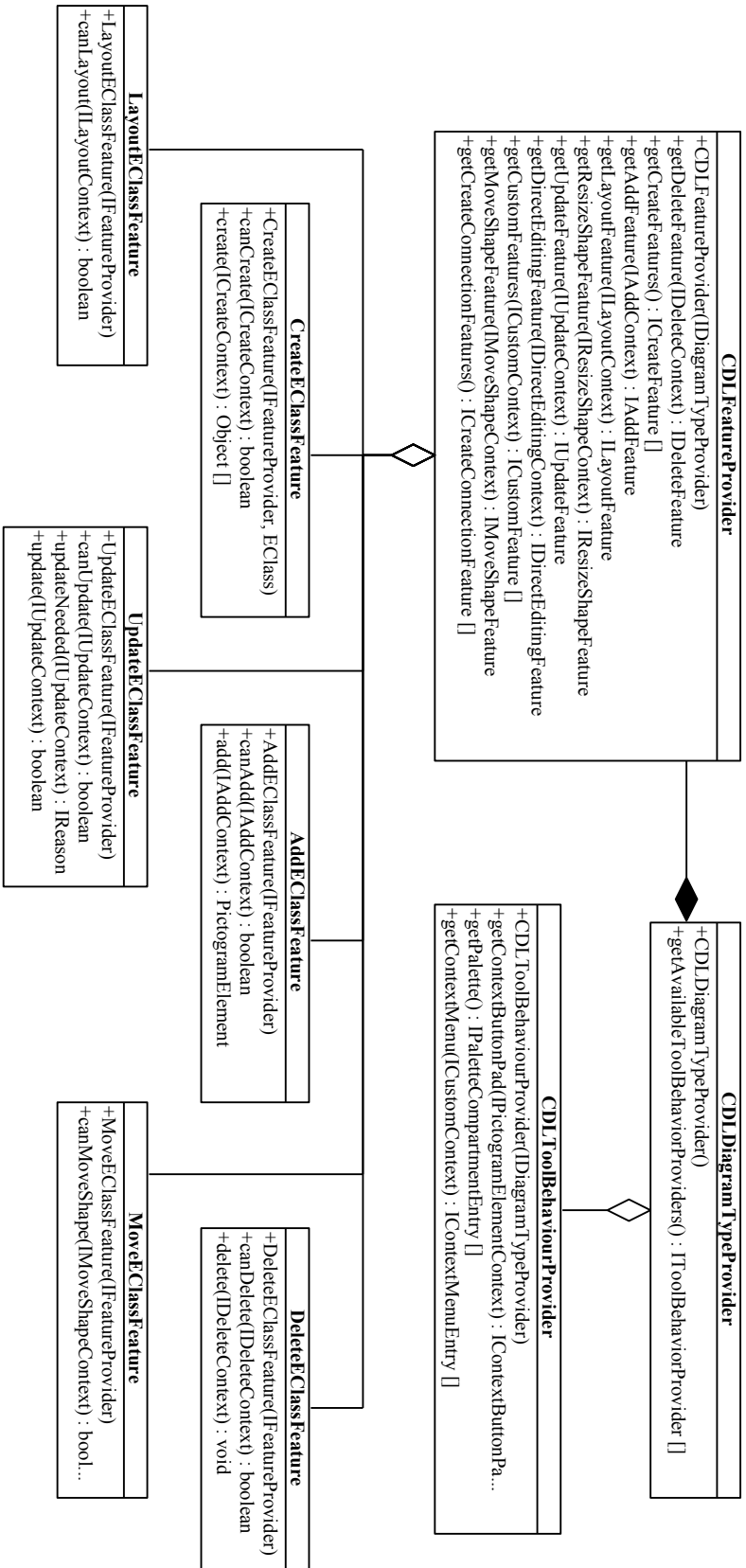


Figure 4.14: Overview of the Graphiti implementation structure (elements for feature ‘Assignment’ shown, others omitted)

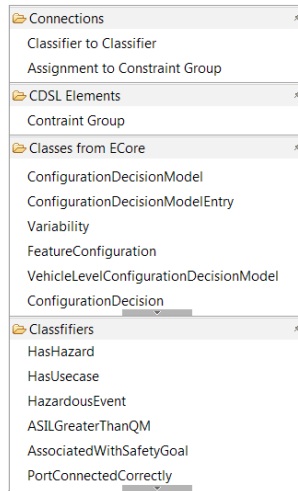


Figure 4.15: Tooling palette with constraints for the EAST-ADL2 meta-model

4.5.2 Auto Layout

The graphical editor makes it possible to automatically lay out the elements within the diagram. This feature supports the Domain Expert in laying out the diagram, especially in situations where elements are automatically inserted due to synchronization (c.f. Section 4.6).

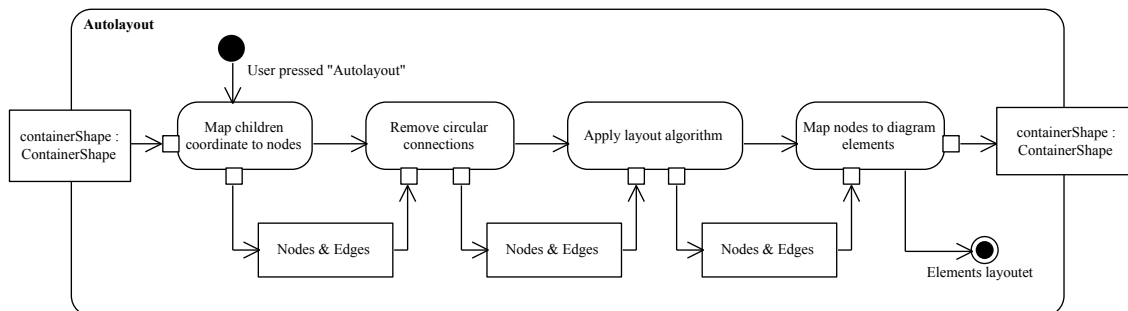


Figure 4.16: Auto-layout of diagram elements

Figure 4.16 presents the sequence of the auto layout feature: first, the CDL framework transforms the coordinates of the diagram elements (children elements of the given container shape) into generic Nodes and Edges (part of the EMF framework). In the next step, circular references are deleted, since most of the layout algorithms do not work properly otherwise. Based on this generic representation, all available layout algorithms (e.g. from ZEST⁹) can be applied. The prototypical CDL framework uses a directed-graph algorithm as the default algorithm. In the last step, the coordinates of the generic representation are transferred back to the diagram elements.

⁹<http://www.eclipse.org/gef/zest/>

In the CDL framework, the auto layout function can be applied to the whole diagram or to a single Constraint Group. Figure 4.17 shows an example of where a Constraint Group was updated by the synchronising feature and afterwards the directed graph algorithm is applied.

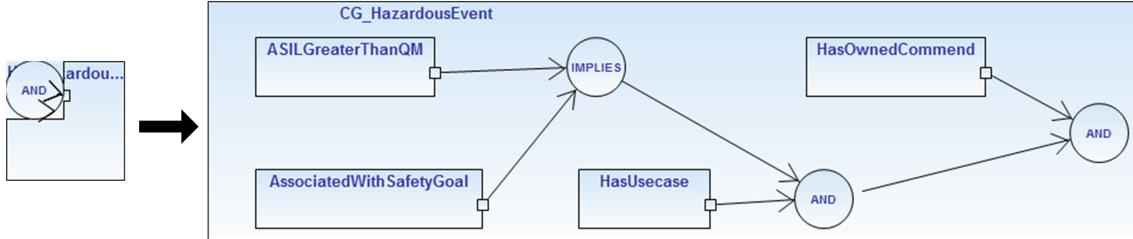


Figure 4.17: Autolayout of a Constraint Group

4.6 Synchronisation

When providing several views of the same model, it is crucial that all views provide an up-to-date view. To achieve that, the textual and the graphical editors are synchronised through the usage of the same CDL instance model. The Xtext editor is synchronised with the CDL instance model (built-in feature), whereas the graphical editors needs to be updated every time the editor is activated (gets the focus) inside the Eclipse platform.

The update-sequence consists of the following steps: iterate through all graphical elements and check if the business objects are still valid. If the business object has been deleted, the graphical element has to be deleted, since the textual representation is up-to-date. The next step is to iterate through all elements in the CDL instance model and check if the graphical representation (labels, sub-elements, connections, etc.) are still up-to-date. In cases where the element has been updated in the textual representation, the Graphiti editor indicates that through a red, dashed border around the element. The Domain Expert then has the option to update the element by pressing a button in the corresponding context menu. Then, the content from the textual representation is read and the graphical element is updated accordingly. As a last step, the Domain Expert has the option to auto layout the elements (c.f. Section 4.5.2) that were updated to enhance the readability of the diagram. Figure 4.18 shows such a graphical element that needs to be updated.

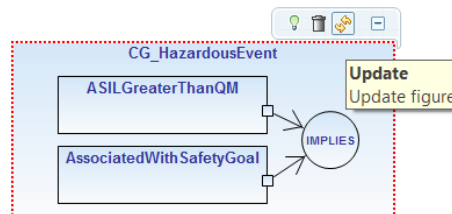


Figure 4.18: Example of a graphical element that needs to be updated

The diagram is updated *every time* the graphical editor gets the focus. In the prototypical implementation, the Domain Expert does not have to actively initialize the synchronization. An exception to this rule is unsaved changes inside one of the editors. In that case a message is shown to the Domain Expert, allowing him to decide which unsaved changes to keep.

Chapter 5

Case Study

In order to demonstrate the capabilities of the CDL framework, a set of constraints from the automotive domain was implemented. As briefly described in Section 2.7, the constraints were defined in [MGL⁺11] and [MAL⁺11] by Mader *et. al.* The meta-model used throughout their work is the EAST-ADL2 [EAS10] meta-model, provided as a plug-in to the Papyrus UML editor.

The remainder of this chapter is structured as follows: In Section 5.1 the domain meta-model of EAST-ADL2 is briefly introduced, Section 5.2 presents the list of properties that were implemented in the case study. Section 5.3 shows implementation of one illustrative Classifier and Section 5.4 presents the results of the case study.

5.1 Modelling in EAST-ADL2

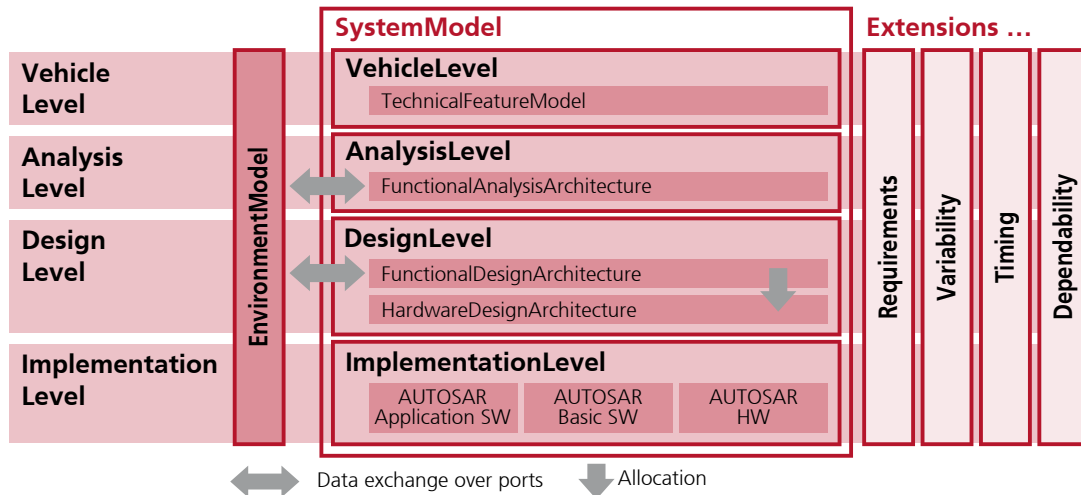


Figure 5.1: Top-Level structure of EAST-ADL2¹

¹<http://www.atesst.org/>

EAST-ADL2 is a domain-specific language to model embedded systems within the automotive domain. The language focuses solely on structural / architectural aspects and does not cover behavioural aspects. Depicted in Figure 5.1, EAST-ADL is organized into four layers, representing the different abstraction levels within the development process.

The domain-specific language aims to support the whole development process by enabling traceability between the different artefacts on the different abstraction levels. At the implementation level, EAST-ADL2 is strongly aligned with the AUTOSAR² standard and also connected with MATLAB/Simulink.

5.2 Implemented Properties

Throughout the work of Mader *et al.*, the EAST-ADL2 meta-model was used as a foundation to carry out a functional safety analysis, according to ISO26262 [ISO09] and derive the required work products. The system under development is a hybrid power train. To accomplish this, the meta-model was enhanced with safety-relevant stereotypes. Along with the safety analysis, certain properties were identified, that do not ensure the correct application of the safety analysis, but *indicate* that all the static characteristics (semantical and syntactical consistency) are met.

The list of implemented properties and the proposed Actions in the scope of the CDL framework are presented in Table 5.1. A detailed description of the involved Classifiers, can be found in Report [Kra12].

| ID | Meta Class | Property Definition | Proposed Action |
|----|----------------|--|--|
| 0 | Item | A complementary description has been defined | Query Modeller to input a new description for the element or set a default description |
| 0a | Item | At least one VehicleFeature has been defined | Choose one existing VehicleFeature and assign it to the Item |
| 1 | Item | At least one Hazard has been identified | Choose one existing Hazard and connect it to the Item |
| 2 | Item | At least one FeatureFlaw has been identified | Choose one existing FeatureFlaw assign it to the Item |
| 0b | VehicleFeature | A complementary description has been defined | Query Modeller to input a new description for the element or set a default description |
| 0c | VehicleFeature | Associated with at least one Item | Choose one existing Item and assign it to the VehicleFeature |
| 23 | VehicleFeature | A Requirement is satisfied | Assign the VehicleFeature to one given Requirement |
| 21 | Requirement | An ID is defined | Query the user to input a new ID |
| 22 | Requirement | A requirements text is defined | Query the Modeller to input a new text for the Requirement |

²<http://www.autosar.org>

| ID | Meta Class | Property Definition | Proposed Action |
|-----|----------------|---|--|
| 24 | Requirement | Is satisfied | No action, Modeller has to repair the property |
| 26 | Mode | A condition has been defined | Query the Modeller to input a new condition for the Mode |
| 3 | FeatureFlaw | At least one Hazard is identified | Choose one existing Hazard that will be assigned to the FeatureFlaw |
| 4 | FeatureFlaw | Associated with at least one Item | Choose one existing Item and assign it to the element |
| 5 | FeatureFlaw | A complementary description has been defined | Query Modeller to input a new description for the element or set a default description |
| 6 | Hazard | At least one FeatureFlaw is associated | Choose one existing FeatureFlaw and assign it to the Hazard |
| 7 | Hazard | At least one Item is associated | Choose one existing Item and assign it to the element |
| 8 | Hazard | At least one HazardousEvent has been identified | Choose one existing HazardousEvent and assign it to the Hazard |
| 9 | Hazard | A complementary description has been defined | Query Modeller to input a new description for the element or set a default description |
| 10 | HazardousEvent | At least one Hazard is associated | Choose one existing Hazard and assign it to the HazardousEvent |
| 11 | HazardousEvent | At least one UseCase is associated | Choose one existing UseCase and assign it to the HazardousEvent |
| 12a | HazardousEvent | At least one SafetyGoal is associated if ASIL greater than QM | Choose an existing SafetyGoal and assign it to the HazardousEvent |
| 13 | HazardousEvent | Associated with at least one OperationalSituation | Choose one existing OperationalSituation and assign it to the HazardousEvent |
| 14 | HazardousEvent | ASIL has been correctly derived from Controllability, Severity and Exposure | Set the ASIL of the HazardousEvent to the calculated value |
| 17 | HazardousEvent | Classification assumptions have been defined | Query the Modeller to input a new text for the classification of the HazardousEvent |
| 25 | HazardousEvent | Associated with at least one Mode | Chooses one existing Mode and assign it to the HazardousEvent |
| 15 | SafetyGoal | A HazardousEvent is associated | Choose one existing HazardousEvent and assign it to the SafetyGoal |
| 16 | SafetyGoal | A safe state is defined | Query the Modeller to input a new safe state for the SafetyGoal |

| ID | Meta Class | Property Definition | Proposed Action |
|-----|----------------------------|--|--|
| 18 | SafetyGoal | The ASIL has been correctly derived from associated HazardousEvents | Set the ASIL of SafetyGoal the to the derived value |
| 20 | OperationalSituation | A complementary description has been defined | Query Modeller to input a new description for the element or set a default description |
| 28 | SafetyGoal | At least one safety requirement is derived | Choose one existing Requirement and assign it to the SafetyGoal |
| 30 | QualityRequirement | Is allocated to at least one AnalysisFunctionPrototype, if it is a safety requirement | Choose one existing AnalysisFunctionPrototype and assign it to the QualityRequirement |
| 32 | Environment | An environmentModel is defined | Choose one existing FunctionPrototype and assign it to the Environment |
| 34 | AnalysisLevel | A functionalAnalysisArchitecture has been defined | Choose one existing FunctionalAnalysisArchitecture and assign it to the AnalysisLevel |
| 39 | FunctionFlowPort | Has at most one FunctionConnector to a FunctionFlowPort of type out or in-out associated, if type in | No action, Modeller has to repair the property |
| 40 | FunctionPort | A type is defined | Choose one existing EADatatype and assign it to the element |
| 40c | FunctionPort | A complementary description has been defined | Query Modeller to input a new description for the element or set a default description |
| 40b | FunctionConnector | Connector is connected to two FunctionPorts | Query Modeller to input a new description for the element or set a default description |
| 41 | AnalysisFunction-Prototype | A type is defined | Choose one existing EADatatype and assign it to the element |
| 42 | AnalysisFunction-Prototype | Has a complementary description | Query Modeller to input a new description for the element or set a default description |
| 42a | AnalysisFunction-Prototype | An ErrorModelPrototype is defined for every AnalysisFunction-Prototype | Choose one existing ErrorModelPrototype and assign it to the AnalysisFunctionPrototype |
| 42b | AnalysisFunction-Type | An ErrorModelType is defined for every Analysis-FunctionType | Choose one existing ErrorModelType and assign it to the Analysis-FunctionType |

| ID | Meta Class | Property Definition | Proposed Action |
|-----|-------------------------|--|--|
| 48 | FaultInPort | Has only one FaultFailure-PropagationLink to a FailureOutPort associated | No action, Modeller has to repair the property |
| 51 | FaultFailurePort | A functionTarget_path is defined | Choose one existing FunctionPrototype and assign it to the FaultFailurePort |
| 51a | FaultFailurePort | A type is defined | Choose one existing EADatatype and assign it to the element |
| 52a | FailureOutPort | Has a complementary description | Query Modeller to input a new description for the element or set a default description |
| 53 | ErrorModel-Prototype | A type is defined | Choose one existing EADatatype and assign it to the element |
| 54 | ErrorModel-Prototype | A functionTarget is defined | Choose one existing FunctionPrototype and assign it to the ErrorModelPrototype |
| 55 | ErrorBehavior | An externalFailure is defined | Choose one existing FailureOutPort and assign it to the ErrorBehavior |
| 57 | ErrorBehavior | An owner is defined | Choose one existing ErrorModelType and assign it to the ErrorBehavior |
| 58 | InternalFault-Prototype | Has a complementary description | Query Modeller to input a new description for the element or set a default description |
| 59 | InternalFault-Prototype | Is owned by at least one ErrorBehavior | Choose one existing ErrorBehavior and assign it to the InternalFault-Prototype |
| 60 | VehicleFeature | Every function is allocated to at least one AnalysisFunctionPrototype | Choose one existing AnalysisFunctionPrototype and assign it to the VehicleFeatureAllocated |
| 63 | EABoolean | A note is defined | Query the Modeller to input a new text for the note |
| 64 | RangeableDatatype | A note is defined | Query the Modeller to input a new text for the note |
| 65 | EAFloat | The lower threshold is defined | No action, Modeller has to repair the property |
| 66 | EAFloat | The upper threshold is defined | No action, Modeller has to repair the property |

Table 5.1: Implemented properties and proposed Actions in CDL, properties from [MGL⁺11] and [MAL⁺11]

5.3 In-Depth Example: "A Requirement is satisfied" on VehicleFeature

To demonstrate the capabilities of the framework, the property 23 "A Requirement is satisfied" for the EAST-ADL2, element "VehicleFeature" is presented in detail here. The property describes that an instance of the class *VehicleFeature* has to have a realize-relationship to an instance of the class *Requirement*. In the EAST-ADL2 meta-model, this connection is implemented as follows: *Satisfy* contains a realization-connector that connects two UML classes with each other. These two UML classes are contained in an instance of the class *Requirement* and an other class *VehicleFeature*. The relation is schematically depicted in Figure 5.2.

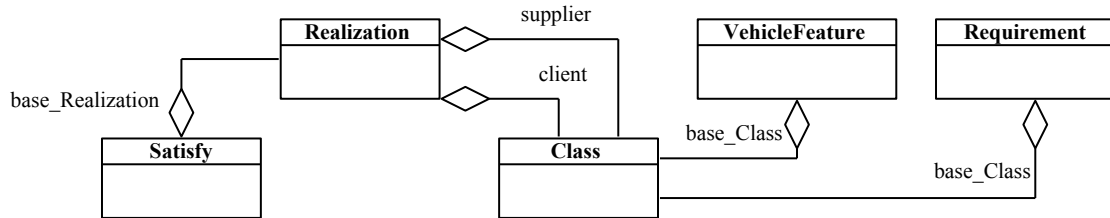


Figure 5.2: Relation of the class *VehicleFeature* with the class *Requirement*

The Criterion to fulfill this property is presented in Listing 5.1. To check this property, all instances of the class *Satisfy*, that contain a *Realization*, as well as all instances of *Requirement* that contain a *Class*, are stored temporarily. In the next step, the EOL functions exists and includes are used to check if there is one *Realization* that fulfills the Criterion.

```

1 //-----
2 // Property ID 23 - VehicleFeature -- A Requirement is satisfied
3 //-----
4 classifier: VehicleFeatureSatisfiesRequirement
5 {
6   model: RecuperationModel
7   description: "Checks property 23, 'A Requirement is satisfied'"
8   severity: WARNING
9
10  criterion:
11  {
12    language: EOL
13    condition: "check{
14      var satisfy = Satisfy.allInstances().select(
15        s | s.base_Realization.isDefined());
16
17      var requirements = Requirement.allInstances().select(
18        r | r.base_Class.isDefined());
19
20      var satisfies = satisfy.select(
21        c | requirements.exists(
22          r | c.base_Realization.supplier.includes(r.base_Class)
23          and c.base_Realization.client.includes(self.base_Class));
  
```


5.3. IN-DEPTH EXAMPLE: "A REQUIREMENT IS SATISFIED" ON VEHICLEFEATURE73

```
24
25
26     return not satisfies.isEmpty();
27     }"
28 }
```

Listing 5.1: Implementation of property 23 from Table 5.1: declaration of the Classifier and Criterion

The corresponding Action of the Classifier presents a list of all requirements to the Modeller and lets him choose one that is further connected to the current *VehicleFeature*, as presented in Listing 5.2. Line 6 to 30 is the code to retrieve a list of all instances of one type, transform it to a list of strings, and present it to the user to select one. In a further release of the CDL framework, this code should be encapsulated into a CDL library function, since it was used slightly modified in several Classifiers. From line 31 on, the base *Realization*- and *Satisfy* classes are created and connected to the corresponding elements.

```
1 action :
2 {
3     name: "AssignVehicleFeatureToRequirement"
4     description: "Assigns the VehicleFeature to one given Requirement"
5     {
6         language: EOL
7         condition:
8         "do {
9             var requirements = Requirement.allInstances();
10            var requirementNames : Set;
11
12            for (requirement in requirements)
13            {
14                if (requirement.name <> '')
15                    requirementNames.add(requirement.name);
16                else
17                    requirementNames.add('Anonym Requirement');
18            }
19
20            var newRequirementName = System.user.choose
21                ('Which Requirement should be connected to $CONTEXT$?',
22                 requirementNames);
23
24            if (not newRequirementName.isDefined())
25                return;
26
27            var newRequirement = requirements.select
28                (requirement : Requirement |
29                 requirement.name = newRequirementName).first();
30
31            var realization = new uml::Realization;
32            realization.name = 'From_'.concat(self.name)
33                .concat('_to_')
34                .concat(newRequirementName);
35
36            realization.supplier = new Set;
37            realization.client = new Set;
38
39            realization.client.add(self.base_Class);
```

```

40     realization . supplier . add ( new Requirement . base _ Class ) ;
41
42     var satisfies = new Satisfy ;
43     satisfies . base _ Realization = realization ;
44     }"
45
46 }
47 }

```

Listing 5.2: Action for property 20 from Table 5.1: query the user for an existing requirement and assign it to VehicleFeature.

5.4 Results

| Category | CDL Framework | Property Checker |
|----------------------------|--|---|
| Decoupling | + Decoupled from the generating editor | – Integrated into the Papyrus editor, applicable for EAST-ADL2 models |
| Execution Performance | – ~2 seconds on the development machine | + ~200 ms on the development machine |
| Development Time | + The case study was implemented in 25 man hours | Not applicable |
| Extensibility | + Extensible due to the use of JAVA and EOL | Not applicable |
| Development of Constraints | + Testable directly on the instance model | – Recompiling the plugin |

Table 5.2: Comparison of the different prototypes to check constraints on the EAST-ADL2 meta-model

In order to make a statement on the benefits of the CDL framework, the results of this case study are compared to the prototypical implementation of the property checker of Mader *et al.* A comparison of the different prototypes can be found in Table 5.2.

One strength of the CDL framework is the separation between the model editor and the definition of the constraints. In the property checker from Mader *et al.*, changing the properties would lead to a redeployment of the plugin, in the CDL framework change can be easily made using the provided editors.

A drawback of the CDL framework is the execution performance: the CDL framework has to read the entire domain instance model every time the constraints are executed. Other approaches, intended to be used with the model editor, can directly operate on the already loaded instance model. On the development machine, the whole case study (69

properties, 46 classifier, 390 objects to check, 2300 lines of CDL code) took around two seconds for every execution. The property checker of Mader *et al.*, in comparison, took around 200ms to check the properties. The overhead of the CDL framework can be roughly divided into reading the instance model (80%) and executing the Classifiers (20%).

A positive aspect of the CDL framework is the integration of different, specialised constraint languages. EOL or OCL allows the definition of constraints more productively due to the use of built-in set theory operators, such as exists or for all. This results in less time to create the constraints, it took 25 man-hours to implement the case study.

Another aspect in favor of the CDL framework is, that in contrast to the property checker of Mader *et al.*, constraints can be directly developed on the current instance model. The plugin does not have to be compiled and deployed every time the constraints change. Furthermore, the graphical representation is a beneficial tool to gain an overview of the constraints, as depicted in Figure 5.3.

5.4.1 Lesson Learned

The approach of directly checking textual constraint expressions that are declared inline, scales up to a certain number of Classifiers. In the case of the EAST-ADL2 meta-model, there should be a separate file for each abstraction level, to keep the number of Classifiers that need to be syntactically checked smaller.

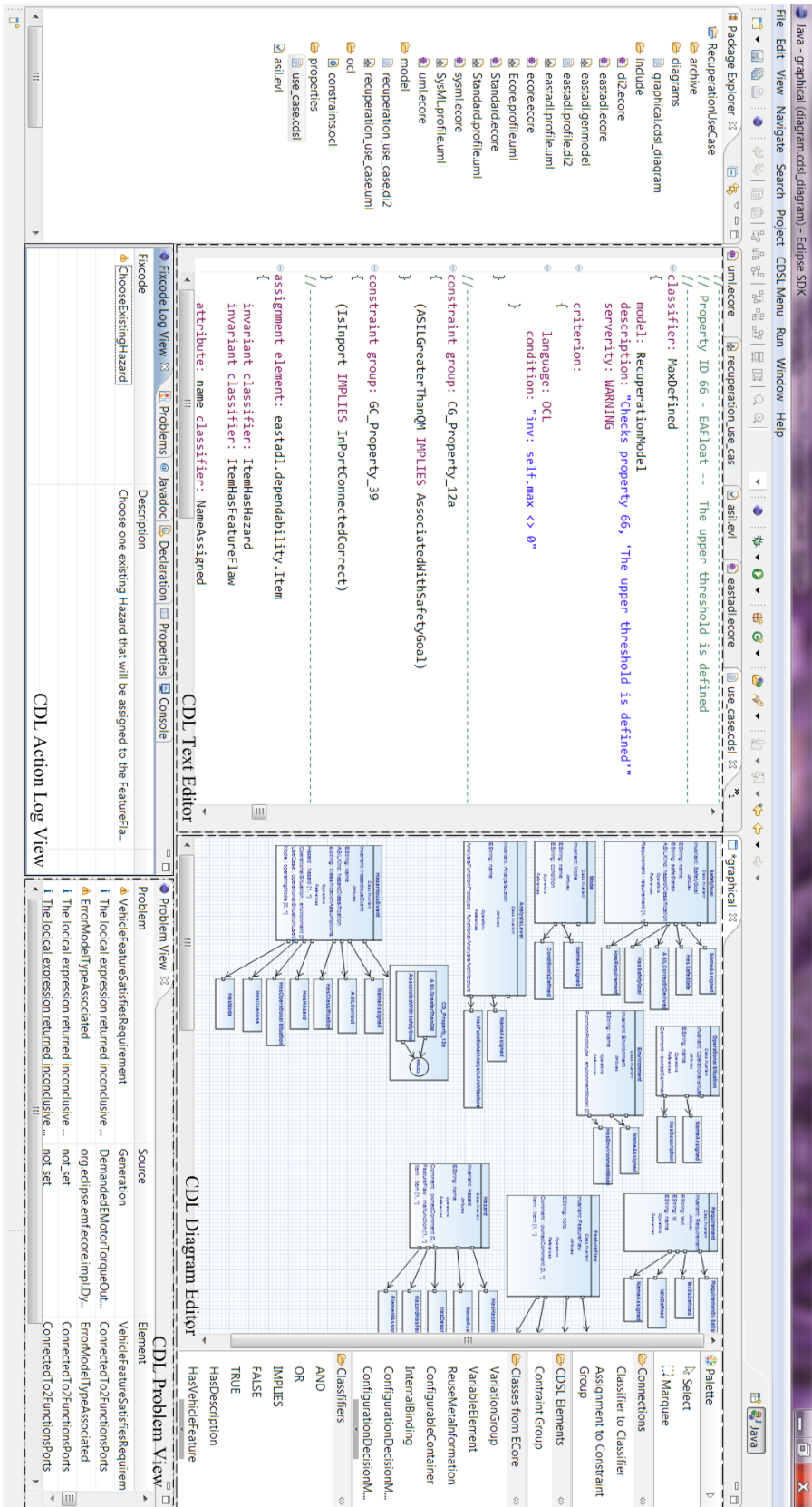


Figure 5.3: Screenshot of the CDL framework plugins, parts of the CDL framework are annotated

Chapter 6

Conclusion and Future Work

6.1 Evaluation with Respect to the Requirements

In Section 2.6, were defined; which were based on the literature research that was conducted. In this section, the compliance of the CDL framework to the defined requirements is presented.

RQ1: The constraint mechanism shall be based on MOF in order to be applicable to all models, no assumption should be made on the editor creating the model. The CDL framework is based on Ecore, the implementation of EMOF in Eclipse. Therefore, the framework can be used to constrain all domain instance models that are defined using an domain meta-model that was defined using Ecore. Furthermore, the CDL framework does not make any assumptions of the editor used to create the domain instance model (c.f. Section 3.2).

RQ2: The constraint mechanism shall be extensible to support models from different abstraction levels in order to support vertical consistency checking. Due to the usage of JAVA and EOL as constraint languages, models from different abstraction levels can be used as input data to define the constraints (c.f. Section 3.2).

In the case study presented in this thesis, vertical consistency of different artefacts *within* the EAST-ADL2 meta-model was shown (c.f. Section 5.3).

RQ3: The constraint mechanism shall be able to integrate existing consistency approaches, such as those presented in Section 2.5 in order to reuse existing effort. In the prototypical implementation of the CDL framework, three different constraint languages, namely OCL, EOL and JAVA, were implemented. To integrated another constraint languages or constraint-defining approaches, the Tool Smith has to implement the well-defined interfaces that are used by the CDL Execution Engine and the Xtext editor (c.f. Section 4.3.4).

RQ4: The constraint mechanism shall support the processing of information from *outside* the model in order to allow the definition of semantic constraints. The CDL framework can use data from outside the model, such as the result of an

interactive query or data from a database, to define constraints. The use of JAVA and EOL as constraint language does not restrict the Tool Smith to information from the model (c.f. 3.2).

RQ5: The constraint mechanism shall be implemented using an open-source modeling tool in order to be validated by others as well. The CDL framework was implemented as an Eclipse plugin, using other open-source plugins such as Xtext or Graphiti (c.f. Section 2.3).

RQ6: The constraint mechanism shall support the Modeller in repairing his instance model. With the possibility of augmenting constraints with Actions, the Tool Smith can provide repair mechanisms to the Modeller. These Actions can query data from the user, open documentation or repair the model using predefined functionality. All the changes of the model can be undone since they are executed within an EMF EditingDomain (c.f. Section 3.3.2).

6.2 Summary

The CDL framework provides an extensible way to define constraints in a model-based development environment. As a novel feature, the framework takes the different roles in the development process into consideration. The framework is designed for three roles: the Tool Smith creates the classifiers since he has the knowledge of the domain meta-model; the Domain Expert combines the classifier into constraint groups and assigns them to elements in the meta-model; and the Modeller applies the constraints on the domain instance model and executes the corresponding Actions when constraints are violated.

Another novel feature of the CDL framework is a reduced language, both textual and graphical, to assign constraints to elements in the domain meta-model, in this way allowing the Domain Expert and the Modeller to define constraints without requiring deep knowledge of the underlying meta-model. The Tool Smith on the other hand, has the possibility of reusing constraints due to mechanisms such as macros and context-independent Classifiers.

The feasibility of this approach has been shown by implementing a subset of properties from [MGL⁺11]. The implemented constraints utilize the strength of the different constraint languages and allow interactive fixing Actions.

6.3 Future Work

The CDL framework does not make any assumption on the editor creating the model. However, in the Eclipse platform the creating editors provide functionality that is already implemented in the CDL framework, such as maintaining the EditionDomain or the resource management. Furthermore, the case study showed that in bigger models, 80% of the execution time is used to read the instance model.

In order to keep the decoupled nature of the CDL framework but eliminate this performance overhead, an extension point could be provided by the CDL Execution Engine that model-creating editors can use to share their models with the CDL framework. In that sense, the framework would still be decoupled, but the need to read the complete model

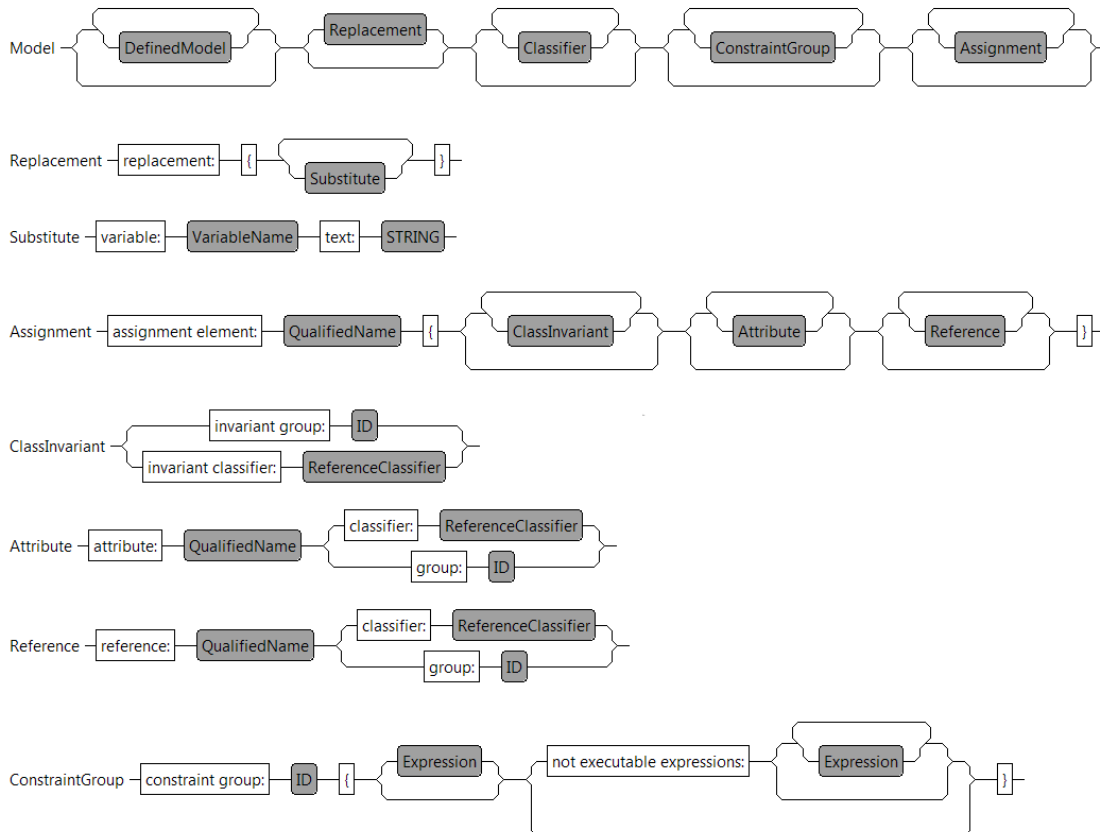
each time would be removed.

Further potential for improvements was identified during the case study: certain functionality, such as querying the Modeller for the correct element or creating an element with a default value, was often used. Therefore, besides the possibility of defining textual macros, the CDL framework should provide a "standard" library of such reusable and parametrized commands.

A more implementation-oriented improvement would be to decouple the integration of the different constraint languages completely and provide them using a package-based extension mechanism. E.g. there could be an OCL or an EOL jar file that includes everything needed to provide the functionality. These packages would then be registered in the CDL framework and loaded at runtime. This mechanism would allow complete independence of the CDL framework and the constraint language interpreters.

Appendix A

Grammar for the Textual Representation



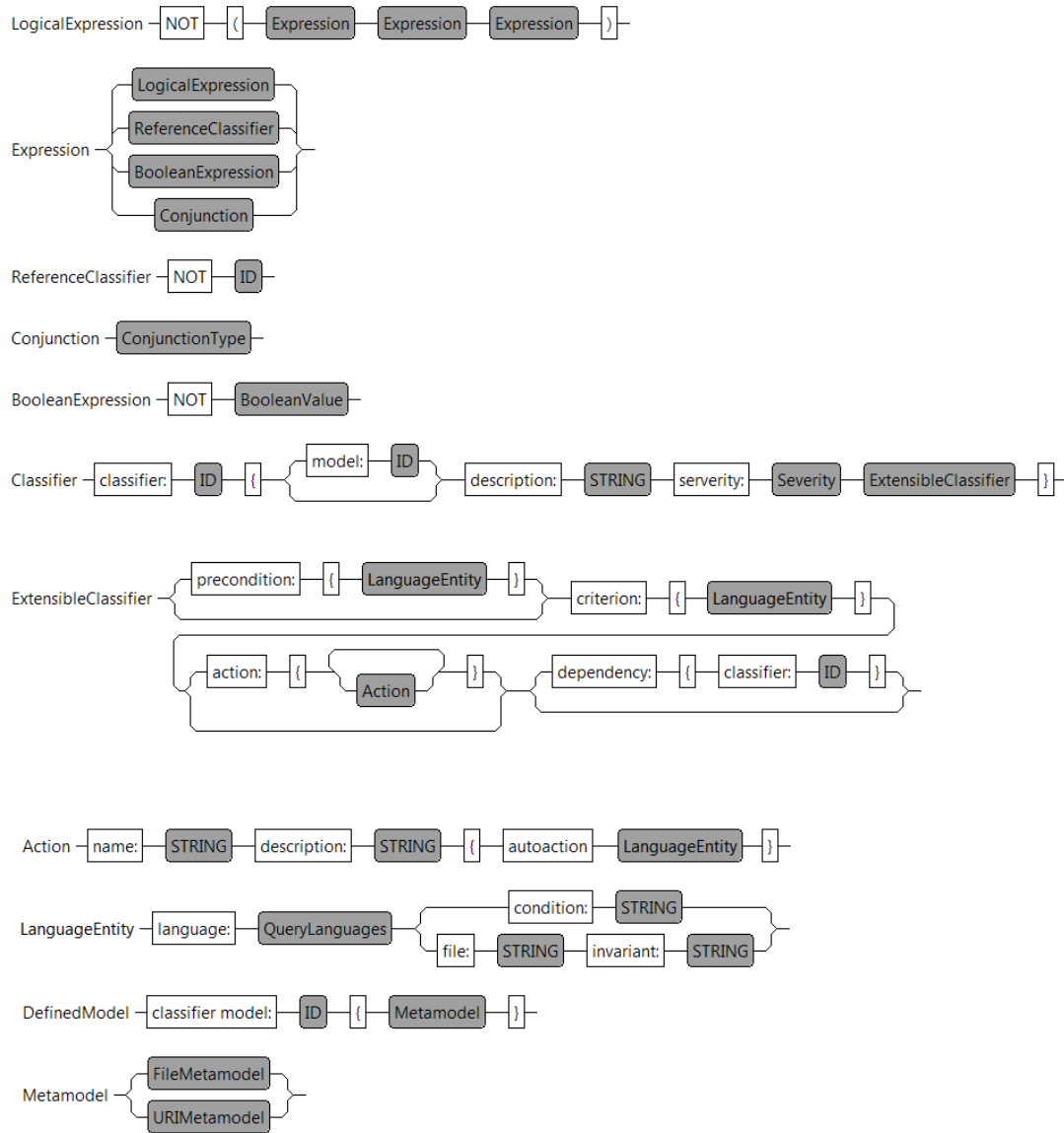


Figure A.1: Syntax tree for the CDL EBNF grammar

Bibliography

- [DH04] Karsten Diethers and Michaela Huhn. Voodoo: Verification of Object-Oriented Designs Using UPPAAL. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 139--143. Springer Berlin / Heidelberg, 2004. 10.1007/978-3-540-24730-2_10.
- [EAS10] EAST-ADL Domain Model Specification, 2010.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.
- [Gro09] Richard C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley, Upper Saddle River, NJ, 2009.
- [Gro11] Object Management Group. Meta Object Facility, 2011.
- [Gro12a] Object Management Group. Object Constraint Language (OCL), 2012.
- [Gro12b] Object Management Group. The Architecture of Choice for a Changing World, 2012.
- [HKRS05] Zbigniew Huzar, Ludwik Kuzniarz, Gianna Reggio, and Jean Sourrouille. Consistency Problems in UML-Based Software Development. In Nuno Jardim Nunes, Bran Selic, Alberto Rodrigues da Silva, and Ambrosio Toval Alvarez, editors, *UML Modeling Languages and Applications*, volume 3297 of *Lecture Notes in Computer Science*, pages 1--12. Springer Berlin / Heidelberg, 2005. 10.1007/978-3-540-31797-5_1.
- [ISO09] Road vehicles - Functional safety - Part 3: Concept phase, 2009.
- [Kle08] Anneke Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, 1 edition, 2008.
- [KPP06] Dimitrios Kolovos, Richard Paige, and Fiona Polack. The Epsilon Object Language (EOL). In Arend Rensink and Jos Warmer, editors, *Model Driven Architecture - Foundations and Applications*, volume 4066 of *Lecture Notes in Computer Science*, pages 128--142. Springer Berlin / Heidelberg, 2006. 10.1007/11787044_11.

- [Kra12] Markus Krallinger. Constraints for a Functional Safety Analysis in CDL. Master Project at the Institute for Technical Informatics, Graz University of Technology, May 2012.
- [LCM⁺03] C. Lange, M. R. V. Chaudron, J. Muskens, L. J. Somers, and H. M. Dortmans. An empirical investigation in quantifying inconsistency and incompleteness of uml designs. In *Incompleteness of UML Designs*, *Proc. Workshop on Consistency Problems in UML-based Software Development, 6th International Conference on Unified Modeling Language, UML 2003*, 2003.
- [LMT09] Francisco J. Lucas, Fernando Molina, and Ambrosio Toval. A systematic review of UML model consistency management. *Inf. Softw. Technol.*, 51:1631--1645, December 2009.
- [MAL⁺11] Roland Mader, Eric Armengaud, Andrea Leitner, Christian Kreiner, Quentin Bourrouilh, Gerhard Griesnig, Christian Steger, and Reinhold Weiß. Computer-Aided PHA, FTA and FMEA for Automotive Embedded Systems. In Francesco Flammini, Sandro Bologna, and Valeria Vittorini, editors, *Computer Safety, Reliability, and Security*, volume 6894 of *Lecture Notes in Computer Science*, pages 113--127. Springer Berlin / Heidelberg, 2011. 10.1007/978-3-642-24270-0_9.
- [MCF03] Stephen J. Mellor, Anthony N. Clark, and Takao Futagami. Guest Editors' Introduction: Model-Driven Development. *IEEE Software*, 20:14--18, 2003.
- [MGL⁺11] R. Mader, G. Griessnig, A. Leitner, C. Kreiner, Q. Bourrouilh, E. Armengaud, C. Steger, and R. Weiss. A Computer-Aided Approach to Preliminary Hazard Analysis for Automotive Embedded Systems. In *Engineering of Computer Based Systems (ECBS), 2011 18th IEEE International Conference and Workshops on*, pages 169 --178, April 2011.
- [MM06] H. Malgouyres and G. Motet. A UML model consistency verification approach based on meta-modeling formalization. In *Proceedings of the 2006 ACM symposium on Applied computing, SAC '06*, pages 1804--1809, New York, NY, USA, 2006. ACM.
- [NEF03] C. Nentwich, W. Emmerich, and A. Finkelstein. Consistency management with repair actions. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 455 -- 464, May 2003.
- [NEFE03] Christian Nentwich, Wolfgang Emmerich, Anthony Finkelstein, and Ernst Ellmer. Flexible Consistency Checking. *ACM Trans. Softw. Eng. Methodol.*, 12:28--63, January 2003.
- [NER00] B. Nuseibeh, S. Easterbrook, and A. Russo. Leveraging inconsistency in software development. *Computer*, 33(4):24 --29, April 2000.
- [OJS05] Dieu Donné Okalas Ossami, Jean-Pierre Jacquot, and Jeanine Souquières. Consistency in UML and B Multi-view Specifications. In Judi Romijn, Graeme

- Smith, and Jaco van de Pol, editors, *IFM*, volume 3771 of *Lecture Notes in Computer Science*, pages 386--405. Springer, 2005.
- [Str05] Ragnhild Van Der Straeten. *Inconsistency Management in Model-Driven Engineering*. PhD thesis, Vrije Universiteit Brussel, 2005.
- [SZ01] George Spanoudakis and Andrea Zisman. Inconsistency management in software engineering: Survey and open research issues. In *in Handbook of Software Engineering and Knowledge Engineering*, pages 329--380. World Scientific, 2001.
- [WGN03] Robert Wagner, Holger Giese, and Ulrich Nickel. A Plug-In for Flexible and Incremental Consistency Management. Technical report, 2003.
- [ZLQ06] Xiangpeng Zhao, Quan Long, and Zongyan Qiu. Model Checking Dynamic UML Consistency, 2006.