FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

A Framework for Modernizing Domain-Specific Languages

From XML Schema to Consistency-Achieving Editors with Reusable Notations

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Patrick Neubauer
Registration Number 1028573

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.-Prof. Mag. Dr. Manuel Wimmer

The dissertation has been reviewed by:

Davide Di Ruscio

Werner Retschitzegger

Vienna, 8th July, 2020

Patrick Neubauer

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

A Framework for Modernizing Domain-Specific Languages

From XML Schema to Consistency-Achieving Editors with Reusable Notations

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der Technischen Wissenschaften

eingereicht von

Patrick Neubauer
Matrikelnummer 1028573

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: Univ.-Prof. Mag. Dr. Manuel Wimmer

Diese Dissertation haben begutachtet:

Davide Di Ruscio	Werner Retschitzegger

Wien, 8 Juli, 2020

Patrick Neubauer

Erklärung zur Verfassung der Arbeit

Patrick Neubauer
7 Barbican Mews, YO10 5BZ York, Vereinigtes Königreich

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 8 Juli, 2020

Patrick Neubauer

Acknowledgements

I would like to thank my supervisor, Prof. Manuel Wimmer, for challenging my intellectual abilities and guiding me towards the successful completion of my doctoral studies. I would like to express my deepest appreciation to my committee members Prof. Werner Retschitzegger and Prof. Davide Di Ruscio for reviewing my work and providing encouraging feedback. I wish to express my deepest gratitude to Prof. Dimitris Kolovos and Prof. Richard Paige for enabling me to join the Enterprise Research group at the University of York and providing unwavering inspiration and constructive feedback during many valuable discussions. I also had the great pleasure of working with Dr Robert Bill and sincerely appreciate his tremendous patience and cooperation during the experimentation and implementation of software components. Especially cheerful to me during this time were Dr Konstantinos Barmpis and Dr Thanos Zolotas who I worked within the research projects CROSSMINER and TYPHON at the University of York. I am extremely grateful to Dr Tanja Mayerhofer and Dr Philip Langer who not only introduced me to the world of academia but also guided me in compiling my first peer-reviewed work. Dr Mayerhofers' attention to detail, which started early in her advisory role in my Masters' thesis, significantly steered the success of this thesis. I also very much appreciate Dr Javier Troja, Dr Martin Fleck, and Dr Alexander Bergmayr and in particular their instrumental role in challenging and motivating me to pursue this work. I would like to pay my special regards to Prof. Gerti Kappel and Prof. Christian Huemer for playing a decisive role in recruiting me for the ARTIST research project and introducing me to Prof. Richard Paige and thus qualifying me to work in an H2020 European research project and enabling me to work on-site at the University of York. I would also like to extend my deepest gratitude to the great advice received from Dr Fady Medhat for reviewing this thesis and providing invaluable guidance in structuring this thesis. And last but not least, I am deeply indebted to my family and friends and in particular my sister Elisa Neubauer and Viesturs Kaugers who never questioned my ability to succeed and kept pushing me towards completing this chapter of my life.

Kurzfassung

Die Einführung von Extensible Markup Language (XML)-Schemadefinitionen (XSDs) stellte einen enormen Fortschritt im Entwurf domänenspezifischer Sprachen (DSLs) dar, da dadurch die formale Definition und maschinelle Verarbeitbarkeit von Domänen-Modellen, wie zum Beispiel das Erfassen von Vokabular und gültigen Sätzen, möglich wurde. Dadurch steigt die Notwendigkeit, integrierte Entwicklungsumgebungen (IDEs) automatisiert zu erstellen und zu warten, um inhärente XML-Einschränkungen, wie zum Beispiel die Syntax von starren Klammern, zu umgehen und um die Formalisierung von gültigen Domänenmodellen zu ermöglichen.

Aktuelle Rahmenwerke modellgetriebener Entwicklung und Sprachentwicklung bieten verschiedene Techniken und Tools, welche die Konstruktion von DSLs grundlegend unterstützen. Nichtsdestotrotz ist es bisher nicht gelungen Unterschiede, welche durch Metamodellen, durch der Transformation und Synthese von XSDs und Sprachgrammatiken aufgedeckt werden, zu automatisieren. Metamodelle, welche von Grund auf unterschiedlich zu Sprachgrammatiken sind, stellen für die Konstruktion und Anwendung von Sprachnotationen somit eine Herausforderung dar. Aktuell sind wissenschaftliche Ansätze aus dem Bereich modellgetriebenen Sprachentwicklung zwar in der Lage, dedizierte Validatoren, Kontextassistenten und Quickfix-Anbietern nahtlos in domänenspezifische IDEs zu integrieren, deren Implementierung und Wartung erfordert jedoch weiterhin Expertise und Arbeitsaufwand im Bereich der Sprachentwicklung.

Im Rahmen dieser Arbeit werden drei Ansätze vorgeschlagen, welche die oben genannten Probleme adressieren. Zunächst wird ein Ansatz vorgeschlagen, welcher die Lücken in der Transformation von Strukturkomponenten schließt und Metamodelle mit strukturellen Einschränkungen aus XSDs anreichert. Dadurch wird die Erstellung von DSL-Grammatiken aus XSD-basierten Sprachen automatisiert. Als Nächstes wird eine Herangehensweise präsentiert, bei der domänenspezifische IDEs mit genauen Validatoren, sinnvollen Kontextassistenten und kostengünstigen Quickfix-Anbietern automatisch generiert werden können, indem zur Laufzeit suchbasierte Softwareentwicklung eingesetzt wird. Der Dritte Ansatz beschreibt wie im Rahmen der Modellierung von Sprachen domänenstrukturunabhängige Textnotationen formuliert werden können. Das Entkoppeln von Repräsentations- und Strukturinformationen in Grammatikdefinitionen sowie das Bereitstellen eines Sprachstil-Rahmenwerkes automatisiert die Konstruktion von Grammatiken aus beliebigen Metamodell- und Stilspezifikationen.

Die Nützlichkeit der vorgeschlagenen Ansätze ist demonstriert durch die Evaluierung der entwickelten prototypischen Implementierungen auf Grundlage der Anwendung eines umfassenden Industriestandards für die Klassifizierung und Beschreibung von Produkten, dem Vergleichs mit aktuellen Rahmenwerken der Sprachentwicklung, der Integration in ein Werkzeug der Unterstützten Modellierung sowie der Durchführung individueller Einzelfallstudien wie Cloud-Topologie und Orchestrierungsmodellierung.

Abstract

The introduction of Extensible Markup Language (XML) Schema Definitions (XSDs) represented a tremendous leap towards the design of domain-specific languages (DSLs) by enabling machine processibility of domain models conforming to formally described language grammar, i.e. capturing vocabulary and valid sentences. Consequently, it elevated the need for automating the creation and maintenance of dedicated and modern integrated development environments (IDEs) evading inherent XML limitations, such as rigid angle-bracket syntax, as well as enabling the support of valid domain model construction.

Techniques and tools provided by model-driven engineering frameworks and language workbench frameworks offer elementary assistance during the initial implementation of a DSL. These frameworks, however, fail to automate DSL generation due to disparities exposed by the transformation and synthesis of XSDs, language grammars, and metamodels. Moreover, fundamental differences in the nature of language grammars and metamodels challenge the construction and application of language notations. Although language workbenches are capable of integrating seamlessly dedicated validators, context assistants, and quick fix providers into domain-specific IDEs, their implementation and maintenance still requires proficient language knowledge and development.

This thesis contributes towards addressing the above-mentioned problems. First, it proposes an approach to generate automatically DSL grammars from XSD-based languages by bridging gaps in the transformations of structural components, and by enriching metamodels with structural constraints imposed by XSD restrictions. Second, it proposes an approach to generate automatically domain-specific IDEs with accurate validators, sensible context assistants, and cost-effective quick fix providers by employing search-based software engineering at runtime. Third, it proposes an approach to formulate domain structure-agnostic textual notations for modeling languages by decoupling representational from structural information in grammar definitions, and by providing a language style framework capable of generating grammars from arbitrary metamodels and style specifications. In order to demonstrate the usefulness of the proposed approaches, the developed prototypical implementations are evaluated based on a comprehensive industrial standard for the classification and description of products, a comparison with state-of-the-art language workbench frameworks, integration with model assistance tooling, and individual case studies such as cloud topology and orchestration modeling.

Contents

CHAPTER 1

Introduction

Software language engineering constitutes an emerging field focusing primarily on language definition, parser building, and compiler construction [215, 121, 75]. Traditionally, there has been a strong focus on language design to provide machine-readable and executable formats. In recent decades, there has been a shift towards paying more attention to concrete syntax, i.e. the notation or visual representation of a language, due to the growing need of program creation and maintenance by non-programming domain experts, and the emergence of powerful tools for the definition of Domain-Specific Languages (DSLs) that automate the construction and maintenance of complex language implementation artifacts, such as compilers and parsers [84, 65, 92].

1.1 Motivation and problem

One of the primary motivations for developing a novel language is to make the construction and maintenance of instructions (programs) more efficient for their primary stakeholder, i.e. the end-user (henceforth referred to as *domain expert*), who composes programs by employing tools offered by the implementation of a language. Ideally, a language offers (*i*) a suitable level of abstraction, (*ii*) a fitting balance between expressiveness and conciseness, (*iii*) an assurance mechanism for critical properties, and (*iv*) rigorous semantics [131].

Several studies report that Domain-Specific Languages (DSLs) are more expressive and easier to use in their domain of application compared to GPLs, such as Extensible Markup Language (XML) and Java, and consequently tend to raise productivity and improve maintenance [210, 222]. For example, DSLs enable the construction of terminology that is closer to their domain than GPLs [133], i.e. they offer generic language constructs, such as elements and attributes in XML Schema Definition (XSD) and classes and class members in Java, the support for domain-specific notations, as well as explicit separation of knowledge in the system in the natural structured form of its domain, such

as car and horsepower in a DSL for the specification of vehicles. Thus, DSLs enable domain experts to exert their existing domain proficiency during activities, such as the reading and learning of notations. More specifically, a repeated empirical study on the comparison of GPLs and DSLs reports that the latter outperform the former in thirteen out of thirteen cognitive dimensions, primarily categorized into learning, perception, and evolution despite the participants' superior experience in the former [91, 130].

In summary, for DSLs to become successful, it is crucial that: (*i*) the entrance barrier for the development of DSLs and, in particular, language exploitation-based DSL design, is lowered; (*ii*) elaborated tool support is available, such that productivity gains can be achieved by accelerating model creation and maintenance, as well as fault resolution; and (*iii*) language notations can be constructed closely to their domain of representation, so that domain experts are able to make use of their existing domain proficiency. Finally, the construction of such DSL implementations requires the least amount of language engineering expertise to outline the preliminaries for both minimal intervention by highly skilled language engineers, and the extended scope of domain experts.

In general, DSLs are constructed by following either the *language invention* design pattern, i.e. DSL construction from scratch, or the *language exploitation* design pattern, i.e. DSL construction based on existing (formal) language specifications [152]. In the latter case, it is essential to provide mechanisms to generate comprehensive implementations with the least amount of developer intervention in order to lower the barrier for the manifestation of DSLs built by language exploitation.

In 1998, the World Wide Web Consortium (W3C) introduced the fully machine-processible XML that represents a tremendous leap towards easing the design of language specifications by leveraging the idea of a generic editing, parsing, and validation methodology. In 2004, W3C recommended the XML Schema Definition (XSD) as a new standard to describe formally a DSL and verify the conformance of XML documents to their DSL [97]. On the one hand, for some XSD-based languages, such as the prominent Business Process Model and Notation, dedicated tool support, such as textual and graphical IDEs, has been made available. On the other hand, for a vast amount of XSD-based languages, dedicated tool support is lacking. XSD-based languages, however, represent ideal candidates for the construction of DSLs by following the language exploitation design pattern. More specifically, methodologies and techniques made available by Model-Driven Engineering (MDE) [31] and, in particular, Model-Driven Language Engineering (MDLE) [136, 113] lower the entrance barrier to capture and manage primitive models containing domain-specific information, by semi-automating the generation of dedicated Integrated Development Environments (IDEs) [127].

XML has primarily been designed as a machine-processible format composed of immutable concrete syntax. More specifically, users of XML-based languages are bound to tree-based angle-bracket syntax that is described as verbose and complex in terms of human comprehension [13]. One of the main consequences of immutable generic textual concrete syntax is its limitation to improve upon human comprehension and, therefore, maintainability. It has been shown that several low-level primitives, which

are often entirely syntactical, represent one of the most substantial cognitive barriers for end-users [144].

Therefore, in order to overcome the limitations of inflexible XML syntax, it is necessary to construct dedicated DSL implementations with customized units of symbols for domain-specific concepts and relationships. The development and maintenance of DSL implementations from scratch, however (i.e. DSL development based on the *language invention* design pattern), involves performing a series of complex and error-prone tasks that require highly skilled language engineering expertise [86, 205, 117]. Therefore, for existing formal language specifications, such as XSDs, the less exhaustive *language exploitation* design pattern may be preferred [152].

Favre et al. highlight that an important property of technological spaces[1] is that there is no "best technological space." Rather, the choice depends on the problem at hand. A case in point is that, while there are machine-oriented languages, like XML, on one side of a spectrum, there are also human-oriented languages, like UML. Technological spaces are not islands, and the notation of bridges between spaces offers extensive value if facilitated. For example, the construction of bridges between such languages may improve the comprehensibility and, thus, the maintainability of DSLs. Further, it has been suggested that effective DSL implementations are designed for both machines, as well as humans [72, 75]. Generally, the former requires formal language definitions, such as abstract syntaxes, that enable straightforward machine-processing, while the latter requires the definition of concrete syntaxes or units of symbols that emphasize human perception, cognition, and usability [155].

Although MDE and language workbench frameworks, such as Xtext, can generate metamodels from XSDs and DSL grammar from metamodels, these bridges are neither integrated with each other nor complete [140, 218]. This would mean that intervention by a language engineer is required to transform concepts and types that are not considered by existing transformations. In particular, challenges introduced by the migration of mixed content and wildcards, data types and restrictions, as well as identifiers and references that may be defined as part of an XSD need to be addressed. Furthermore, effective language migration is challenged by the maintenance of backward-compatibility and interoperability between DSL models and existing language implementations, such as XML applications.

Although efficiency gains in the construction of textual modeling language implementations have been attributed to the reduction of initial engineering workload and complexity through the emergence of language workbenches, and to the application of automation techniques for the generation of executable language artifacts, such as parsers, serializers, and basic IDEs, several obstacles that impede the practical development, maintenance, and use of dedicated IDEs prevail. This implies that dedicated IDEs of textual modeling languages that retain syntactical model correctness and offer vigorous

[1] Kurtev et al. define a technological space as "a working context with a set of associated concepts, body of knowledge, tools, required skills, and possibilities." [140]

tooling, such as model consistency-preservation and low-cost consistency-repair, known from the IDEs for popular GPLs, such as Eclipse for Java, still remain the exclusive result of highly skilled language engineering expertise combined with domain knowledge—a combination that is rarely found in practice [86, 205, 117, 152].

Although it has been shown that notations of DSLs are better learned by domain experts compared to notations of GPLs [48], the latest developments in construction and maintenance of dedicated textual language notations is still challenging due to the development of DSL grammar requiring the execution of a series of complex and error-prone tasks [49]. This is why the construction and maintenance of dedicated IDEs is limited to developers with significant expertise in the design and engineering of languages, a fact which holds back the industrial adoption of DSLs.

The engineering of a DSL is usually initiated by the construction of an artifact that captures the concepts and relationships that are inherent in their domain of representation. Typical artifact types include variations of grammars and metamodels[2]—each of inherently different nature [122]. Generally, grammars are employed to describe domain concepts and their textual representation, using production rules and terminal rules respectively. Conversely, metamodels are employed to capture the concepts and relationships of a domain but not their syntactical constructs. Although state-of-the-art language workbenches provide mechanisms to generate grammars from metamodels and vice-versa, they provide a single (default) notation, i.e. graphical or textual, that has to fit the needs of all types of domain experts or requires dedicated language engineering skills for adaptation and extension. The construction of bridges between metamodels and grammars, in particular, is commonly approached by introducing annotations in metamodels or metamodel-to-grammar transformations. The construction and maintenance of such bridges, however, are inherently complex and error-prone due to the fundamental differences between metamodels and grammars. Moreover, bridges between metamodels and grammars that employ metamodel annotations or metamodel-to-grammar transformations are not metamodel-agnostic and, thus, not applicable to arbitrary domains. Therefore, building a particular notation for a modeling language that instantiates either of the above-mentioned bridges, thus encoding style information as (part of) the annotations in the metamodel, the metamodel-to-grammar transformation, or the adaptation of grammar that has been generated as a result of instantiating the constructed bridge, neglects reusability. For example, building a DSL, such as a vehicle DSL mentioned above, through the construction of metamodel-annotations, a metamodel-to-grammar transformation, or generated grammar-adaptation that reflects a white-space aware notation does not enable employing white-space aware styling to a DSL reflecting concepts and relationships in the domain of biology.

In summary, problems in the migration of language structure and, in particular, the integration of modeling languages from XSDs, the development and maintenance of dedicated IDEs, and textual language notations have been presented and analyzed. The

[2]DSLs that employ metamodels to represent domain-specific concepts and relationships are also referred to as modeling languages.

following section outlines the objectives of this thesis and, in particular, the tackling of the presented problems, as well as the fundamental methodology used during the iterative constitution and evaluation of the individual contributions of this thesis.

1.2 Objective and methodology

This thesis aims at addressing the challenges of Domain-Specific Language (DSL) modernization and the bridging of the technical spaces XMLware, Modelware, and Grammarware [139, 218]. The goal of this thesis is to advance the: (*i*) automated migration of modeling languages from XSD-based languages; (*ii*) automated generation of consistency-achieving IDEs from formally constrained language structures; and (*iii*) modeling and application of textual notations among arbitrarily structured modeling languages.

In particular, the following research questions are addressed in this thesis.

(I) Is language exploitation suitable for the automated migration of backward-compatible textual modeling languages from XSDs and integrated modeling assistance?

(II) Are formal constraint specifications in structural language specifications suitable for the automated generation of consistency-achieving language implementations?

(III) Are metamodel-agnostic definitions of textual language notations suitable for the development and maintenance of representations of domain concepts and relationships?

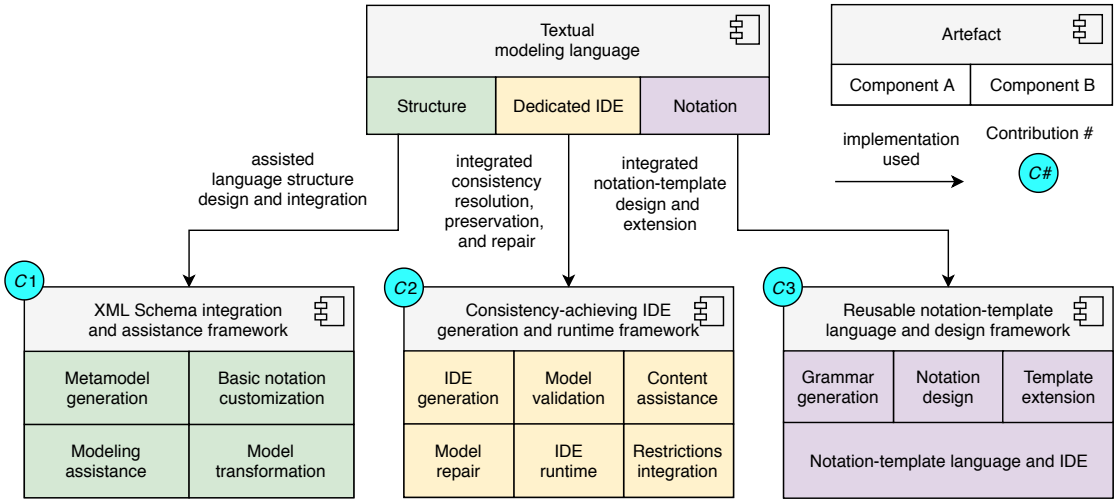

Figure 1.1: Contributions of this thesis.

These research questions have been addressed in this thesis resulting in three contributions (cf. Figure 1.1) that are summarized below.

Contribution 1: Automated migration and integrated modeling assistance for modeling languages by XML Schema-based language exploitation. This contribution (cf. C1 in Figure 1.1) provides an approach and implementation that automates the migration of textual modeling languages by employing the *language exploitation* design pattern on XML Schema descriptions. In particular, the limitations of state-of-the-art bridges between the technical spaces of XMLware, Modelware, and Grammarware are exposed in a reproduction study and subsequently conquered [162]. As a result, the basic requirements for sovereignty from immutable angle-bracket syntax are established, and an integration for automated modeling assistance is provided [192]. Further, the implementation of the approach is evaluated in the context of a large and complex industrial standard that is formally represented as a set of XML Schema descriptions. Furthermore, in order to achieve the backward-compatibility of migrated languages to existing languages and, particularly, XML-based applications, a generic bidirectional transformation for model serialization and parsing is constructed, thus, enabling language users to take advantage of the amenities of both comprehensive handcrafted implementations of an existing language, where available, and the implementation of a migrated language in the more powerful and less restrictive technical space of Modelware. This contribution addresses research question *(I)*.

Contribution 2: Consistency-achieving IDE generation and runtime for formally constrained structural language specifications. This contribution (cf. C2 in Figure 1.1) provides an approach and implementation that automates the generation of consistency-achieving language implementations, i.e. dedicated language IDEs and tool support, for modeling languages in general and for XSD-migrated languages in particular [164, 165]. Advanced IDE features, which are the result of highly skilled language development expertise known from popular GPL implementations, such as Java, and particularly IDEs, such as Eclipse and IntelliJ, include precise validation reports, sensible content-assist suggestions, and low-cost model repair solutions that are automatically generated from formally constrained structural language specifications and, particularly, Ecore metamodels that are constrained by OCL expressions. More specifically, model repair solutions are generated and ranked by the development and employment of a custom search technique for model repair, tackling the state space explosion problem by dividing the state space into three stages, and executing them in parallel. Thus, repair solutions for timely visualization are generated for the end-user. This contribution addresses research question *(II)*.

Contribution 3: Textual notation modeling language and framework for the definition and application of style models that are structure-agnostic, structure-dependent, or a combination thereof. This contribution (cf. C3 in Figure 1.1) provides an approach for the definition and application of textual language notations by introducing a language for capturing concrete representations of abstract concepts and relationships that may be structure-agnostic, structure-dependent, or a combination thereof. The approach is implemented by a concrete syntax specification language, inspired by the Cascading Style Sheets (CSS) language for markup language

documents, such as those of the Hypertext Markup Language, and a framework that enables the construction of style models, containing grammar rule templates and injection-based property selection, and the application thereof to arbitrary metamodels [163]. The approach aims at reducing the redundancy between metamodel and grammar elements that is introduced by employing state-of-the-art language workbenches for the development of textual modeling languages and, particularly, the model-first approach in which language development is initiated by metamodel construction and followed by grammar generation and adaptation. This contribution addresses research question *(III)*.

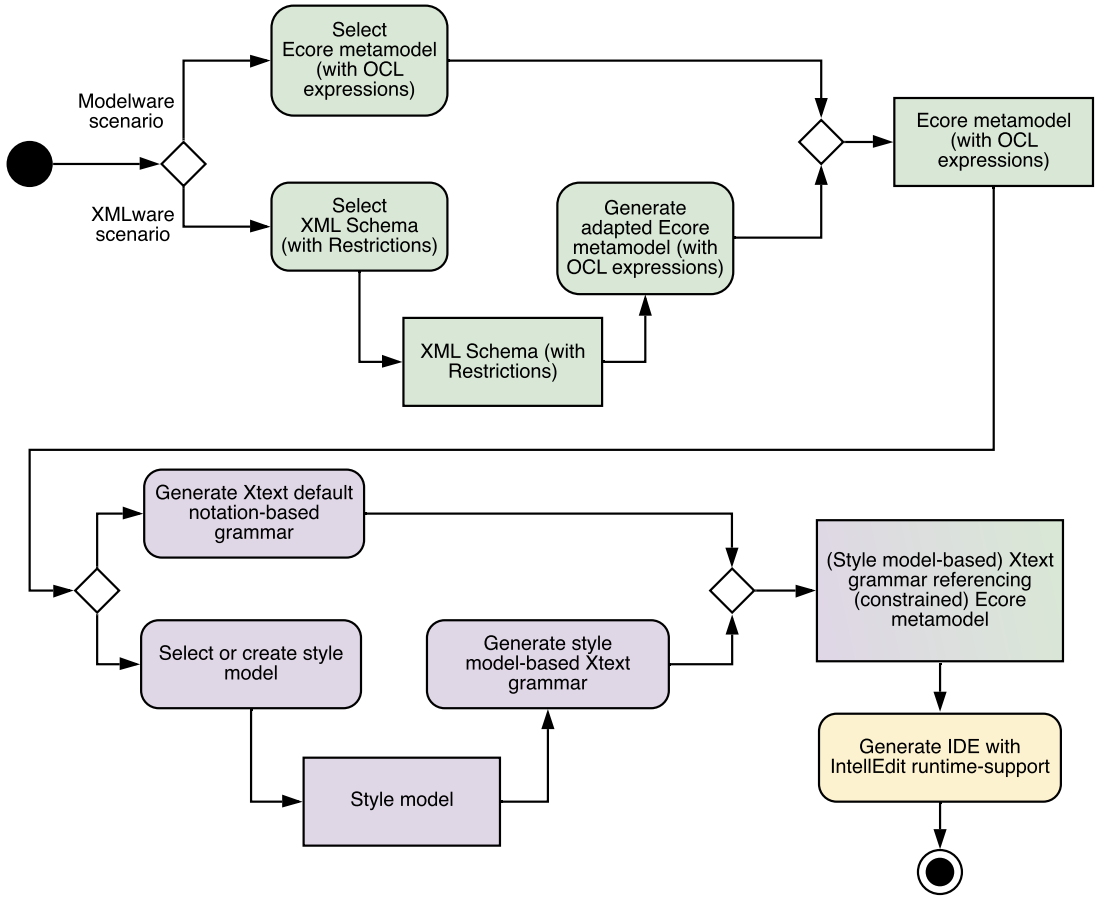

Figure 1.2: Primary flows of operation integrating the contributions of this thesis.

Figure 1.2 illustrates the primary flows of operation that integrate the contributions of this thesis in form of an UML activity diagram. First, a (formally constrained) language structure is selected. More specifically, in the Modelware and XMLware scenario an Ecore metamodel (with OCL expressions) and XML Schema definition (with XML Schema Restrictions) is selected respectively. In the latter case, the first contribution of this thesis is responsible for establishing Ecore metamodels from structural language specifications in XML Schema definitions. Second, Xtext grammar which references the respective input

Ecore metamodel (with OCL expressions) is generated immediately (and thus follows the Xtext grammar default notation) or subsequently to the selection or creation of a style model. In the latter case, the third contribution of this thesis addresses the generation of style model-based Xtext grammar. Third, the second contribution of this thesis accounts for the generation of consistency-achieving IDEs from (style model-based) Xtext grammars which reference (constrained) Ecore metamodels and offer precise validation reporting, sensible content-assistance, and economical model repair.

The design science paradigm has been applied as the fundamental methodology that constitutes and evaluates the contributions of this thesis. Design science is a constructive methodological approach, according to which knowledge is created by building and evaluating innovative artifacts. Hevner et al. presented a conceptual framework, composed of seven guidelines, that illustrates the application of design science in information systems research [104]. These seven guidelines have been applied in this thesis as described in the following.

1. Design as an Artifact. This thesis provides the theoretical foundation, as well as the implementation of a set of contributions. Notably, the following artifacts have been produced as the outcomes of this thesis:

- A framework for automating the migration and integrated modeling assistance of textual modeling languages from XSDs;
- A framework for automating the generation of executable language implementations, i.e. dedicated language IDEs, from formally constrained language structures that offer vigorous tool support through accurate validation, sensible content-assist, and low-cost quick fix solutions;
- A language and framework for the definition of style models, i.e. textual language notations, as well as the application of style models to arbitrary metamodels for the automated generation of executable language implementations.

2. Design Evaluation. The artifacts developed in the course of this thesis have continuously been evaluated to determine whether they are useful, as well as to guide the direction of further development (cf. Runeson and Höst [187]).

3. Research Contributions. The main research contribution of this thesis is the experience gained in the development of an approach to bridge the technological space of XMLware with textual modeling language implementations that tackle the limitations of existing approaches, as well as the theoretical foundations and implementation thereof. A detailed description of individual research contributions is presented in Sections 4, 5, and 6.

4. Research Rigor. The existing set of works in the area of Domain-Specific Language (DSL) engineering and the bridging of technological spaces has been explored during the course of this work, and cited where appropriate. Thus, existing literature and tool implementations have been surveyed for individual scientific contributions, as well as artifact evaluation strategies and their applicability within the context of this work.

5. Design as a Search Process. All artifacts in this thesis have been constructed in an iterative fashion. First, all techniques and solutions have been evaluated manually on systematically developed scenarios, such as reproduction studies, in order to provide an initial verdict on their feasibility. The approach has been generalized capturing these simple scenarios. Subsequently, more complex scenarios have been evaluated and automated until either the initially defined target was reached, or their automation rendered itself too cumbersome or not possible at all. During this search, alternative paths have been identified and pursued.

6. Communication of Research. The research has been disseminated through well-known communication channels in the MDE, software language engineering, as well as the general computer science community. Individual contributions of this thesis have been published in the following periodicals, venues, and co-located workshops or events:

- International Conference on Software Language Engineering (SLE, ACM)
- International Conference on Software Analysis, Evolution, and Reengineering (SANER, IEEE)
- Doctoral Symposium at International Conference on Software Technologies: Applications and Foundations (STAF, Springer)
- International Journal of Computer Languages, Systems and Structures (COMLAN, Elsevier)
- International Workshop in OCL and Textual Modeling at Conference on Model Driven Engineering Languages and Systems (OCL at MODELS, ACM/IEEE)

1.3 Structure of work

This thesis is structured according to its elaborated contributions. In what follows, an overview of this thesis is provided that briefly describes the contents of each chapter. Some of the contributions of this thesis have already been published in peer-reviewed venues, such as workshops, conferences, and journals. Hence, the contents of these publications overlap with the contents of this thesis.

Chapter 2: Preliminaries and running example. The aim of this chapter is to introduce the basic concepts on which the contributions are founded. In particular, the technical spaces XMLware, Modelware, and Grammarware are introduced and, in particular,

their mechanisms for capturing language structure and structural constraints. Finally, technological spaces are summarized with a comparison of individual terminology.

Chapter 3: Related work. This chapter introduces the latest developments in: (*i*) the bridging of technological spaces; (*ii*) language engineering through the application of language workbenches; and (*iii*) the design of textual notations for DSLs.

Chapter 4: XML Schema modeling integration and assistance. This chapter presents the contribution of this thesis regarding the automated migration of textual modeling language implementations from XSDs based on the language exploitation design pattern, as well as a reproduction study. Further, integrated modeling assistance for XSD descriptions is presented alongside an evaluation of XML Schema-based language exploitation involving an XSD-based industrial specification for classification and product descriptions. Finally, the latest developments in the bridging of technological spaces (*i*) modelware and grammarware, (*ii*) XMLware and modelware, and (*iii*) XMLware and grammarware are compared with the proposed approach for the automated migration of textual modeling languages from XSD.

Chapter 5: Consistency-achieving integrated development environment. This chapter presents the contribution of this thesis concerning the automated generation of consistency-achieving language implementations from formally constrained language structures, alongside a custom approach for mitigating the state-space explosion problem in a timely manner. Moreover, an evaluation is presented that compares IDE tooling generated by state-of-the-art language workbenches and IDE tooling generated by the implementation of this contribution and, in particular, model validation, content-assist, and model repair.

Chapter 6: Reusable notation-template language and design framework. This chapter presents the contribution of this thesis regarding the development and application of textual language notations that are structure-agnostic, structure-dependent, or a combination thereof. Further, an evaluation is presented that compares the cost of adapting language grammar that has been generated from structural language specifications with the construction and application of style models yielding similar language grammar and, thus, executable language implementations. Finally, the latest developments of textual notation design are compared with the proposed contribution for the construction and maintenance of textual notations for modeling languages.

Chapter 7: Conclusion and future work. This chapter summarizes the contributions of this thesis, points out their limitations, and gives directions for future lines of inquiry.

CHAPTER 2

Preliminaries and running example

This chapter introduces the elementary concepts and technologies on which the contributions of this thesis are founded. First, the general notion of technical spaces is introduced alongside an example language for the definition of space transportation services. Second, the technical spaces of XMLware, grammarware, and modelware are presented. Third, different methodologies for capturing the structural semantics and constraints that are employed by individual technical spaces are compared and summarized.

2.1 Technical spaces

A technical space is defined as a working context with a set of associated concepts, body of knowledge, tools, required skills, and possibilities. Simply put, a technical space is a representation of a designated technology at a higher level of abstraction, intended to enable reasoning about the correlation and the individual characteristics of heterogeneous technologies. Bézivin and Kurtev [139] argue that individual technologies may have emerged in rather isolated communities with dedicated knowledge, skills, tooling, educational support, literature, and research venues, consequently impeding the development and maintenance of implementations that rely on the integration of various technologies. The understanding of the advantages and disadvantages of individual technical spaces, as well as their common characteristics and complimentary features, however, is of paramount importance to improve knowledge transfer and, ultimately, accomplish integration. Bridging technical spaces refers to the ability to integrate with or interact among individual technologies. In other words, a bridge between two technical spaces enables the transferring of artifacts from one technical space to another. The aim of bridging technical spaces is to combine the capabilities that are offered by individual

technical spaces for the purposes of enabling reuse and lowering the cost of solution development. In addition, building bridges to technical spaces that (already) provide the desired capabilities renders the need of costly (re-)implementation superfluous. For example, a multitude of standards, such as the Topology and Orchestration Specification for Cloud Applications (TOSCA) [26], are often formally defined by means of XSD documents. Subsequently, such XSD documents are employed as formal blueprints for the implementation and maintenance of supporting language infrastructure (e.g. dedicated tools and complex engines) providing the ability to edit, visualize, and execute conforming XML documents. Although dedicated tools, such as IDEs and visual notations, and execution engines, such as cloud orchestration engines for instance, are available for some (prominent) XSDs (e.g. Vino4TOSCA [35] and OpenTOSCA [25] in the case of TOSCA), a multitude of XSD-based Domain-Specific Languages (DSLs) fail to obtain the resources required for the development and maintenance of supporting language infrastructure. Consequently, domain-experts, i.e. the end-users of DSL implementations, are unable to escape the limitations of a particular technical space (e.g. immutable angle-bracket notation in the case of XMLware), or exploit the capabilities that are available in (other) technical spaces (e.g. cost-effective implementation of dedicated tooling and notation).

The attempt to bridge technical spaces imposes several problems [218, 24]. First, semantic incompatibilities between technical spaces render the transformation of artifacts difficult or impossible. In other words, while a concept, such as the definition of terminal rules for the specification of language notations, exists in one technical space, such as grammarware, it may not exist in another technical space, such as XMLware. Second, the differences in expressiveness among technical spaces may result in loss of information. This means that, while a concept can be expressed in one technical space, it may not be expressible in another technical space. Third, there may exist a multitude of options or alternatives to transform an artifact from one technical space to another. Therefore, while there may (only) exist a singular type of representation for a particular artifact in one technical space, such as attributes in XMLware, there may exist a multitude of types of representation in another technical space, such as attributes and references in modelware, i.e. each intended for a distinctive purpose.

The upcoming sections present the technical spaces of XMLware, grammarware, and modelware, most notably, their individual mechanisms for the definition of structural semantics and constraints that are illustrated on a language for the specification of space transportation services that is defined by the following concepts and relationships: a *SpaceTransportationService* can own launch sites, spacecrafts, and engine types; a *Spacecraft* with name, relaunch-cost, stages, manufacturer, country of origin, physical properties, functions (such as being an orbital launcher or intercontinental transport vehicle), and launch sites; a *Stage* with name, such as booster or spacecraft, an engine type, and physical properties; a *PhysicalProperty* with type, such as length, volume or mass, unit, and value; a *LaunchSite* with name, location, operator, number of launchpads, operational status, and physical properties.

2.2 XMLware

XMLware, also referred to as *Documentware*, is the technical space [139] of XML and, in particular, XSD, which enables the definition of markup languages [73, 34]. XML is standardized by the World Wide Web Consortium (W3C) and represents a widely accepted representation and exchange format for structured and semi-structured data. XML documents are constrained by well-formedness, and are validated, e.g. based on XSD or XML Document Type Definitions (DTDs) documents.

2.2.1 Extensible markup language

XML is a markup language that enables the definition of structured and semi-structured data and documents, and is intended to be a generic exchange format that is applicable across different platforms that is simple to use [33]. Moreover, the XML Specification describes the intent of XML-encoded documents to be both human-readable and machine-readable [34]. Although the advent of XML represents a leap towards the definition of languages, existing literature highlights the limitations of XML documents, and presents alternatives in regards to human-processibility [84, 185, 199]. The key concepts illustrated in the XML Specification include the declaration and encoding of XML documents, the processing of XML documents, the specification of content and markup in XML documents, and the use of markup tags and markup attributes in XML documents.

Declaration and encoding. An XML document is represented as a string, i.e. a sequence of characters, that follows an explicit encoding, such as UTF-8, i.e. a one-byte encoding that conforms to the ISO 10646/Unicode standard specification [21]. For example, an XML document declaration that indicates the application of XML version 1.0 and UTF-8 character encoding is specified at the beginning of the document by the statement *<?xml version="1.0" encoding="UTF-8"?>*.

Processing. An XML processor, i.e. also referred to as XML parser, analyzes and constructs a structure of the information depicted by the content of the XML document being processed, and returns the result to the caller of the XML processor, i.e. the application requesting to process an XML document.

Content and markup. XML documents are split into markup and content that are typically distinguished by syntactic rules, such as opening and closing angle-brackets for the former, and any valid stream of characters for the latter, i.e. following the document character encoding. For example, *<manufacturer>Acme Corporation </manufacturer>* marks *Acme Corporation* as *manufacturer*, while the string *Acme Corporation* without the enclosing *manufacturer* tag may indicate mixed content, i.e. capable of capturing attributes, elements, and free-form text.

Markup tags. A tag that indicates markup in an XML document is either defined by a start and end, or an empty element tag. For example, *<manufacturer> </manufacturer>*

indicates a start- and end-tag and *<manufacturer/>* an empty element tag. The latter is typically employed to markup elements that either do not contain information or specify information (only) by the mechanism of attribute values, such as *<manufacturer name='Acme Corporation'/>*.

Markup attributes. Attributes that indicate markup in an element of an XML document are composed of name-value pairs and appear within the context of a tag or, put differently, angle-brackets that enclose the markup indicated by a tag. For example, *<country code='US'> </country>* indicates the use of attribute *code* with value *US* in the context of the element *country*.

2.2.2 XML Schema definitions

At the time of writing, XML Schema Definition (XSD) had been first standardized by the World Wide Web Consortium (W3C) in 2004 (version 1.0), and its last standardization occurred in 2012 (version 1.1), through a two-part recommendation [202, 28]. Part one of the recommendation describes the XSD language, and its ability to define the structure and constraints of conforming XML documents. Part two of the recommendation describes the use of XSD for the definition of datatypes that may be employed in an XSD and XML document, representing a superset of datatypes that are made available by XML Document Type Definitions (DTDs). Consequently, XSDs are employed particularly to specify formally valid compositions of XML documents.

In what follows, the principal components of the XSD Specification are presented by means of an example language for the specification of space transportation services. More specifically, Listings 2.1–2.5 provide excerpts of the XSD specification of the space transportation services domain (full version in Appendix 1) in order to illustrate the XSD concepts *document and encoding*, *namespaces*, *elements and particles*, *complex types*, *attributes*, *groups*, and *simple types*.

Document and encoding. Listing 2.1 line 1 declares the type of the document as XML version *1.0* with character encoding *UTF8*, i.e. a variable width character encoding that is capable of encoding all valid code points in Unicode by the use of one to four bytes of eight bits each [109].

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xsd:schema
3   xmlns:sts="http://cs.york.ac.uk/ecss/examples/spacetransportationservice"
4   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
5   targetNamespace="http://cs.york.ac.uk/ecss/examples/
        spacetransportationservice">
6   <!-- ... -->
7 </xsd:schema>
```

Listing 2.1: *Document*, *encoding*, and *namespace* in XML Schema Definition.

Namespaces. Namespaces represent a mechanism to evade name clashes that occur when mixing XML documents from different XML applications with elements of the same name. Listing 2.1 lines 3–5 declare the namespaces *xmlns:sts*, i.e. the namespace of the space transportation services domain, and *xmlns:xsd*, i.e. the namespace of the W3C for the XSD version of the year 2001, and *targetNamespace*, i.e. the namespace that this XSD document is intended to target and validate.

Elements and particles. In general, elements in XSD are defined by *xsd:element*. Elements that only contain text and commonly assign the attribute *type* with value *xsd:string*, *xsd:decimal*, *xsd:integer*, *xsd:boolean*, *xsd:date*, or *xsd:time* are referred to as simple elements. Elements that assign the attribute *type* with values such as *sts:LaunchSite*, *sts:EngineType*, and *sts:Spacecraft* (lines 3–5 in Listing 2.2) are referred to as custom elements and require the definition of custom types (type definitions of *sts:LaunchSite*, *sts:EngineType*, and *sts:Spacecraft* are depicted in Appendix 1). Moreover, the attributes *minOccurs* and *maxOccurs* in an *xsd:element* illustrate examples for element particles that indicate the minimum and maximum number of times an element may be instantiated.

```
1 <xsd:complexType name="SpaceTransportationService">
2   <xsd:sequence>
3     <xsd:element maxOccurs="unbounded" minOccurs="0" name="launchSites" type="
          sts:LaunchSite"/>
4     <xsd:element maxOccurs="unbounded" minOccurs="0" name="engineTypes" type="
          sts:EngineType"/>
5     <xsd:element maxOccurs="unbounded" minOccurs="0" name="spacecrafts" type="
          sts:Spacecraft"/>
6   </xsd:sequence>
7 </xsd:complexType>
```

Listing 2.2: *Element*, *particles*, and *sequence* in XML Schema Definition.

Sequences. Elements that are enclosed by a sequence must appear in the same order as that declared (lines 2–6 in Listing 2.2). Hence, valid instances of the element *SpaceTransportationService* require instances of *LaunchSite* to occur before instances of *EngineType*, and instances of *EngineType* to occur before instances of *Spacecraft*.

Complex Types. Listing 2.3 represents the definition of the complex types *EngineType* and *NamedElement*. The former (lines 5–11) extends the latter (lines 1–3), as defined by the attribute *base* in element *xsd:extension* (line 7) and, thus, inherits the (required) attribute *name*.

```
1 <xsd:complexType abstract="true" name="NamedElement">
2   <xsd:attribute name="name" type="xsd:string" use="required"/>
3 </xsd:complexType>
4
5 <xsd:complexType name="EngineType">
6   <xsd:complexContent>
7     <xsd:extension base="sts:NamedElement">
8       <xsd:attribute name="fuelKind" type="xsd:string"/>
```

```
9     </xsd:extension>
10   </xsd:complexContent>
11 </xsd:complexType>
```

Listing 2.3: *Complex types*, *sequence*, and *attributes* in XML Schema Definition.

Attributes. Lines 5–7 in Listing 2.4 declare attributes that are capable of holding the value of *type* "xsd:string", "xsd:string", and "xsd:integer" respectively. Further, the use of an attribute may be mandatory or optional and is indicated by the attribute *use* in *xsd:attribute*. For example, line 7 indicates that the use of the attribute *relaunchCostInMioUSD* in complex type *Spacecraft* is required. Thus, valid instances of *Spacecraft* necessarily specify a value for the relaunch cost in millions of US dollars.

```
1 <xsd:complexType name="Spacecraft">
2   <xsd:complexContent>
3     <xsd:extension base="sts:NamedElement">
4       <!-- ... -->
5       <xsd:attribute name="manufacturer" type="xsd:string"/>
6       <xsd:attribute name="countryOfOrigin" type="xsd:string"/>
7       <xsd:attribute name="relaunchCostInMioUSD" type="xsd:integer" use="
            required"/>
8     </xsd:extension>
9   </xsd:complexContent>
10 </xsd:complexType>
```

Listing 2.4: *Attributes* in XML Schema Definition.

Groups. Model groups and attribute groups represent a macro mechanism for the reuse of elements and attributes, and may be created by enclosing definitions of *xsd:element* and *xsd:attribute* with *xsd:group* (lines 3–5 in Listing 2.2), and *xsd:attributeGroup* (lines 5–7 in Listing 2.4) respectively.

Simple Types. Generally, simple types define constraints on a base type. For example, Listing 2.5 defines the simple type *PhysicalPropertiesType* that may be instantiated with value "LENGTH", "WIDTH", "DIAMETER", "PERIMETER", "AREA", "VOLUME", or "MASS". In the following, a more detailed introduction on restrictions is provided.

```
1 <xsd:simpleType name="PhysicalPropertyType">
2   <xsd:restriction base="xsd:string">
3     <xsd:enumeration value="LENGTH"/>
4     <xsd:enumeration value="WIDTH"/>
5     <xsd:enumeration value="DIAMETER"/>
6     <xsd:enumeration value="PERIMETER"/>
7     <xsd:enumeration value="AREA"/>
8     <xsd:enumeration value="VOLUME"/>
9     <xsd:enumeration value="MASS"/>
10  </xsd:restriction>
11 </xsd:simpleType>
```

Listing 2.5: *Simple type* in XML Schema Definition.

2.2.3 Restrictions

The XML Schema specification introduces a mechanism for the definition of acceptable values in XML documents that are referred to as restrictions and facets [28]. In general, restrictions may be imposed on values, a set of values, a series of values, whitespace characters, length, and datatypes. In what follows, several kinds of restrictions that may be defined by an XSD document are illustrated using the example of the space transportation services domain.

Series of values. Listing 2.6 defines an element that is restricted by a series of values intended to represent country codes, as defined by ISO 3166 [108]. For instance, a valid value for the country of origin of a spacecraft is composed of exactly two uppercase letters of the English alphabet, such as "CH" for the country code of Switzerland. This constraint is realized by restricting the use of string to match the *pattern* attribute value "[A-Z][A-Z]" (line 4).

```
1 <xsd:attribute name="countryOfOrigin">
2   <xsd:simpleType>
3     <xsd:restriction base="xsd:string">
4       <xsd:pattern value="[A-Z][A-Z]"/>
5     </xsd:restriction>
6   </xsd:simpleType>
7 </xsd:attribute>
```

Listing 2.6: *Series of values* restriction in XML Schema Definition.

Values. Listing 2.7 defines a constraint on the decimal number value that is used to represent the attribute *locationLatitude*, i.e. part of the location definition of a launch site for spacecrafts. More specifically, valid values for a latitude degree, such as "-80.604333", range from "-180" to "+180" degrees (west and east coordinates of the Prime Meridian) are constrained by creating a restriction based on "xsd:decimal" (line 3), and define children *xsd:minInclusive* and *xsd:maxInclusive* with values "-180" (line 4) and "180" (line 5) respectively.

```
1 <xsd:attribute name="locationLongitude">
2   <xsd:simpleType>
3     <xsd:restriction base="xsd:decimal">
4       <xsd:minInclusive value="-180"/>
5       <xsd:maxInclusive value="180"/>
6     </xsd:restriction>
7   </xsd:simpleType>
8 </xsd:attribute>
```

Listing 2.7: *Values* restriction in XML Schema Definition.

Length. Listing 2.8 depicts the definition of the attribute *fuelKind* that is restricted to a sequence of 128 characters for the representation of propellants, such as "Liquid Oxygen

/ Rocket Propellant-1 (LOX / RP-1)". More specifically, the element *xsd:restriction* with attribute *base* value *xsd:string* and child element *xsd:length* with value "128" is defined (lines 3–5).

```
1 <xsd:attribute name="fuelKind">
2   <xsd:simpleType>
3     <xsd:restriction base="xsd:string">
4       <xsd:length value="128"/>
5     </xsd:restriction>
6   </xsd:simpleType>
7 </xsd:attribute>
```

Listing 2.8: *Length* restriction in XML Schema Definition.

Set of values. Listing 2.9 defines a restriction on a set of values that represent valid functions of a spacecraft. In particular, the element *Function* defines an *xsd:restriction* with attribute *base* value "xsd:string" and child elements *xsd:enumeration* with values referring to Mars colonization, Earth lunar transport, multiplanetary transport, intercontinental transport, and orbital launcher (lines 3–9).

```
1 <xsd:element name="Function">
2  <xsd:simpleType>
3     <xsd:restriction base="xsd:string">
4       <xsd:enumeration value="MARS_COLONIZATION"/>
5       <xsd:enumeration value="EARTH_LUNAR_TRANSPORT"/>
6       <xsd:enumeration value="MULTIPLANETARY_TRANSPORT"/>
7       <xsd:enumeration value="INTERCONTINENTAL_TRANSPORT"/>
8       <xsd:enumeration value="ORBITAL_LAUNCHER"/>
9     </xsd:restriction>
10  </xsd:simpleType>
11 </xsd:element>
```

Listing 2.9: *Set of values* restriction in XML Schema Definition.

Whitespace characters. Listing 2.10 defines a restriction on the value for the operator of a launch site for spacecrafts. In this example, the value of attribute *operator* is defined as an *xsd:simpleType* with a *xsd:restriction* that is based on type "xsd:string" and child element *xsd:whiteSpace* with value "preserve". Thus, it enables the representation of postal addresses that may contain whitespace, such as line-breaks.

```
1 <xsd:attribute name="operator">
2   <xsd:simpleType>
3     <xsd:restriction base="xsd:string">
4       <xsd:whiteSpace value="preserve"/>
5     </xsd:restriction>
6   </xsd:simpleType>
7 </xsd:attribute>
```

Listing 2.10: *Whitespace characters* restriction in XML Schema Definition.

In summary, the technical space of XMLware offers the formalization of domain-specific concepts through elements, attributes, and (restricted) types that may be introduced within XSD documents, thus capturing the structural semantics and constraints of DSLs. The notation of DSLs that are defined in the technical space of XMLware, however, is bound to the use of angle-brackets, i.e. reported to constrain human-processibility [84, 185, 199], or require the implementation and maintenance of dedicated handcrafted IDEs, which, ideally, match the capabilities offered by modern Integrated Development Environments (IDEs).

2.3 Grammarware

The technical space [139] of grammarware comprises a set of concepts, body of knowledge, tools, required skills, and possibilities in the context of *language grammar* [122]. Context-Free Grammar (CFG), class dictionaries, and tree and graph grammars represent examples of formalism and notations that are employed in the technical space of grammarware. Grammars are applied for a variety of purposes, such as the definition of the concrete and abstract syntaxes of Domain-Specific Languages (DSLs), as well as formal blueprints for the construction of language infrastructure, also referred to as grammar-dependent software. In the following sections, DSLs are briefly introduced alongside the parser generator Another Tool For Language Recognition (ANTLR) tool for the automated generation of DSL infrastructure, such as dedicated language parsers. Furthermore, the CFG formalism is introduced with a particular focus on the Extended Backus–Naur Form (EBNF) [110] and its application for the definition of the structural semantics and constraints of a DSL for the specification of space transportation services.

2.3.1 Domain-specific languages

The major difference between General-Purpose Languages (GPLs) and DSLs is that the former are broadly applicable across domains, while the latter are employed for specific domains of application. For example, Java may be utilized in the implementation of object-oriented software for a wide variety of application domains. To the contrary, the Structured English Query Language (SQL) may (only) be employed for the definition of relational database queries, and reflects how people use tables to obtain information [41]. In comparison, DSLs focus on particular domains of application and, generally, aim at higher levels of abstraction than GPLs. Moreover, the construction of a DSL is usually the outcome of a collaborative effort between language engineers and domain experts. As a result, DSLs tend to ease and speed up the development of applications for specific domains compared to the application of GPL libraries [134, 132]. Moreover, the results of empirical studies indicate that DSLs increase performance in all three groups of the cognitive dimensions framework by Blackwell et al. [30] and, particularly, in the closeness of mappings, diffuseness, error-proneness, role expressiveness, and viscosity [134].

Tool Support

Another Tool For Language Recognition (ANTLR) is a popular tool in the technical space of grammarware for the automated generation of executable finite-state machines from language grammars that are capable of recognizing grammar-conforming language instances [175]. More specifically, ANTLR generates the source code for left-to-right parsers, also referred to as *leftmost derivation* and *LL parser*, and tree parsers that enable the processing of Abstract Syntax Trees (ASTs) from Context-Free Grammars (CFGs) that follow the EBNF, also (simply) referred to as EBNF grammars [175].

2.3.2 Context-free grammar

CFG is employed for the definition of the syntax and hierarchical structure of a language. A CFG is composed of a start symbol and sets of terminal symbols, non-terminal symbols, and production rules [4]. In the following, the grammar depicted in Listing 2.11 is used to describe the concepts that appear in CFGs.

```
1 S → NP VP
2 NP → The N
3 VP → V | NP
4 V → flies | operates
5 N → Spacecraft | LaunchSite
```

Listing 2.11: Example of a context-free grammar.

A *terminal symbol* is also referred to as *token*, and represents an elementary symbol in the language defined by a CFG. For example, Listing 2.11 presents the terminal symbols "The" (line 2), "flies" and "operates" (line 4), and "Spacecraft" and "LaunchSite" (line 5).

The left-hand side of a production rule represents a *non-terminal symbol*, also referred to as *syntactic variable*. For example, Listing 2.11 presents the non-terminal symbols S, NP, VP, V, and N.

A *start symbol* is represented by a non-terminal symbol and denotes the entry point for the construction of a sentence in the language represented by a CFG. For example, line 1 in Listing 2.11 presents the start symbol S.

The *language* of a CFG is composed of all possible sentences that can be derived by a number of steps from the start symbol. Therefore, a *sentence* of a language is a sequence of terminal symbols that is derived from the CFG of the language. Stated differently, a *sentence* is composed of terminal symbols that occur in the alphabet of the language defined by the CFG. As an example, Listing 2.11 specifies a language with alphabet "The", "flies", "operates", "Spacecraft", and "LaunchSite". Although a sentence of a language may be legal, i.e. constructed by terminal symbols that occur in the language alphabet, it may not be meaningful. As an example, the grammar in Listing 2.11 is able to generate sentences that are composed of the nouns "Spacecraft" and "LaunchSite", and the verbs "operates" and "flies". Therefore, both sentences that are meaningful, such as "The LaunchSite operates" and "The Spacecraft flies", and less

meaningful, such as “The LaunchSite flies”, may be legally constructed. For example, the sentence “The Spacecraft flies” is derived as illustrated in Listing 2.12, and described by the following five steps. Firstly, start symbol S produces non-terminal symbols NP and VP. Secondly, production rule NP generates terminal symbol “The” and non-terminal symbol N. Thirdly, production rule N yields terminal symbol “Spacecraft”. Fourthly, production rule VP produces non-terminal symbol V. Fifthly, production rule V creates terminal symbol “flies”.

```
1 S → NP VP → The N VP → The Spacecraft VP → The Spacecraft V → The Spacecraft
    flies
```

Listing 2.12: Example of CFG sentence derivation.

A *production rule* is composed of a left-hand side, an arrow, and a right-hand side. The non-terminal symbol that makes up the left-hand side of a production rule is also referred to as *head*. The right-hand side of a production rule is also referred to as *body*, and may be composed of a series of terminal symbols, non-terminal symbols, and logical symbols or separators. More specifically, the right-hand side of a production rule is represented by either a sequence or selection of terminal symbols (lines 4 and 5 in Listing 2.11), a sequence or selection of non-terminal symbols (lines 1 and 3), or both (line 2) and logical symbols or separators indicating that either may be selected for the construction of a sentence, e.g. production rule V line 4 states the either terminal symbol “flies” or “operates” is produced. In case the body of a production rule states a sequence (as opposed to a selection), all symbols are necessarily applied for the derivation of a sentence. For example, line 1 defines the production rule S with a sequence composed of the production rules NP and VP.

Extended Backus–Naur Form

The Extended Backus–Naur Form (EBNF) [110], i.e. an extended version of the Backus–Naur form (BNF), is employed for expressing CFGs and is, thus, applicable to the definition of both the structural semantics and notation of DSLs. Listings 2.13 and 2.13 present an excerpt of the space transportation service language defined by the use of BNF and *Extended* BNF grammar respectively, and highlight production rules in bold. In contrast to the XMLware technical space, employing CFG requires the definition of the complete alphabet of the DSL being represented through BNF or EBNF grammar rules. For example, lines 1 and 3 in Listing 2.13 define terminal rules for *Letter* and *Digit* respectively. Similarly, lines 5 and 7 define the concept of *Character* and *String*. More specifically, a *Character* is represented by either a *Letter*, a *Digit*, an underscore, or an empty space, while a *String* is represented by a sequence of one or more *Character*s that are enclosed by either double-quotes or single-quotes. Further, the concept of *Identifier* and *Integer* is defined in lines 9 and 11 respectively. More specifically, the former begins with a *Letter*, and may be followed by a sequence of *Letter*, *Digit*, and underscore, while the latter starts with a *Digit* and may be followed by a sequence of *Digit*s.

```
1  Letter :== 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | '
       L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X'
        | 'Y' | 'Z' | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j'
       | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' |
        'w' | 'x' | 'y' | 'z' ;
2
3  Digit :== '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ;
4
5  Character :== Letter | Digit | '_' | ' ' ;
6
7  String :== '"' Character { Character } '"' | "'" Character { Character } "'"
       ;
8
9  Identifier :== Letter { Letter | Digit | '_' } ;
10
11 Integer :== Digit { Digit } ;
12
13 Spacecraft :== 'Spacecraft' Identifier
14   [ 'stages' Stage { Stage } ]
15   [ 'functions' Function { Function } ]
16   [ 'physicalProperties' PhysicalProperty { PhysicalProperty } ]
17   [ 'manufacturer' String ]
18   [ 'countryOfOrigin' String ]
19   'relaunchCostInMioUSD' Integer ;
```

Listing 2.13: Example of an original BNF grammar.

EBNF extends the meta-notation of BNF and allows the definition of shorter versions of production rules by introducing * (Kleene Star), + (Kleene Cross), question mark, and rounded parentheses for zero or more occurrences, one or more occurrences, zero or one occurrence, and grouping. For example, BNF does not offer the explicit specification of one or more occurrences of *Stage* in *Spacecraft* and, thus, requires to denote a sequence consisting of a single occurrence of *Stage* that is followed by an occurrence of zero or more occurrences of *Stage* (line 14 in Listing 2.14). EBNF offers the ability to define one or more occurrences of *Stage* by the use of the + operator and, thus, increases the conciseness of production rules (line 2 in Listing 2.14).

```
1  Spacecraft :== 'Spacecraft' Identifier
2    ( 'stages' Stage+ )?
3    ( 'functions' Function+ )?
4    ( 'physicalProperties' PhysicalProperty+ )?
5    ( 'manufacturer' String )?
6    ( 'countryOfOrigin' String )?
7    'relaunchCostInMioUSD' Integer ;
```

Listing 2.14: Spacecraft production rule employing *Extended* BNF.

2.3.3 Terminal rules

Although regular expressions present a powerful mechanism for the definition of context-free languages, their application for the specification of structural constraints within

CFGs may result in grammar rules that are complex and verbose in both construction and maintenance. For instance, a simple constraint, such as restricting the value of a character sequence to exactly two consecutive uppercase letters (cf. corresponding XMLware-based solution depicted in Listing 2.6), may be introduced by *UpperCaseLetter*, i.e. a dedicated terminal rule that defines the alphabet of uppercase letters (line 1 in Listing 2.15), and the *CountryOfOrigin* production rule, i.e. defining two consecutive occurrences of the *UpperCaseLetter* terminal rule (line 3).

```
1 UpperCaseLetter :== 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' |
      'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | '
     W' | 'X' | 'Y' | 'Z' ;
2
3 CountryOfOrigin :== UpperCaseLetter UpperCaseLetter ;
```

Listing 2.15: EBNF-based restriction of country of origin value to two uppercase letters.

Further, a restriction of the value of location latitude to decimal numbers that range from -180 to 180 may be introduced by *UnsignedDecimal*, i.e. a dedicated terminal rule for the representation of an unsigned decimal number (line 1 in Listing 2.16) and a dedicated production rule defining all possible occurrences (lines 3–11). More specifically, the former (line 1) defines an *UnsignedDecimal* to be composed of a single *Digit* that may be followed by a decimal point "." and one or more instances of *Digit*; the latter (lines 3–11) defines the optional occurrence of a negative sign "-" (line 4), i.e. indicating a negative decimal number, followed by either the occurrence of an *UnsignedDecimal* (line 5), a *Digit* and an *UnsignedDecimal*, a sequence of the *Digit* "1" followed by either the *Digit* "0", "1", "2", "3", "4", "5", "6", or "7" and an *UnsignedDecimal* (line 9), or a sequence of the *Digit* "1" that is followed by the *Digit*s "8" and "0" and (optionally) a decimal point ".", and one or more occurrences of the *Digit* "0" (line 11). In contrast, the corresponding XMLware-based solution depicted in Listing 2.7 is notably more concise.

```
1  UnsignedDecimal :== Digit ( '.' Digit+ )?
2
3  LocationLongitude :==
4  ( '-' )?
5  ( UnsignedDecimal )
6   |
7  ( Digit UnsignedDecimal )
8   |
9  ( '1' ( '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' ) UnsignedDecimal )
10  |
11 ( '1' '8' '0' ( '.' '0'+ )? )
```

Listing 2.16: EBNF-based restriction of location latitude value to decimal numbers from -180 to 180.

Further, a restriction on the length of the character sequence of *FuelKind* may only be introduced by a dedicated production rule for the representation of a character sequence that is composed of a maximum of 128 characters. Listing 2.17 illustrates the production

rule of *FuelKind* that limits the value *FuelKind* to a maximum of sixteen consecutive characters. Similarly to the restriction imposed on *country of origin*, the corresponding XMLware-based solution for the restriction of *location latitude* (cf. Listing 2.8) is (likewise) more concise.

```
1 FuelKind :== Character? Character? Character? Character? Character? Character
    ? Character? Character? Character? Character? Character? Character?
    Character? Character? Character? Character?
```

Listing 2.17: EBNF-based restriction of fuel kind value for character sequences with maximum length of sixteen characters.

Moreover, in order to support line-breaks in character sequences (required to represent the postal address within attribute *operator* of launch site), the production rule *Character* (line 5 in Listing 2.14) may be extended to support carriage return and new lines within the values of attribute *operator* (cf. Listing 2.18). The corresponding XMLware-based solution is depicted in Listing 2.10.

```
1 Character :== Letter | Digit | '_' | ' ' | \r | \n ;
```

Listing 2.18: EBNF-based extension of operator value for multi-line support.

Finally, in order to restrict the value of attribute *Function* to a set of predefined values, a production rule may be defined that specifies terminal values separated by logical or operators (cf. listing 2.19). The corresponding XMLware-based solution is depicted in Listing 2.9.

```
1 Function :== 'MARS_COLONIZATION' | 'EARTH_LUNAR_TRANSPORT' | 'MULTIPLANETARY_
    TRANSPORT' | 'INTERCONTINENTAL_TRANSPORT' | 'ORBITAL_LAUNCHER'
```

Listing 2.19: EBNF-based restriction of function value.

In summary, the technical space of grammarware enables the specification of grammar, by means of EBNF grammar rules, and thus captures the structural semantics and constraints of DSLs. Moreover, in contrast to the immutable angle-bracket notation employed by the technical space of XMLware, EBNF terminal rules allow the specification of terminal symbols or tokens that define the notation of textual DSLs. As illustrated in Section 2.3.3, the apprehension and maintenance of semantic constraints in EBNF grammars is complex and verbose in comparison to both XSD restrictions (Section 2.2.3) and textual formal constraints (Section 2.4.3) employed in the technical space of XMLware and modelware respectively.

2.4 Modelware

The technical space [139] of modelware comprises a set of concepts, a body of knowledge, tools, required skills, and possibilities in the context of *Model-Driven Engineering (MDE)* [170]. In general, the definition of modeling languages is based on the Meta Object

Facility (MOF) standard, composed of a set of layers, and the Eclipse Modeling Framework (EMF), which offers a quasi-reference implementation for the MOF standard [168]. More specifically, several Model-Driven Language Engineering (MDLE) frameworks, such as Xtext and the Graphical Modeling Framework (GMF), offer workbenches for the development of textual and graphical modeling languages, also referred to as modeling languages, and in particular the definition of the structural semantics and constraints of modeling languages.

2.4.1 Model-driven engineering

Model-Driven Engineering (MDE) is a software engineering paradigm that considers models as first-class citizens instead of exclusive artifacts of documentation. Basically, MDE is a software development approach in which models play a central role in all software engineering processes by abstracting real-world systems for specific purposes and, hence, enables focusing on different aspects that are of interest to individual stakeholders [191]. Consequently, the employment of models is rendered applicable to all engineering disciplines, as well as domains of application [54]. In comparison, the Model-Driven Development (MDD) approach, at the same level of abstraction as MDE, centers around requirements, analysis and design, and implementation disciplines [10]. One of the primary goals of applying MDE in software engineering is to enable the rigorous specification of systems instead of informal documentations. In other words, MDE focuses on *prescriptive* modeling, i.e. the specification of system design and implementation in the form of blueprints that support planning and validation prior to realization, rather than *descriptive* modeling, i.e. the documentation of system knowledge, such as requirements and domain analysis [137]. MDE enables the alleviation of platform complexity imposed by third-generation languages, such as Java and C#, and the efficient articulation of domain concepts by the application of lessons learned from the development of high-level platform and language abstractions. Moreover, in contrast to third-generation languages, i.e. based on general-purpose notations, such as *interface* and *class* in Java, the application of MDE and, in particular, the activity of metamodeling, allows to match precisely the structural semantics and notations of languages that are intended to serve a specific domain of application. In other words, MDE acts as an engineering paradigm that aims at easing the complexity imposed by the definition of structural semantics and in particular by enabling explicit expressibility of concepts and relationships that result from the design of languages targeting specific domains [191].

Modeling stack

The modeling stack is composed of four layers in the Object Management Group (OMG) Standard Modeling Stack (cf. left-hand side of Figure 2.1) that are directly supported by the Meta Object Facility (MOF) [10, 168]. In general, a model and a metamodel may be defined as an instance of the metamodel of a modeling language, and as an instance of a metamodeling language respectively. The uppermost layer in the modeling stack is represented by the *meta-metamodel* layer (M_3), offering a metamodeling language that

enables the definition of a modeling language. In MOF, the uppermost layer is instantiated from its own model and, thus, offers a solution for the recurring problem of how to set the initial metamodel in the metamodeling process. The *metamodel* layer (M_2) enables the expression of the linguistic components and relationships of a domain in the form of a metamodel, i.e. instantiating concepts that are made available by the metamodeling language (M_3), that defines the structural semantics of the modeling language being constructed (M_2). The *model* layer (M_1) allows the definition of models that define snapshots of a real-world phenomenon, i.e. part of the system being represented, by instantiating the concepts that are made available by its modeling language (M_2). The lowermost layer in the modeling stack is represented by the *real-world elements* layer (M_0) and captures snapshots of real-world phenomena, such as the execution of a software system at a specific point in time.

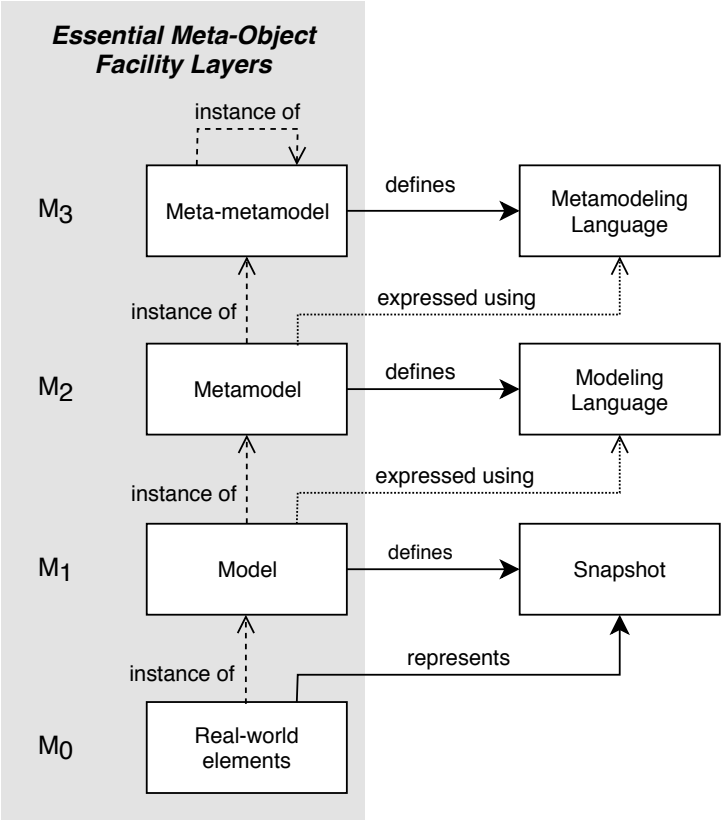

Figure 2.1: Modeling stack based on the layers of the Essential Meta-Object Facility [10, 168, 137].

2.4.2 Modeling languages

Modeling languages manifest as General-Purpose Modeling Languages (GPMLs), generic Unified Modeling Languages (UMLs) or a dedicated EMF-based language that defines domain-specific structural semantics and constraints. Gray et al. [90] define the activity of domain-specific modeling as tightly coupled with a language that is linked to the domain over which it is valid. Simply put, DSMLs represent languages that are tailored to a specific domain of application [152].

Kleppe [121] introduces DSMLs as the creation of domain-specific languages using metamodels. She highlights that the definition of a DSML requires considerable effort and, thus, finding answers on how to define such languages in a consistent, coherent, comprehensible and easy manner is of great importance. Her approach describes the creation of metamodels for use in the development of DSMLs.

According to Schmidt, DSMLs are based on technologies offered by the technical space of modelware, and enable the formalization of application structure, behavior, and requirements within particular domains through the application of metamodels for the definition of concepts and relationships among concepts that may appear in a domain, as well as key semantics and constraints that are associated with such concepts [191].

The definition of a DSML is initiated by activities for the abstraction, conceptualization, and synthesis of domain knowledge or, put simply, the identification and capturing of concepts and relationships that may manifest in a particular application domain. The conduct of these activities requires knowledge about the application domain and is, thus, usually the result of direct interaction among domain experts, namely representatives of domain knowledge, and modeling language engineers [152, 119]. The development of modeling languages is considered challenging as it requires expertise both in the domain of application, as well as the engineering of modeling languages, which is rarely manifested in practice [152].

In general, modeling languages are constructed based on a *metamodeling language* (cf. right-hand side of Figure 2.1), enabling the specification of a metamodel, also referred to as abstract syntax and Abstract Syntax Tree (AST). The metamodel of a modeling language captures the concepts and relationships that may be instantiated by a *model* of the modeling language. In other words, a metamodel is a model that defines the structural semantics of a modeling language. An *instance* that conforms to a modeling language is referred to as a model. The notation of a modeling language, also referred to as concrete syntax, defines the visual representation of the concepts and relationships captured by the metamodel. Moreover, a modeling language may define a multitude of notations to accommodate different types of stakeholders.

Tool support

Model-Driven Language Engineering (MDLE) frameworks, such as the EMF-based frameworks Xtext and Graphical Modeling Framework (GMF), facilitate the development

of modeling languages by leveraging advances in IDE technology of mainstream IDEs. In particular, Xtext employs automation, e.g. for the creation of language-specific parsers, serializers, and textual IDEs that provide basic syntax highlighting, content-assistance, folding, jump-to-declaration, and reverse reference look-up across multiple models [70, 85]. Their validation, content-assistance, and quick fix resolution capabilities, however, are limited, not only due to unavailable information, such as semantic constraints, but also due to the capacity in which available information is exploited. As an example, validators typically highlight an entire class as invalid instead of (only) the feature that violates a constraint, thus rendering the identification and resolution of errors more difficult and time-consuming. A more detailed summary of the Xtext framework is presented alongside other language workbench frameworks in Section 3.2.

Essential Meta-Object Facility and Eclipse Modeling Framework

Eclipse Modeling Framework (EMF) is the quasi-reference implementation of the Meta Object Facility (MOF) standard and, in particular, the Essential Meta Object Facility (EMOF) compliance point [168]. The EMF provides a closed and strict metamodeling architecture and code generation facility for building tools and applications based on a structured data model that is serialized in the form of an XML Metadata Interchange (XMI) document. More specifically, the EMF defines the model on the uppermost layer to conform to itself (*meta-metamodel* on layer M_3 in Figure 2.1), as well as the correspondence of every model element with a model element on the layer above (model elements on layers M_1–M_3) respectively.

In the EMF, an Ecore model—also referred to as EMF-based metamodel or AST—corresponds to layer M_2 in MOF (cf. Figure 2.1), instantiates the reflexive meta-metamodel on layer M_3, and enables the capturing of the concepts, properties, and relationships that are components of a real-world system (*real-world elements* in Figure 2.1) and, in particular, the structural semantics of modeling languages. Further, the layer M_1 in the EMOF represents instances specifying actual values for concepts, properties, and relationships, as defined in their corresponding Ecore model, i.e. located on layer M_2. The principal components of the Ecore metamodeling language are illustrated in Figure 2.2 and are described below.

In general, the concepts of class, attribute, and reference, known from the Object-Oriented Programming (OOP) paradigm, are referred to as *EClass*, *EAttribute*, and *EReference* in the Ecore metamodeling language, located on layer M_3, respectively. The abstract class *EModelElement* may define a multiple cardinality-reference with name *eAnnotations* and type *EAnnotation* for the indication of annotations, such as manually applicable language extensions, and is extended by the classes *EAnnotation* and *ENamedElement*. The abstract class *ENamedElement* is extended by the classes *EPackage*, *EClassifier*, and *ETypeElement*, and defines an attribute with name *name* and type *EString*. The class *EPackage* represents the root element in Ecore metamodels and may contain a multitude of instances of type *EClassifier*, i.e. indicated by the multiple cardinality containment-reference *eClassifier*. It also defines an attribute with

name *nsURI* and type *EString* indicating the Uniform Resource Identifier (URI) of a package namespace. The abstract class *EClassifier* is extended by the classes *EClass* and *EDatatype*. The class *EClass* defines an attribute with name *abstract* and type *EBoolean* for the indication of abstract classes, a multiple cardinality-reference with name *eSuperTypes* and type *EClass* for the indication of inheritance, and a multiple cardinality containment-reference with name *eStructuralFeatures* and type *EStructuralFeature* for the indication of the structure of an attribute or reference. The abstract class *ETypeElement* is extended by the abstract class *EStructuralFeature*, i.e. extended by the classes *EReference* and *EAttribute* respectively. The class *EAttribute* defines a single cardinality-reference with name *eAttributeType* and type *EDatatype*. The class *EReference* defines an attribute with name *containment* and type *EBoolean* indicating containment references, an optional single cardinality-reference with name *eOpposite* and type *EReference*, and a single cardinality-reference with name *eReferenceType* and type *EClass*.

Figure 2.3 illustrates the Ecore-based structural semantics specification of the space transportation services language. More specifically, the principal components of Ecore,

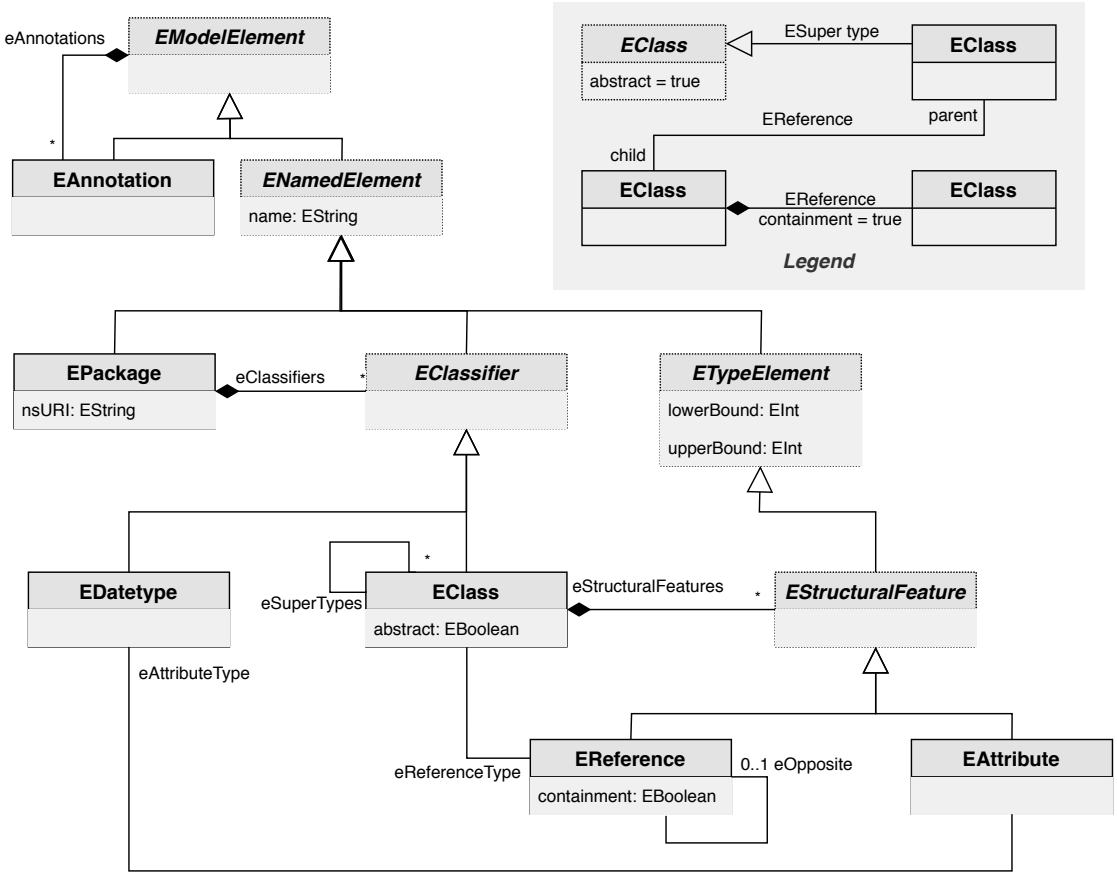

Figure 2.2: Principal components of the Ecore language metamodel.

such as (abstract) classes, attributes, and (containment) references, are instantiated by the following concepts and relationships in the space transportation service language definition (cf. Figure 2.3): a class with name *SpaceTransportationService* defines multiple cardinality containment-references for launch sites, launch schedule, engine types, and spacecrafts; an abstract class with name *NamedElement* that is extended by the classes that define space transportation services, launch sites, engine types, and spacecrafts; a class with name *EngineType* that defines the attribute *FuelKind* with type *EString*; a class with name *LaunchSite* that defines the attributes *locationLatitude*, *locationLongitude*, *operator*, *numberOfLaunchpads*, and *operational* with types *EDouble*, *EDouble*, *EString*, *EInt*, and *EBoolean* respectively, as well as the multiple cardinality-reference with name *physicalProperties* and type *PhysicalProperty*; a class with name *Spacecraft* that defines the attributes *functions*, *manufacturer*, *countryOfOrigin*, and (mandatory) *relaunchCostInMioUSD* with types (enumeration) *Function*, *EString*, *EString*, and *EInt* respectively, as well as the multiple cardinality-references named *stages*, *launchSites* (i.e. sites for which the spacecraft is certified for launch), and *physicalProperties* and type *Stage*, *LaunchSite*, and *PhysicalProperty* respectively; a class with name *LaunchSchedule* that defines the multiple cardinality containment-reference *launchEvents* for launch events; a class with name *LaunchEvent* that defines the attributes *missionTitle* and *startDateTime* with types *EString* and *EDate* respectively, as well as the single cardinality-references with names *spacecraft* and *launchSite*.

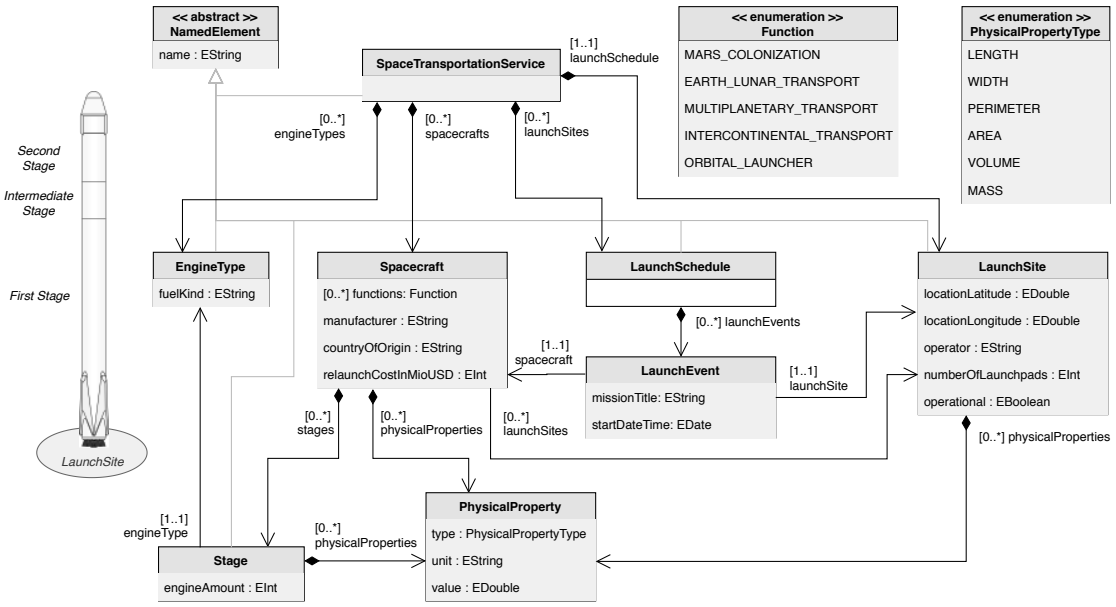

Figure 2.3: Ecore metamodel defining the structural semantics of the space transportation service modeling language.

2.4.3 Structural constraints

The standardized Object Constraint Language (OCL) [40] represents a textual formal constraint language that is widely used to increase precision in the structural semantics of modeling languages with unambiguous and rigorous constraints. Therefore, OCL complements the limited expressiveness of modeling languages. OCL constraints are captured as invariants that employ a typed, declarative, and side-effect free specification language that is based on first-order predicate logic, and offers navigation and querying facilities for (meta)models. *Typed* refers to OCL expressions that evaluate to a predefined OCL type or a type defined in the model, where the OCL expression is applied, and must conform to the rules and operations of that type. *Declarative* implies that OCL excludes the ability to define imperative constructs, such as assignments. *Side-effect free* means that OCL expressions may constrain or query and may *not* change the state of a system. *Specification* implies that the OCL language definition does not include implementation guidelines and implementation details. Moreover, OCL has been designed as a compact language that enables the supplementation of information in various forms through concise expressions. As a result, OCL enables the definition of: invariants that state all necessary conditions that must be satisfied in a valid language instantiation; query operations; initialization of properties of a class; derivation rules expressing how the value of derived model elements are computed; and operation contracts that enable the imposition of pre- and post-conditions to operations.

For example, a constraint that restricts the value of the attribute *countryOfOrigin* to two consecutive uppercase letters may be introduced by the invariant *twoUpperCaseChars* (lines 1–3 in Listing 2.20), i.e. restricting the length of the value of *countryOfOrigin* to a size of exactly two (line 2), and evaluating the equality of the uppercase-value of *countryOfOrigin* (left-hand side of equation in line 3) with the value of *countryOfOrigin* (cf. right-hand side of equation in line 3). The corresponding XMLware and grammarware solution for the restriction of attribute *countryOfOrigin* is depicted in Listings 2.6 and 2.15 respectively.

```
1 invariant twoUpperCaseChars:
2    countryOfOrigin.size() = 2 and
3    countryOfOrigin.toUpperCase() = countryOfOrigin;
4
5 invariant angleDecimal:
6    locationLatitude >= -180 and locationLatitude <= 180;
7
8 invariant maxLength:
9    fuelKind.size() <= 128;
```

Listing 2.20: OCL invariants defining the structural constraints of the space transportation service modeling language.

Moreover, a constraint that restricts the value of the attribute *locationLatitude* to decimal numbers with value between -180 and 180 may be introduced by the invariant *angleDecimal* (lines 5–6 in Listing 2.20), i.e. restricting the value of *locationLatitude* to

be both *equal or greater than -180* and *less or equal to 180*. The corresponding XMLware and grammarware solution for the restriction of attribute *locationLatitude* is depicted in Listings 2.7 and 2.16 respectively.

Further, a constraint on the length of a character sequence, such as the value of *fuelKind*, may be introduced by the invariant *maxLength* (lines 8–9 in Listing 2.20), i.e. restricting the length of the *FuelKind* attribute to a size of *less than or equal to 128*. The corresponding XMLware and grammarware solution for the restriction of attribute *fuelKind* is depicted in Listings 2.8 and 2.17 respectively.

In order to provide support for line-breaks in character sequences, i.e. required to represent the postal address within the attribute *operator*, the EMF employs an HTML encoded line feed character for line-break representation within instances of *EString*. Therefore, no additional invariant is required to fulfill the requirement of supporting line-breaks in character sequences. When employing textual language workbench frameworks, however, such as Xtext, which build on both the technical spaces of modelware and grammarware, the limitations of grammarware prevail. The corresponding XMLware and grammarware solution for the restriction of attribute *operator* is depicted in Listings 2.10 and 2.18 respectively.

Finally, the restriction of values of type *Function* to a predefined set of character sequences is realized by the use of enumerations (cf. top-right rectangle in Figure 2.3). The corresponding XMLware and grammarware solution for the restriction of the value of type *Function* is depicted in Listings 2.9 and 2.19 respectively.

In summary, the technical space of modelware enables language engineers to capture the structural semantics and constraints in metamodels and formal constraints respectively, thus facilitating the definition of DSMLs. Although textual language workbench frameworks, such as Xtext, facilitate the development of DSMLs by automating the generation of language infrastructure, such as IDEs, the capabilities provided by validation, content-assistance, and quick fix resolution are limited despite the availability of concise domain-specific structural constraints that are formulated through OCL invariants. More specifically, several shortcomings of OCL are reported in the literature and include: poor support for user feedback; no support for warnings; no support for dependent constraints; limited flexibility in context definition; and no support for repairing invariants [126]. Furthermore, the popular language workbench Xtext builds upon both the technical spaces of modelware and, in particular, the EMF, while grammarware does not provide mechanisms for the design and reuse of textual notations that are applicable to arbitrary domains of application.

2.5 Summary and comparison of technical spaces

The OMG Standard Modeling Stack is composed of four layers, i.e. the meta-metamodel layer M_3, the metamodel layer M_2, the model layer M_1, and the real-world layer M_0. In what follows, the modeling stack is employed to illustrate the location of individual

artifacts of the technical spaces of *XMLware*, *modelware*, and *grammarware*, as well as the correspondence among them (cf. Figure 2.4).

The system is located in the real world (layer M_0) and represents a snapshot of the artifact represented by layer M_1 at a particular point in time.

Layer M_1 is captured by a *document*, *model*, and *program* in XMLware, modelware, and grammarware respectively. More specifically, a *document* in the technical space of XMLware is represented by an XML document, i.e. instantiating elements of a particular *schema* located in layer M_2. Similarly, a *model* in the technical space of modelware may be represented by an XML Metadata Interchange (XMI) document, i.e. instantiating elements defined by a *metamodel* located in layer M_2. Likewise, a *program* in the technical space of grammarware is represented by a textual document, i.e. instantiating elements defined by a particular *grammar* located in layer M_2.

Layer M_2 is represented by a *schema*, *metamodel*, and *grammar* in XMLware, modelware, and grammarware respectively. More specifically, a *schema* in the technical

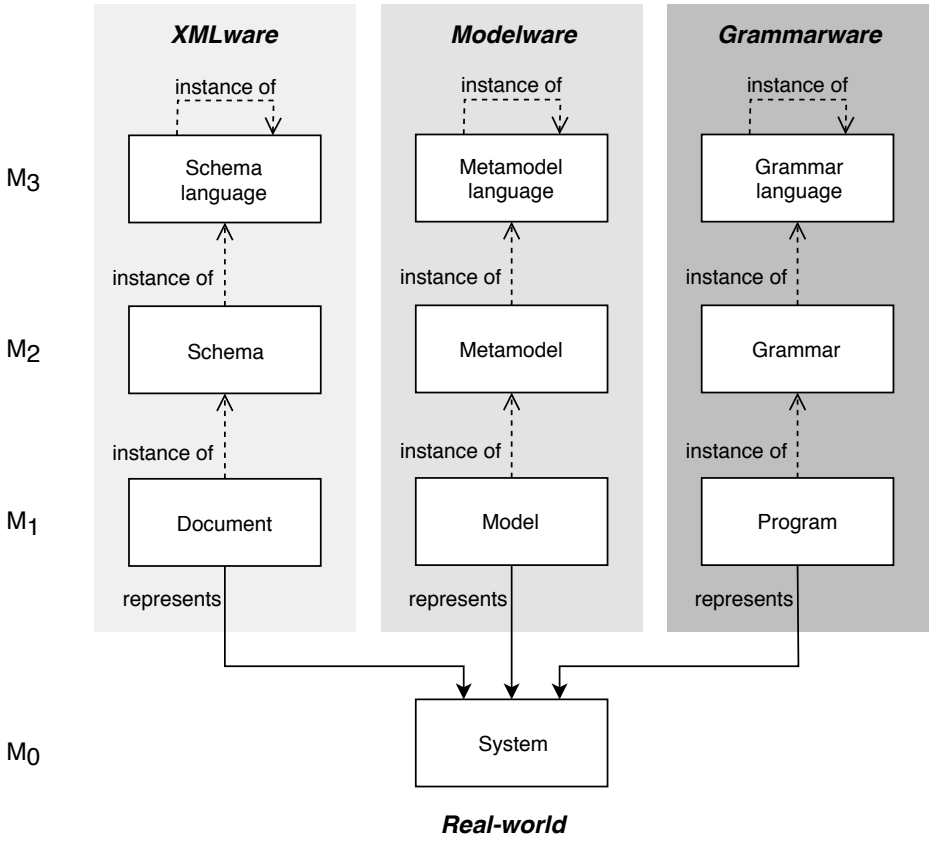

Figure 2.4: Comparison of technical spaces XMLware, modelware, and grammarware on the four-layer modeling stack (cf. Figure 2.1).

space of XMLware is represented by an XML Schema Definition (XSD) document, i.e. instantiating elements of a particular *schema language* located on layer M_3. Similarly, a *metamodel* in the technical space of modelware may be represented by an Ecore model, i.e. instantiating elements defined by the Ecore *metamodel language* located in layer M_3. Likewise, a *grammar* in the technical space of grammarware may be represented by a EBNF grammar, i.e. instantiating elements defined by a *grammar language*, such as the Xtext grammar language [22], located in layer M_3.

Layer M_3 is represented by a *schema language*, *metamodel language*, and *grammar language* in XMLware, modelware, and grammarware respectively. More specifically, a *schema language* in the technical space of XMLware is represented by the XSD Specification, i.e. reflexively instantiating elements defined in the XSD Specification [202]. Similarly, a *metamodel language* in the technical space of modelware may be represented by the Ecore metamodeling language, i.e. reflexively instantiating elements defined in the Essential Meta Object Facility (EMOF) specification [168]. Likewise, a *grammar language* in the technical space of grammarware may be represented by an EBNF grammar, such as Xtext grammar, i.e. reflexively instantiating elements defined in the Xtext grammar language.

	XMLware	**Modelware**	**Grammarware**
Methodology	Semi-structured markup	EMOF	CFG
Structural semantics	XSD document	Ecore model	EBNF grammar
Structural constraints	XSD restrictions	OCL invariants	Terminal rules
Notation	Angle-bracket	Graphical, textual, or hybrid	Textual
Tooling	Generic or dedicated XML-based IDE	Reflective, generated default, or dedicated Eclipse-based IDE	ANTLR-based lexer and parser

Table 2.1: Comparison of technical spaces XMLware, modelware, and grammarware.

Table 2.1 summarizes the technical spaces of XMLware, modelware, and grammarware presented in Sections 2.2, 2.3, and 2.4 respectively. The methodology in the technical space of XMLware, modelware, and grammarware is based on semi-structured element and attribute markup, the EMOF specification [168], and CFG respectively. The specification of structural semantics for a DSL of the technical space of XMLware, modelware, and grammarware is performed by the use of XSD documents (Section 2.2.2), the Ecore models (Section 2.4.2), and EBNF grammar (Section 2.3.2) respectively. Structural constraints for a DSL in the technical space of XMLware, modelware, and grammarware are defined by the use of XSD restrictions (Section 2.2.3), OCL invariants (Section 2.4.3), and terminal rules (Section 2.3.3) respectively. The notation for a DSL in the technical space of XMLware, modelware, and grammarware is defined by immutable angle-bracket

syntax, graphical, textual, or hybrid syntax, and textual EBNF grammar respectively. The tooling offered for DSLs that manifest in the technical space of XMLware, modelware, and grammarware may include a generic XML-based or dedicated handcrafted editor, a reflective, generated (default) or dedicated handcrafted EMF-based editor, and an ANTLR-generated lexer and parser respectively.

In summary, XMLware, grammarware, and modelware represent the technical spaces that comprise markup languages, language grammars, and MDE respectively. XMLware offers technologies for the representation, exchange, and validation of structured and semi-structured data. The immutable syntax-limitation of XML renders the technical space of XMLware unable to cope with notations that deviate from the use of angle-brackets and is, thus, rendered inapplicable to the definition of DSLs with customized notations, such as the Human-Usable Textual Notation (HUTN) [93]. Technologies offered by modelware and, in particular, for the construction of modeling languages, intend to overcome the shortcomings of high-level GPLs in regards to platform complexity and the effectiveness of expressing domain concepts [191]. The EBNF, as well as the EMOF, represent metasyntax notations employed for the definition of meta-languages, such as the *Xtext grammar language* and the *Ecore language*. Next, meta-languages that are based on EBNF and EMOF are employed for the definition of DSLs and modeling languages respectively.

Moreover, although the combination of Ecore and OCL by the EMF enables modeling language engineers to capture both structural semantics, as well as constraints, the limitations of OCL restrict generated IDEs from providing accurate validation, sensible content-assist, and low-cost quick fix solutions. Furthermore, popular frameworks, such as the EMF and Xtext, do not provide mechanisms for the design and reuse of textual notations that are applicable to arbitrary domains of application.

The following three chapters of this document present the contributions of this thesis to the above-summarized problems. More specifically, Chapter 4 presents an approach for the automated migration of textual modeling language implementation from XSDs based on the design pattern of language exploitation. Chapter 5 presents an approach for the automated generation of consistency-achieving DSML implementations from structurally constrained metamodels and, in particular, Ecore metamodels that are refined by OCL invariants. Chapter 6 presents an approach for the development and application of textual language notations that are structure-agnostic, structure-dependent, or a combination thereof.

CHAPTER 3

Related work

This chapter outlines literature related to the contributions of this thesis and in particular work on the bridging of technical spaces, language workbenches and the engineering of languages, and the design of textual notations. It also includes a brief overview of language composition and model management in general. Note that although some approaches may only appear in a specific section of this chapter, they may present work that overlaps with other sections of this chapter. The differences and commonalities between individual works presented in the literature and the proposed work on the automated generation of modeling language grammars from XML Schema-based languages, the automated generation of SBSE-driven domain-specific IDEs, and domain structure-agnostic reusable textual notations for modeling languages are analyzed in Sections 4.8, 5.7, and 6.7, respectively.

3.1 Bridges between technical spaces

In general, the bridging of technical spaces for the development and maintenance of modeling languages is typically initiated by either employing the model-first approach or the grammar-first approach. In the former, a modeling language is manifested through the creation of a domain-specific language metamodel, i.e. capturing the concepts, attributes, and relationships in the form of an abstract syntax representation, and the generation of domain-specific language grammar, i.e. interlacing the domain-specific language model with a predefined textual representation that acts as the concrete syntax representation of the language being constructed. In the latter, a modeling language is manifested by the creation of domain-specific language grammar that is followed by the derivation of a domain-specific language model. Next, in both the model-first and grammar-first approaches, an executable modeling language implementation may be generated by employing a language workbench framework (cf. Section 3.2).

In the following, related work on the bridging of technical spaces is presented by categorizing existing approaches depending on their source technical space, i.e. XMLware, grammarware, and modelware, or, in other words, the technical space to which an approach may be applied. An analysis that compares the proposed approach with existing work is presented in Section 4.8.

3.1.1 XMLware

XMLware to modelware Moreover, several approaches for realizing the engineering of XMLware to modelware [189, 147] exist. Schauerhuber et al. [189] present an approach on the mining of metamodels from existing language specifications and, in particular, XML Document Type Definitions (DTDs) that is based on a semi-automatic model transformation and MOF-based metamodels. The model transformation is (only) semi-automatic if it is split into an automatic step that establishes an initial version of the target metamodel based on unambiguous concept-correspondences between the DTD and MOF, and a manual step that forces the developer to validate and eventually refactor the initial metamodel. Their approach is showcased in the creation of a metamodel for the web modeling language WebML from the accompanying tool WebRatio that employs DTD-based language specifications of WebML and Extensible Stylesheet Language Transformation (XSD)-based model-to-code transformations for code generation. The evaluation of this approach highlights that completeness is achieved, i.e. enabling the transformation of WebML DTD-conforming XML instances to WebML Ecore metamodel-conforming models, without losses. Further, quality in terms of expressiveness, accuracy and understandability is improved by the resolution of unfit cardinalities, the introduction of package structure, inheritance and roles, and the partial resolution of IDREF-typed XML attributes into native references in Modelware. The limitations of this approach include but are not limited to the requirement of manually refactoring generated metamodels, as well as tooling implementations that rely on heuristics and incomplete solutions.

From XML and UML to Xtext and GMF Eysholdt and Rupprecht present a report [71] on the migration of a modeling environment from XSD and UML to Xtext and GMF. The context of their report is represented by 150 software developers employed within several business units that include the German pharmacies clearing house, the clearing and IT for German health care professionals, and pharmacy administration systems. Their goal is to address the inefficiency of a legacy modeling environment by migration to a modernized modeling environment. More specifically, XML-based modeling is described as inefficient due to the verbose syntax of XML and the lack of tool support, for example, customized validation, content-assist for references to model elements and files, and assistance with the modeling of textual expressions that involve arithmetic and logical terms. In particular, five XSD and UML-based languages have been migrated to five Xtext and GMF-based languages with textual and graphical IDEs that operate on the same textual model, referred to as “Xtext-models” in their work. In

other words, changes made by employing the graphical IDE are persisted by the serializer of the textual modeling language. Their migration of XSD-based languages to textual modeling languages is described as guided by a series of steps that include replacement of angle-bracket tags with tag-name, followed by (if available) a tag-identifier and curly braces, removal of remaining angle-brackets, removal of XML directives and metadata such as namespaces declarations, determination and verification of default values for attribute and element values, removal of XML attributes and elements with values equal to derived default values, introduction of import statements to improve conciseness, e.g. by the use of Java class names, restructuring where reasonable, e.g. consolidation of separate listings for table columns and class attributes into class attributes containing optional column specifications, and implementation of a well-readable, writable, and understandable textual modeling language notation. Moreover, two Xtend [22] model-to-model transformations have been implemented to enable re-use of a generator that produces code from XML files, and more precisely from serialized textual models to XML documents, referred to as "XML-models" in their work. The completeness and correctness of these model-to-model transformations have been evaluated by round-trip testing, and to be more specific, a sequence of steps that involve loading, transforming, and serializing (original) XML documents as textual models; loading, transforming, and persisting textual models as (resulting) XML documents; and comparison of original and resulting XML documents on relevant differences. In conclusion, it is reported that the productivity issues of the legacy modeling infrastructure have been solved by the employment of a new modeling infrastructure and, in particular, by reducing the time span to generate code from models from minutes to seconds, model validation and error or warning notifications that occur before code generation, automatically executed validation rules, powerful model editing assistance, such as content assist, find-references, open model elements, and navigation, and textual and graphical model out-of-sync risk elimination by use of a single serializer instantiated by both the textual and graphical modeling language.

XML to annotated Java class translation Chodarev et al. [43, 44] present an approach for the development of a translator between XML and a DSL with customized concrete syntax. The motivation of their approach is to address the challenges, complications and time expenditure, associated with the re-implementation of languages that are already formally defined, such as XML-based languages. In summary, their solution approaches these challenges through the development of a translator that is both capable of reading documents in a notation that is different from the notation of the source language, as well as write documents in the notation of the source language for the purpose of reusing source language tooling. Their approach is composed of a series of steps that include the application of the Java Architecture for XML Binding (JAXB) framework [79] and, in particular, the JAXB XML binding compiler for the transformation of XSD documents to Java classes, the extension of generated Java classes with Yet Another Java Compiler compiler (YAJCo) [179] annotations, the use of the YAJCo tool to generate a parser and pretty-printer for the translation of XML documents to annotated Java

classes, and the use of the YAJCo tool and an intermediary representation to generate a parser and pretty-printer for annotated Java classes to instances with a different textual notation. Their approach targets Java developers, as the extracted language structure is represented by Java classes and the notations are defined by Java class annotations.

3.1.2 Grammarware

Grammar-driven approaches [218, 138, 111, 39] exist, in which metamodels are generated out of existing grammar definitions.

Grammar-first: EBNF to MOF Bridge Wimmer et al. [218] propose a generic bridge between the technical spaces of grammarware and modelware that is based on a formally defined relationship between Context-Free Grammars (CFGs) and Meta Object Facility (MOF)-based metamodels presented by Analen et al. [5]. More specifically, their approach presents a semi-automated bridge between individual EBNF-based grammars and corresponding sentences, and individual MOF-based metamodels and corresponding models respectively. This approach is motivated by the need to make the interoperability between the technical spaces of grammarware and modelware a less tedious task. In brief, the approach proposes to generate automatically a metamodel from EBNF-based grammar, and then optimize the resulting metamodel. The latter is semi-automatically refined by employing model transformations automatically and metamodel annotations manually. As a result, a bridge is created that enables the transformation of sentences to corresponding models that conform to an EBNF-based grammar and corresponding MOF-based metamodel respectively. More specifically, their approach operates on layers M_1 to M_3 (cf. Figure 2.4) and utilizes the reflexivity property of EBNF for the construction of an attributed grammar that both implements an individual EBNF-based language grammar, as well as transformation rules that represent the correspondences between EBNF and MOF. Subsequently, two parsers are automatically generated that are capable of parsing an EBNF-based language grammar and conforming sentences to initial versions (referred to as "raw" in [218]) of metamodel and models respectively. Next, optimized versions (referred to as "condensed" in [218]) of metamodel and models are created by the automated execution of transformation rules to eliminate undesired properties, such as some anonymous classes that have been introduced by the initial grammar-to-metamodel transformation. Afterwards, language engineers annotate these optimized metamodels attaching additional semantics, such as data types, and employ a transformation that transforms them into final versions of metamodels (referred to as "customized" in [218]). Their work builds upon Analen et al. [5] by additionally taking layer M_1 into account or, in other words, the transformation of individual EBNF-conforming sentences to individual MOF-conforming models.

Grammar-first: EBNF to MOF Bridge by elimination of non-semantic information Kunert et al. [138] present a similar approach to Wimmer et al. [218] that (also) employs annotations. Additional semantics, however, are supplemented by grammar

annotations, as opposed to metamodel annotations. Consequently, initially generated metamodels are not refined by the attachment of additional semantics and, therefore, the loss of information by re-generating metamodels, i.e. resulting in overwritten metamodels, is eliminated. Additionally, metamodel quality is based on its applicability for a particular purpose that is characterized by its level and efficient use of abstraction and, in particular, the elimination of syntactical information, such as terminal symbols and identifiers, and its ability to be employed for the transformation of conforming models to models that conform to another metamodel. The hypothesis of their approach is that the elimination of non-semantic information from metamodels that are generated from grammars provides a vantage point for the application of model transformations. Although their approach does not attach additional information to the metamodel that is generated from grammar, refactored versions of the metamodel and conforming models are created by the application of a metamodel and model transformation respectively.

Gra2MoL J. Cánovas et al. presented Grammar-to-Model Language (Gra2MoL) [111]—a model-driven software evolution or modernization approach that supports bridging grammarware and modelware through the application of a dedicated rule-based transformation language and engine. The process of model-driven software evolution is described as requiring the extraction of models from legacy software that is represented by source code, i.e. conforming to a particular programming language. Therefore, the construction of a bridge from grammarware to modelware is motivated by scenarios that involve the evolution of source code. Gra2MoL realizes such evolution scenarios by extracting models, i.e. conforming to a target metamodel, from source code, i.e. conforming to the grammar of a programming language, as defined and executed by a Gra2MoL transformation model and execution engine respectively. The primary difference of Gra2MoL and existing model-to-model transformation languages, such as ATLAS Transformation Language (ATL) [114] and Epsilon Transformation Language (ETL) [125] is that Gra2MoL was specifically designed for the transformation of programming language source code to models. More precisely, the source element in a Gra2MoL transformation rule is represented by a grammar element, as opposed to a source metamodel element, as in ATL and ETL transformation rules. From an implementation perspective, however, Gra2MoL employs modelware to capture the syntax tree of the parsed source code through a metamodel that consists of the concepts *Tree*, i.e. the root node of the syntax tree; *Leaf*, i.e. a terminal symbol; and *Node*, i.e. a non-terminal symbol composed of one or many children nodes. Further, the correspondence between metamodel elements, i.e. representing the syntax tree, and symbols in grammar definitions includes metaclass and non-terminal, primitive type and terminal, attribute and terminal in the right-hand side of a rule, and containment reference and non-terminal in the right-hand side of a rule.

From Grammars to Accurate Metamodels Butting et al. [39] presented an approach for the translation of MontiCore [135] grammars, i.e. EBNF-like grammars, to restricted Ecore metamodels. In brief, they realize the generation of Ecore metamodels from MontiCore grammars by translating CFGs to Java classes, and generating a parser

to transform textual models to corresponding instances of these Java classes. Further, the translation of textual models, i.e. models conforming to a MontiCore grammar, to instances of a (corresponding) Ecore metamodel is realized by leveraging the resource serialization-capability provided by the EMF. Implicit cardinalities that are imposed by grammar rules are translated into linear equation systems and embedded into metamodels. Model validation is performed by employing individual constraint solvers that are generated for each class in a given metamodel. Thus, the bridging of grammarware and modelware is facilitated by automating the specification of restrictions for (naive) metamodels that are generated from grammars. The problem that is being tackled by this approach relates, on the one hand, to the lack of integration of grammars with Ecore that would allow access to a rich set of existing tooling that is available to Ecore-based languages and, on the other, to the availability of validation by means of parsers, i.e. generated from grammars, that requires additional effort to be realized in the technical space of modelware, especially for the implication of cardinality constraints on Ecore-based languages. More specifically, solving these problems would allow the semi-automated generation of potentially complete prototypes, such as Sirius [211] and Xtext [70] IDEs, that otherwise requires significant effort due to the translation of grammars to metamodels, as well as the development of IDE implementations from scratch. The approach presented by Butting et al. translates grammar rules to metamodel concepts and, in particular, MontiCore-extended CFGs, covering the complete EBNF, to Ecore metamodels that are augmented with OCL constraints. More specifically, non-terminal definitions are transformed into equally named concepts and associations in Ecore metamodels; cardinalities, such as arbitrary occurrences, non-empty occurrences, and optional occurrences, are transformed into implicitly expressed cardinalities in MontiCore; disjunctions, i.e. requiring the selection of exactly one alternative, are transformed into exclusive disjunctions in MontiCore, i.e. only allowing the instantiation of one alternative but not both; and non-terminal references are transformed into non-terminal references that allow the creation of associations to classes in MontiCore, i.e. instead of referencing a generic type such as a String.

3.1.3 Modelware

Early approaches for realizing the engineering of modelware to XMLware include those presented by Bird et al. [27] and Conrad et al. [47].

Bird et al. [27] presented work on the use of Object-role modeling (ORM) (a graphical and conceptual modeling technique) to ease the design of XSDs. Note that this work has been published before the release of the Eclipse Modeling Framework (EMF). The ORM approach employs objects and roles to capture the semantics of a universe of discourse, as well as the specification of constraints, such as uniqueness, subset, mandatory role, and frequency. Their approach entails an algorithm that enables generating XSDs from ORM conceptual data models. The authors report that they discovered a variety of different solutions to realize the mapping from ORM to XSD, and outline their solution as aiming to increase the connectivity of resulting XML documents and reduce data redundancy.

ORM has been selected as a candidate to design XSDs due to: its modeling support that is more advanced in comparison to that provided by UML and ER; its linguistic basis and role-based notation that allows model validation with domain experts by sample populations and natural verbalization; and its stability in comparison to attribute-based approaches.

Conrad et al. [47] presented work on the conceptual modeling of DTDs using UML as the connecting link between software engineering and document design, by describing a transformation from UML class diagrams, i.e. the static part of UML, to DTDs. Therefore, their work is challenged by determining a suitable mapping that reflects the semantics of the UML specification in terms of a DTD. The realization of their approach required the extension of UML and presents an early endeavour to bridge the gap between UML-based object-oriented software design and DTD.

In addition, more recent metamodel-driven approaches that generate grammars out of existing metamodels, such as EMFText [101] and HUTN [159], or link metamodels with grammars, such as TCS [115], are introduced in Section 3.2.

Summary In conclusion, this section presented bridges between the technical spaces of XMLware, modelware, and grammarware, including XMLware to modelware (Schauerhuber et al. [189]), a case study-based migration of XML and UML to Xtext and GMF (Eysholdt et al. [71]), XML to annotated Java class translation (Chodarev et al. [43, 44]), annotation-based EBNF to MOF transformation (Wimmer et al. [218] and Kunert et al. [138]), rule-based transformation (Gra2Mol [111]), translation of MontiCore (i.e. EBNF-like) grammars to restricted Ecore metamodels (Butting et al. [39]), and two early approaches in the transformation of modelware to XMLware (Bird et al. [27] and Conrad et al. [47]).

3.2 Language workbenches

This section presents language workbenches in chronological order of appearance.[1] Language workbenches are represented by frameworks and tools that enable the engineering of modeling language implementations that may include a range of handcrafted and/or (semi)automatically generated features, such as dedicated IDEs, validation, content-assist, and model repair solutions [84]. Section 5.7 presents an analysis of the proposed contribution on the automated generation of consistency-achieving DSML implementations with a selection of language workbenches that are summarized in the following.

MetaEdit+ Smolander et al. presented MetaEdit+ [195, 206]—a repository-based tool for the creation of modeling languages and code generators. MetaEdit+ originally emerged as a metaCASE research prototype that was developed between 1988 and 1995, and commercialized by the company MetaCase in 1993. The definition of modeling

[1] Note that some works were published at the same time or very near to that.

languages in MetaEdit+ is based on the dedicated metamodeling language Graph-Object-Property-Port-Role-Relationship (GOPPRR) [195] that is applied to capture the design of metamodels and model instances. More specifically, GOPPRR-based metamodels may define objects, properties, relationships, and roles such as entities in entity-relationship diagrams, textual labels in diagrams, lines between shapes in diagrams, and directed arrowheads indicating the end of a data flow relationship. Moreover, the application of a modeling language is restricted by rules and constraints that are realized as bindings, i.e. describing valid combinations between connection typos and object types by the use of objects, roles, relationships, and ports, default values, and regular expressions, i.e. validating input values. Models in MetaEdit+ may be accessed by a set of IDEs and browsers, and include four types of representations: graphical diagrams, matrices, tables, and tree views. Further, the concepts of object, property, role, relationship, and port may be defined as graphical symbols through the MetaEdit+ Symbol Editor. The MetaEdit+ tool architecture includes an object-oriented repository system for concurrent multi-user access based on a transaction and locking system. The evolution of metamodels in MetaEdit+ is handled by the specification of model transformations that update models that conform to previous versions of the metamodel and is realized either by operation on the API or the definition of XML-based transformations. MetaEdit+ code generation definitions are created either by the use of an IDE that employs a dedicated scripting language, a web services API, or intermediate files.

Intentional Software Simonyi et al. presented the *Domain Workbench* approach [194] that is based on the notion of structured IDEs, structured source code, and generative programming for the purpose of establishing *Intentional Software*, i.e. complex system intentions that are encoded and maintained by the application of high-level domain vocabulary, relationships, and rules. The early goals of combining structured source code and structured IDEs were to improve compiler performance and support the implementation of syntax-directed IDEs. In contrast, in regards to the latter combination, the goal of the Domain Workbench is to separate the syntax from the domain and, thus, allow notational flexibility. The Domain Workbench IDE applies the What You See Is What You Get (WYSIWYG) technique to software sources, and represents software in an intentional tree that is separated from the underlying representation of generated software such as source code [38]. Generative programming represents a technique in which domain(-specific) code is provided to a generator for producing target code, i.e. the final program code that is either compiled or interpreted, and represents the behavior intended by the domain code. Thus, generative programming primarily focuses on the implementation of code generators that produce target code from parameterized domain code and increase efficiency by domain code maintenance and generator re-execution. The process of establishing intentional software entails a *domain schema* that acts as an interface between the domain workbench and a generator, and is implemented and maintained by both domain experts and programmers; *domain code* that is maintained by domain experts; and a *generator* that is implemented and maintained by programmers and defines the processing of domain code to target code. As a result, the generator

implementation represents a manifestation of engineering design choices, platforms, and code patterns.

Rascal Klint et al. presented Rascal [123, 16]—a DSL and IDE for meta-programming that merges existing concepts and language features for the analysis and manipulation of source code, the reverse- and re-engineering of software systems, and the mining of software repositories. The goal of Rascal is to eliminate the overhead from analysis and transformation tool integration by offering an environment for the construction and experimentation of analyses and transformations that is usable by a large group of developers. The key features of Rascal include integrated syntax definitions, build-in data types, constructs for analysis and transformation, and an ecosystem of libraries. Integrated syntax definitions refer to the declaration of CFGs within the Rascal code and the application of concrete syntax trees and non-terminals as first-class values and types respectively. More specifically, syntactic features in Rascal are based on the SDF. Build-in data types (also) refer to the ability of applying values for pattern matching, such as the production of concrete syntax trees from Rascal-based grammar definitions, and represent the primary mechanism for case distinction in Rascal. Constructs for analysis and transformation refer to the use of lists, maps, sets, and relations for the expression of analysis tasks, such as aggregation and projection in Rascal's scripting language Rscript and the computation of the McCabe complexity [149] of methods in a Java project. The ecosystem of libraries refers to re-usability and language-specific functionality based on the employment of libraries.

EMFText Heidenreich et al. presented EMFText [101, 102]—an EMF-based language workbench for the development of textual languages from Ecore models and the generation of accompanying tool support[2]. The concrete syntax of EMFText-built languages are defined by a dedicated concrete syntax specification language that is based on the EBNF, relies on ANTLR for parser generation, and follows the concept of *convention over configuration*, i.e. the preference of compactness over expressiveness. EMFText offers support for generating textual languages from Ecore models that follow the design of Human-Usable Textual Notation or Java. The abstract syntax of EMFText-built languages are defined by Ecore models and, thus, rely on Essential Meta Object Facility (EMOF) (cf. Section 2.4.2). The static semantics of a language refers to the use of algorithms for static analyses, such as name analysis and static type resolution, and is implemented in EMFText by a default name resolution mechanism and (alternatively) by the use of the JastEMF tool [37] for the specification of static semantics of Ecore models through Reference Attribute Grammars (RAGs) [98]. The dynamic semantics of a language refer to the meaning of language artifacts and may be defined in EMFText by operational and translational semantics. Typically, the former refers to the implementation of GPL-based interpreters, while the latter to that of code generators. In summary, the process of developing a language in EMFText entails the following

[2]The EMFText guide is available online at https://github.com/DevBoost/EMFText/blob/master/Core/Doc/org.emftext.doc/pdf/EMFTextGuide.pdf.

steps: specification of an Ecore-based language metamodel; specification of concrete syntax through the use of EMFText's dedicated concrete syntax specification language; (optionally) specification of static semantics by utilizing reference resolvers or JastEMF; (optionally) specification of dynamic semantics, e.g. by implementation of an interpreter or post-processor; generating IDE tooling; and (optionally) customizing IDE tooling, e.g. by adaptation or implementation of syntax highlighting, content-assist, and quick fix solutions.

MontiCore Krahn et al. presented MontiCore [135]—a framework for composition-based DSL development. MontiCore employs a dedicated language for the specification of DSLs that may be categorized as meta-grammarware, as it represents an extension of CFGs with the capabilities known from modelware and OOP, in particular, inheritance and language embedding. The former allows the incremental development of changes to a language, while the latter the combination of language fragments into (new) coherent languages. Although the arguments against the integrated definition of both abstract and concrete syntaxes are discussed by Krahn et al. (i.e. "languages may have different concrete but only one abstract syntax" and "abstract syntax is often different from concrete syntax so that tasks of semantic analysis can focus on the structure of the language without being overrun by syntactical issues"), MontiCore employs such an integrated definition motivated by simplicity and catering for unskilled language developers who neglect consistency among separated definitions [135]. As a result, the bidirectional mapping between abstract and concrete syntaxes is not retained, in contrast to other approaches, such as the Textual Concrete Syntax (TCS) [157, 82]. A MontiCore-based DSL is defined by its MontiCore grammar and used to derive abstract syntax representations and, more specifically, an EBNF version of its MontiCore grammar that is subsequently supplied to the ANTLR tool to generate a predicated-LL(k) parser. The key differences between CFGs that employ the EBNF and MontiCore grammar include the ability of the latter to represent constructs, such as abstract classes, interfaces, associations, and (detailed) cardinality, i.e. based on exact numbers instead of (only) zero, one, or many. MontiCore realizes the ability of language embedding by allowing the specification of external non-terminals in its grammar. In more detail, external non-terminals, i.e. also referred to as bottom non-terminals in grammar fragments [141], are used on the right-hand side of production rules and represent MontiCore's mechanism to continue parsing according to the grammar of the embedded language. Grammar inheritance in MontiCore is realized in the same way as the LISA tool [148] and, more specifically, by enabling the specification of a set of grammars from which all production rules are *inherited* by a new grammar, as opposed to *integrating* an existing set of grammars into a new grammar. In addition, the MontiCore ecosystem provides a tool for the implementation of generative and analytic tools for DSLs, such as Eclipse plugins and DSL-specific IDEs, which focuses on modular decoupled development, the integration of different languages, multi-platform executability, flexible configuration, and API support for recurring generator-related tasks.

Spoofax Kats et al. presented Spoofax [118]—a platform for the design of textual DSLs in an integrated environment that offers declarative metalanguages for syntax definition, name binding, type analysis, program transformation, and code generation. For example, Syntax Definition Formalism (SDF)3 and NaBL represent Spoofax metalanguages for the specification of syntax and name bindings respectively. More specifically, the former is used to describe both concrete and abstract syntaxes in the form of templates that combine lexical and context-free syntax, while the latter is used to define declarative name bindings and scope rules. The implementation of Spoofax entails a mapping of declarative language designs, i.e. instances that conform to their metalanguages, to parameters of a set of language-parametric runtime systems. In a nutshell, Spoofax metalanguages are bootstrapped to derive Spoofax-based IDEs for language designs specified by the use of their metalanguages.

Xtext Eysholdt et al. presented Xtext [70]—a language workbench framework that enables the construction of modeling language implementations with sophisticated features similar to modern IDEs for GPLs. Xtext builds on the Eclipse Modeling Framework (EMF) (cf. Section 2.4.2) and implements modeling languages (cf. Section 2.4.2) typically by following the grammar-first or model-first approach (cf. Sections 2.3 and 2.4 respectively). The former entails the construction of Extended Backus–Naur Form (EBNF) grammar and derivation of an associated Ecore metamodel, i.e. an abstract representation of the language under construction. The latter entails the construction of an Ecore metamodel and the generation of (default) grammar, i.e. targeting a HUTN-like visual appearance. The selection of either approach is usually the result of an engineer's familiarity with it in a particular technical space [139, 218]. Thus, grammarware engineers, i.e. language developers that are more familiar with traditional CFGs as opposed to metamodels, frequently follow the grammar-first approach. In contrast, modelware engineers, i.e. language developers that are more familiar with MDE-based technologies than with language grammars, commonly follow the model-first approach. Although Xtext supports both, the main focus is to provide grammars at the front-end and metamodels at the back-end in order to facilitate tool interoperability [171, 223].

Next to either the employment of the grammar-first or model-first approach, a dedicated ANTLR-based parser [174] and skeleton IDE are generated and manually refined by supplying customized implementations of validation rules, content-assistance, and quick fix solutions. The reasoning behind choosing Xtext as a candidate language workbench framework for the implementation and evaluation of the contributions of this thesis include but are not limited to: rich feature-support such as mentioned above; integration and re-engineering of formal constraint language OCL [217]; the official nature and maturity of a project such as Eclipse that continuously participates in the Eclipse release train [83] and, thus, offers a stable release schedule; detailed documentation [22] offering valuable assets for the development of third-party implementations; and an active community that operates a forum[3] and offers support to developers.

[3]The Eclipse TMF (Xtext) Community Forum is available online at https://www.eclipse.org/

Cedalion Lorenz et al. presented Cedalion [146]—an approach that follows similar goals as approaches of the *language workbench* category in the Language Oriented Programming (LOP) paradigm, i.e. also including categories for approaches that build on *internal DSLs* and *external DSLs*. Their approach differs from other language workbenches, such as Jetbrains MPS [176], Spoofax [118], and Intentional Software [194], by providing external DSL tooling to internal DSLs, as opposed to providing external DSLs with the tooling of internal DSLs. In other words, Cedalion represents an IDE workbench that hosts internal DSLs. Their focus is on overcoming the limitations of internal DSLs in regards to the definition of syntax and semantics that are the inherent consequence of the host language confinement. Moreover, their motivation is that the implementation of internal DSLs is more cost-effective when compared to external DSL implementations. Their approach is implemented as an Eclipse-based IDE that features static validation and projectional editing, i.e. based on direct Abstract Syntax Tree (AST) manipulation, as opposed to the parsing of character sequences. As a result of projectional editing, concrete syntax definitions in Cedalion require the specification of projection definitions, i.e. statements that formalize the visualization of language concepts, that are analogous to grammar production rules except that the result is visualized through rendering instead of parsing. The evaluation of their approach is based on a case study in the domain of biology and, in particular, the design of DNA microarrays for molecular biology research.

SugarJ Erdweg et al. presented SugarJ [68]—an approach that offers library-based syntactic extensibility based on the Spoofax language workbench [118]. This approach is implemented through the Eclipse-plugin Sugarclipse and offers support for the host languages Java, Haskell, and Prolog. A more detailed summary of their approach is presented in Section 3.2.

Jetbrains Meta Programming System Pech et al. presented the Java-based tool Jetbrains Meta Programming System (MPS) [176] that is based on the concept of projectional editing (also referred to as *structured editing* [213] and *syntax-directed editing* [120]), thus enabling the use of syntactic forms that are non-parseable and non-textual, such as mathematical symbols and tables. In general, syntax-directed IDEs constantly operate on an AST and instantiate AST elements as templates with holes, i.e. act as placeholders, that are continuously filled as the user provides further input. Consequently, programs in such IDEs are syntactically valid and unambiguous at any point in time. The goal of MPS is to increase developer productivity through the composition and modularization of languages. The implementation of a language through the use of MPS entails the definition of abstract syntax, concrete syntax, IDE functionality, type system, and code generator rules that are briefly introduced below. The abstract syntax of MPS languages is defined by their structure, constraints, and behavior. Language structure includes concepts, properties, and relationships. Language constraints define restrictions on the values of language properties and references. Language behavior relates implementation methods with language concepts. The concrete syntax of MPS languages

forums/index.php?t=thread&frm_id=27.

is defined by their projectional IDE and, in particular, by the assignment of visual notations to language concepts. As a result, grammar and parsing is neglected and the use of non-parsable notations enabled. In regards to IDE functionality, MPS languages may be customized by the use of scoping rules and dataflow aspects that determine the results displayed in the code-completion menu and highlight unreachable code errors respectively. In terms of the type system in MPS, an engine that evaluates types on correctness is provided for build-in types. The support for dedicated type rules and associated error- or warning-reporting in MPS is discussed in Section **??**. Further, MPS languages may define transformation rules that employ template-based model-to-model transformations and model-to-text generators to create target language program code in the form of a model, and textual program code that is compatible with the target language compiler respectively.

Eco Diekmann et al. presented Eco [60]—a tool implementation of an approach to edit composed programs that is based on an incremental parser extended with *language boxes*. The motivation of Eco is to enable the composition of arbitrary syntaxes, while maintaining the operation on a valid syntax tree, (as in syntax-directed IDEs such as JetBrains MPS), in order to implement IDE features such as name binding analysis. In a nutshell, the goal of Eco is to solve the issues of syntax-directed IDEs. Each language box in Eco has its own type, e.g. SQL, incremental parser (maintaining its own parse tree as its value), and editor. Language boxes allow the embedding of languages inside one another and expose consistent interfaces to a global tree (referred to as "Concrete Syntax Tree" in [60]) that integrates the internal trees of individual language boxes and allows to render programs on the screen. Incremental parsing constitutes an online process and, in particular, involves the continuous parsing and updating of user input and parse trees. Eco implements an incremental parsing algorithm presented by Wagner [214] and extends it to create Abstract Syntax Trees (ASTs). More precisely, every non-terminal in the parse tree is extended by a new attribute that references a corresponding AST node, while the AST directly references tokens in the parse tree. The advantage of sharing tokens between the parse tree and the AST is that token values are kept updated. Eco saves files in a custom tree format, as opposed to the source code typically applied by parser-based IDEs. The implementation of incremental ASTs in Eco relies on the fact that AST nodes are only created from the parse tree, with no references created from AST child to parent nodes. More specifically, Eco creates ASTs from parse trees by employing a rewriting language, similar to TXL [50] and Stratego [32], that provides the option to define a single rewrite rule alongside each production rule. Eco employs ASTs for the implementation of IDE features, such as scoping rules and code completion, by implementing a subset of the Name Binding Language approach by Konat et al. [129].

Racket Felleisen et al. presented Racket [77, 78]—a language-oriented approach to problem solving. The Racket language is inspired by Lisp and Scheme and represents an untyped functional language. The implementation of a Racket-based language entails the development of a compiler that maps client programs to a target language, an interpreter

that continuously traverses client programs, Redex reduction semantics [76], a linguistic derivative of an existing Racket language, or a combination thereof. In general, Racket programs are structured into modules that may each employ a different language. Further, the mapping that a Racket module specifies to a particular language is used to publish the implementation of a language, as well as its customization, such as syntax coloring and static analysis. Hence, the development of a Racket-based language may entail the addition, subtraction, and re-interpretation of runtime facilities and constructs from a base language such as core Racket. Racket uses a descendant of Scheme and Lisp's hygienic macro system [45] for the representation of syntactic terms by virtue of syntax objects, i.e. containing both the properties of the source syntax and those defined by a language developer, as well as the specification of rewriting rules for the translation of language modules into the Racket core syntax. More specifically, Racket syntax objects are data structures that include syntax properties and symbolic representations of program fragments. DrRacket, i.e. the IDE of Racket, employs a tool that exploits information supplied by Racket macros, such as syntax bindings and syntax properties, to customize user experience during source code manipulation.

Melange Degueule et al. presented the *Melange* framework [59] that may be employed for the assembly and customization of DSLs from legacy artifacts through the use of specific constructs for the representation of operational semantics and abstract syntax. The Melange framework is bundled as a set of Eclipse plug-ins and composed of a meta-language and supporting tooling for the implementation of DSLs as first-class citizens that may be restricted, reused, extended, or adapted into other DSLs. Further, DSLs that are built with Melange employ Xtext or Sirius for the implementation of textual or graphical IDEs respectively. Melange builds on the EMF Ecore language and the Xtend programming language [22] for the expression of abstract syntax and operational semantics respectively. More specifically, operational semantics are defined by the use of aspects that have been introduced as annotations in a Melange-customized extension of the Xtend programming language. The Melange language specifies the concept of a language to be defined by a metamodel and semantics. The composition of the former is a set of classes and of the latter a set of aspects. Consequently, aspects are employed to weave behavior into the classes of a metamodel that define a language.

Ensō Loh et al. presented Ensō [145]—a two-level approach to data abstraction that entails the definition of data description and manipulation mechanisms, and the use of such mechanisms for the specification of various kinds of data. In other words, Ensō offers an implementation of the concept of *Managed Data* that is based on data model bootstrapping, i.e. the use of data models to describe themselves, and the interpretation of data models by enabling programmers to define the behavior of data manipulation operations. The motivation of Ensō is to tackle the limitations of predefined data structuring mechanisms that may be insufficient for the implementation of common data management requirements, such as caching, persistence, transactions, and access control. The implementation of Ensō is based on the reflective capabilities of the Ruby

programming language. The key components of Ensō include schemas, data managers, and integration. Schemas define the structure and properties of data in a self-describing manner, i.e. similar to the MOF meta-metamodel and the BNF grammar, and are loaded into memory by a boostrapping mechanism. Data managers are responsible for the creation and manipulation of schema-conforming data instances and may be composed by a stack of managers to combine behavior. Finally, integration refers to the management of data instances as practiced by a programming language, such as *objects* in an OOP-based programming language.

Summary In conclusion, this section provided an overview of existing frameworks and tools for the engineering of DSMLs and, in particular, approaches that are based on bindings and transformations (MetaEdit+ [195, 206]), generative programming (Intentional Software [194]), integrated syntax definitions and pattern matching (Rascal [123]), concrete syntax convention over configuration (EMFText [101, 102]), meta-grammarware (MontiCore [135]), metalanguages for the specification of syntax and name bindings (Spoofax [118]), grammar-first and model-first development (Xtext [70]), projection (Cedalion [146] and Jetbrains MPS [176]), language boxes and production rule rewriting (Eco [60]), compiler development (Racket [80, 78]), annotations and transformations (Melange [59]), and data manipulation operations (Ensō [145]). Section 5.7 analyzes DSML engineering frameworks in comparison with the proposed approach on the automated generation of consistency-achieving DSML implementations.

3.3 Design of textual notations

This section presents literature on visual language representations by focusing on work on the specification of textual notations.

Classification of Concrete Textual Syntax Mapping Approaches Goldschmidt et al. [89] presented a classification of existing concrete textual syntax mapping approaches, as well as the identification of a set of issues associated with incremental parsing, model updating, and partial and federated views. This classification distinguishes between three different cases: (*i*) manual development or auto-generation of a metamodel from an existing formal language specification, such as a language grammar, (*ii*) manual development of grammar based on an existing metamodel, and (*iii*) manual development of a mapping between an existing metamodel and grammar. Moreover, existing work can be categorized [89] into solutions that (*i*) bridge grammar-first and model-first, such as [218] and [116], (*ii*) offer grammar-based code transformation for MDA, i.e. the direct attachment of visual representation specifications to existing models, such as in the Textual Concrete Syntax (TCS) approach [115], MontiCore [135], the Human-Usable Textual Notation (HUTN) [159], the Textual Editing Framework (TEF) [190], the JetBrains Meta Programming System (MPS) [112], and Gymnast [88], and (*iii*) differ from previous categories, such as the Eclipse IDE Meta-tooling Platform (IMP) [42], the

Eclipse Textual Modeling Framework (TMF)[4,5], and Intentional Programming [194]. A more detailed summary of some of these works is presented alongside other language workbenches in Section 3.2.

Human-Usable Textual Syntax Notation Muller et al. [159] presented an experience report on the use of the Human-Usable Textual Notation (HUTN) as a bridge between modelware and grammarware. The goal of their work is to enable the generation of parsers and IDEs for textual modeling languages defined by metamodels. Their motivation is to facilitate low turn-around time for DSL prototyping. Their approach entails the generation of grammar from MOF-based metamodels that follows the OMG HUTN Specification [93] and can be supplied to grammarware tools that automate the generation of parsers and IDE implementations for DSLs. The main advantages of HUTN-based languages include: universal applicability to MOF-based models; suitability for the automated generation of parsers; and design-conformance with human-usability criteria. Muller et al., however, reported that there are several ambiguities in the HUTN Specification. For example, syntax tree ambiguities may be introduced by: keyword attributes; class contents such as references; and class instance references. Solutions to eliminate these ambiguities are proposed and implemented in a HUTN parser generator integrated into the Kermeta workbench. Similarly to Muller et al., Rose et al. [185] presented an implementation of HUTN that provides a generic concrete syntax for MOF-based metamodels. They encountered similar problems and employed the same solutions as Muller et al. In addition, they encountered a problem related to syntactic shortcuts for HUTN adjectives and, in particular, that terminal rules *AttributeName* and *ClassName* (cf. HUTN Specification [93]) produce the same token type, which causes ambiguity. Their solution to the latter is realized by the adaptation of the HUTN grammar to require HUTN adjectives to be prefixed with the symbol of a colon. In summary, they identified means in which HUTN can be applied to improve the productivity of MDE. For example, HUTN may be used to increase the quality of test suites for verifying model management operations such as model-to-model transformations.

Textual Concrete Syntax Jouault et al. presented Textual Concrete Syntax (TCS) [115]—an extension to the ATLAS Model Management Architecture (AMMA) [23] framework with a dedicated DSL for the definition of concrete textual representations that is also referred to as Textual Concrete Syntax Specification Language (TCSSL) [158]. The objective of their approach is to enable bi-directional translation of grammarware models, referred to as "text-based DSL sentences" in their work, and their equivalent model representation that is motivated by the reduction of redundancy between the specification of metamodels and grammars, such as that introduced by the duplicated definition of element-multiplicity, and the re-use of existing grammarware tooling. Hence, the goal is to provide a bridge between modelware and grammarware by linking metamodel elements

[4] Xtext [65] is the only component mentioned on the Eclipse TMF website.
[5] The Eclipse TMF project is available online at `https://www.eclipse.org/modeling/tmf/`.

with the textual notation of a DSL and, in particular, by employing a sequence of transformations on TCS models and metamodels. Their approach defines textual concrete syntaxes through associations of syntactic elements, e.g. language-specific keywords, with metamodel elements by the application of model annotations. The definition of TCS-based DSLs entails employing TCS models alongside metamodels for the generation of annotated grammar and IDE that is built on TCS services, i.e. released as part of the AMMA framework. More precisely, the AMMA framework provides several DSLs for the definition of DSL components and, in particular, KM3 for domain concepts, ATL for transformations, and TCS as a bridge between modelware and grammarware based on KM3 and ANTLR version 2, i.e. a parser generator for LL(k) grammars. The bridge between modelware and grammarware that is proposed by TCS employs a pair of translators that consist of an injector and an extractor, and is generated by grammar and a metamodel respectively. The former employs a grammarware model for the production of a corresponding modelware model. The latter uses a modelware model for the production of a corresponding grammarware model. In both cases, grammarware and modelware models conform to an ANTLR-based grammar and a KM3-based metamodel that captures the structural semantics of a particular domain respectively.

The basic constructs of TCS include primitive templates and class templates. The former define the lexer token that corresponds to a data type in the metamodel. The latter define the representation of a class in the metamodel and consist of a sequence of syntactic elements. While the former may be defined multiple times for the same data type, the latter may only be defined once for the same class. The syntactic elements that are used within class templates include keywords, special symbols, and properties. Keywords are reserved words with specific meaning and are defined between double quotes. Special symbols correspond to separators or operators such as curly braces. Properties refer to structural features that are defined by attributes and references in a metamodel class or one of its super classes.

Additional constructs of TCS include abstract class templates, conditionals, operators, and a symbol table. Abstract class templates permit the navigation of the inheritance hierarchy and are realized as an alternative in the non-terminal rule that corresponds to the subclass(es) of their associated class. Conditionals impose requirements to the presence of syntactic element sequences. TCS employs a symbol table to manage cross-references. Operators are defined with a priority, may be referred to by operator templates, and are employed by the generated LALR(1) parser during shift-reduce conflicts.

Furthermore, in regards to coding style and indentation, TCS models may define blocks, special symbol spacing, and custom separators. Blocks are delimited by square brackets and contribute indentation information. Special symbol spacing definitions specify the prefix and suffix of symbols and may, thus, be employed to enable whitespace support. Example use cases of custom separators include the forced serialization of spaces and line feeds.

The implementation of TCS employs text-to-model traceability by keeping track of lines and columns to visualize links and tool-tips in models. Moreover, a generic textual

IDE is provided that may be parameterized by information from TCS models to provide a tree representation of models.

Textual Concrete Syntax Analysis and Synthesis Muller et al. [156, 157] presented an experiment in the field of MDE that proposes bidirectional mappings between abstract and concrete syntax specifications, both of which conform to the same meta-language, such as MOF or Ecore, and thus do not intend to bridge technical spaces but rather remain within the technical space of modelware. Their work highlights the complexity of such mappings, as well as the perception that grammar may be considered as metamodel but not vice-versa, as investigated in an earlier work [122]. The metamodel of their concrete syntax language is composed of: an abstract class Rule, i.e. representing the root element, with sub-classes Class and Feature that reference a class and a property in their abstract syntax language respectively; a class Template that is used to specify the relationship between a class in the abstract syntax model and one or many Rule classes; and the classes Iteration and Value that are employed to establish the relationship between the properties and values of a class in the abstract syntax model. In their work, the generation of operational tooling is achieved by the use of templates, and includes parsers and text generators.

Textual Editing Framework Scheidgen et al. presented the Textual Editing Framework (TEF) [190]—an approach to embed generated EMF-based textual model IDEs into graphical and tree-based IDEs that are created with GMF. The goal of TEF[6] is to enhance the effectiveness of graphical modeling IDEs, which act as host IDEs that partly rely on textual representations, such as mathematical expressions, by enabling language engineers to provide styles (referred to as "notational descriptions" in Scheidgens work) for metamodel elements based on "TSL", i.e. their dedicated notation description language composed of a combination of Context-Free Grammar (CFG) and BNF elements that are augmented with uni-directional mappings to a domain metamodel. They report that the initial and final approaches for updating the model employed the MVC pattern and background parsing respectively. The method of background parsing allows textual models to be used as primary artifacts, i.e. as opposed to EMF models. Language engineers that employ TEF may provide a complete or partial textual notation mapping for a metamodel. In the case of the latter, engineers have to make sure that all elements of a metamodel that are intended to be instantiated in the embedded textual IDE are defined in the (partial) notation specification. Further, TEF introduces the component of a layout manager that allows the creation of whitespaces for a set of roles and, more specifically, those that represent a space, an empty string, a statement, the start of a block indentation, and the end of a block indentation. The authors of this work conclude that in cases where both textual and graphical notations are to be defined, such as for graphical IDEs that embed a textual editor, it may be more natural to integrate both

[6]The TEF documentation is available online at https://www2.informatik.hu-berlin.de/sam/meta-tools/tef/documentation.html.

types of notation descriptions in a single notation description to increase conciseness and coherence.

SugarJ Erdweg et al. [68] presented SugarJ—a Java library-based approach for the development of language extension modules, such as IDE support, domain syntax, and static analyses, for embedding DSLs into the Java host language. The goal of their approach is to provide the ability to express domain concepts using domain syntax that is neglected by many approaches for embedding DSLs into a host language, i.e. primarily focusing on the integration of domain concepts at the semantic level. SugarJ employs the concept of syntactic sugar for the definition of object language grammar extensions that are activated and composed by the use of library import statements. SugarJ employs the Syntax Definition Formalism (SDF) [99] and transformation language Stratego [64] for defining the syntax of grammars and desugaring implementations respectively. Desugaring transformations are responsible for the transformation of extended syntax to base syntax, i.e. the syntax of the host language, and entail incremental parsing and grammar adaptation that results in an abstract syntax tree. SugarJ encapsulates syntactic language extensions as libraries that define both syntax extensions, as well as desugaring transformations. Further, these libraries are imported and declared at the topmost level of files (only) to allow the SugarJ compiler to interleave parsing and desugaring at the granularity of top-level entries.

EMFText Heidenreich et al. presented EMFText [101, 102]—an approach for defining visual textual representations for Ecore-based metamodels, as well as generating textual IDEs from such definitions. A more detailed summary of EMFText is presented alongside other language workbenches in Section 3.2.

Sirius Viyović et al. presented Sirius [211]—an Eclipse project that enables the development of graphical modeling languages by following the model-first approach that is initiated by the construction of a metamodel, referred to as "domain model" in Sirius. Although the contributions of this thesis focus on textual modeling languages, Sirius represents a popular language workbench that is, similarly to Xtext [70], based on the EMF, and employed for the creation of *graphical* DSMLs. Moreover, Sirius conceptually separates abstract from concrete representations by the use of a set of mapping models, referred to as "viewpoint specification models" in Sirius, that define the appearance, structure, and behavior of a Sirius IDE in harmony with the abstract domain-specific concepts captured by its associated metamodel [184]. In a nutshell, these mapping models capture the visual graphical representation of metamodel elements in a Sirius-based graphical modeling language IDE. More precisely, different types of mapping models include: representation descriptions that define the type of representation, such as diagrams, tables, and trees; representation extensions defining additional functionality for customizing representations; validation rules that may be used to define validation procedures and quick fix repair suggestions; and Java extensions that specify supplementary IDE functionality, such as commands and gestures.

Concrete and Abstract Syntax Transformation Bridge Herrera et al. presented a domain-specific transformation language to bridge concrete and abstract syntaxes that facilitates the construction of (complex) bridges between grammars and metamodels by language engineers of a dedicated DSL [103, 36]. In general, their work differs to the proposed approach by facilitating the construction of bridging specifications between dedicated metamodels and grammars as first-class citizens instead of metamodel-agnostic notational specifications. Thus, the effective construction of valid bridges between domain metamodels and grammars requires language engineers to be fully aware of actions and tool specifications available on both sides of the bridge.

Multi-paradigm modeling Tendeloo et al. [200] presented an approach based on multi-paradigm modeling, and explicit modeling of mappings between abstract and concrete syntaxes. The goal of their approach is to enable the definition of multiple different notations for the same abstract concept, referred to as "front-end" and "back-end" in their work respectively. Their approach explicitly distinguishes between back-end and front-end by the responsibility for the perceptualizing, comprehending, and (meta-)modeling of concepts by the former, and the rendering of models on a specific platform, such as Scalable Vector Graphics (SVG), by the latter. Consequently, the back-end handles the transfer of models, e.g. as JavaScript Object Notation (JSON) files transferred over network sockets, and the front-end the mapping of concepts to a specific platform. Although the interface of the front-end is described independently from a specific platform, both front-end and back-end are required in order to implement the same platform-independent render's interface.

Layout-Sensitive Languages Amorim et al. [57] presented a perspective on the tooling for parsing and pretty-printing of layout-sensitive languages by the declarative specification of indentation rules. They highlight that state-of-the-art language workbenches offer little support for layout-sensitive languages and, thus, restrain their development. Their approach illustrates the specification of indentation rules and their use for the automated derivation of layout-sensitive parsers and pretty-printers. The authors highlight that the annotation of CFG production rules with layout constraints is a rather verbose and low-level procedure implied by the comparison of lines and columns of tokens of different sub-trees. As a result, their approach provides the ability to specify (common) indentation declarations that involve Landin's offside rule [142], i.e. indicating tokens that occur further to the left than the first line as invalid structures, the alignment and indentation of constructs, and tree selectors. Their approach creates abstract representations of program structure and layout through the use of the Box language [208]. In comparison, their distinction between indentations and newline-indentations, i.e. enforcing the start of a target sub-tree to occur further to the right than another sub-tree, renders the specification of the latter less cumbersome than an equivalent implementation that employs the Xtext grammar language, i.e. involving the definition of offside rules through the use of synthetic tokens.

Summary In conclusion, this section introduced existing work on the design of visual presentations and specifically those that target textual representations, namely HUTN (i.e. a single notation for all purposes), background on tool support for layout-sensitive languages, and approaches that are based on the explicit modeling of mappings between abstract syntax and concrete syntax (i.e. realized as transformations at model- or code-level). Section 6.7 analyzes works in comparison with the proposed approach on the design and application of reusable textual styles for modeling languages.

3.4 Model composition and management

Erdweg et al. [67] presented work that is based on the idea of language-oriented software development [152], and that language composition is required for realizing language-oriented software projects. Their goal is to provide a separate DSL for each domain occurring in a project and employ these DSLs together. They present five forms of language composition: language extension, i.e. the combination of a base language with a language extension, which inherently depends on and reuses the base language; language restriction, i.e. the extension of the base language validation phase offering the capability of rejecting non-conforming language instances; language unification, i.e. the combination of independent languages, which is rarely practiced due to challenges imposed by deep and bidirectional integration requirements, such that the implementation of both languages can be reused unchanged solely by supplementing glue code; language self-extension, i.e. the extension of a host language by reusing and retaining its own implementation, e.g. by reusing the host language compiler; and language extension composition, i.e. the combination of multiple different language extensions for offering combined language features.

This section briefly introduces existing work in the field of model management and, in particular, the ModelGen operator [12], Higher-Order Transformations (HOTs) [204], and Epsilon [172].

ModelGen Atzeni et al. presented ModelGen [12]—an implementation of the ModelGen operator initially illustrated by Bernstein [20]. Conceptually, the ModelGen operator is based on the management of models from a high-level perspective by the application of generic model operations, such as matching, merging, and composing, in sequences to enable complex integration scenarios. In a nutshell, the ModelGen operator defines a general pattern which uses bridges on the meta-language level to derive transformations on the language- and instance levels. The implementation provided by Atzeni et al. includes the use of ER and UML diagrams and the relational data model. Traditionally, this pattern is proposed and used in the database field for schema-independent transformations, but it is also applicable in language engineering.

Higher-Order Transformations Tisi et al. [204] highlighted the need for the direct manipulation of model transformations to address more complex applications of modeling

and transformation paradigms. They argue that the same flexibility and uniformity of the model-driven paradigm can be applied to the transformation infrastructure by introducing the concept of transformation models. Such transformation models include the input of HOTs, a different class of model transformations.

Epsilon Kolovos et al. presented Epsilon [172]—an integrated and uniform platform for model management in MDE that typically involves operations on models that are expressed by a variety of languages and technologies. Epsilon provides a set of interoperable languages for task-specific operations, such as comparison, validation, transformation, merging, and refactoring, that are built on an implementation sharing operational semantics, and abstract and concrete syntaxes. Primarily, the architecture of Epsilon consists of five layers: the model connectivity layer, enabling the management of heterogeneous models in a uniform manner; a core language that offers a set of generic features that may be reused by task-specific languages; a set of languages that target distinct model management tasks; and a workflow layer that enables the integration of individual task-specific language tasks and, in particular, their composition, execution, debugging, and monitoring. More specifically, the model connectivity layer provides a uniform interface for the access of heterogeneous models through the use of the Epsilon Object Language [124], i.e. the core language in the Epsilon platform. The set of task-specific languages is composed of a language for model validation (Epsilon Validation Language), model comparison (Epsilon Comparison Language), model merging (Epsilon Merging Language), model-to-model transformation (Epsilon Transformation Language), and in-place model transformation (Epsilon Wizard Language).

3.5 Summary

In general, users of modelware and grammarware approaches rely on language engineers handcrafting transformation rules between either individual grammar rules or terminal rules and metamodel elements. Neglecting the automated transformation of XML Schema definitions to grammar specifications makes it impracticable for language engineers to rapidly prototype textual modeling language implementations for existing XML Schema definitions. The generation of language grammar from metamodels is still challenged by the requirement of manual customization and extension by language engineers [162]. For example, the report on the migration of a modeling environment from XML/UML to Xtext/GMF presented by Eysholdt and Rupprecht [71] requires language engineers to manually perform customizations to grammars generated from imported XML Schema definitions. Izquierdo et al. [111], Kunert et al. [138], and Butting et al. [39] present approaches based on ANTLR and EBNF-based grammar evolution and thus do not consider language specifications in form of XML Schema definitions or metamodels.

The inherent differences between grammars and metamodels render the construction and maintenance of complex structural constraints infeasible in the former type of language specification. Model-Driven Language Engineering (MDLE) frameworks such as

Xtext [70], DrRacket [80], and MetaEdit+ [195, 206] address the infeasibility of language grammars to capture complex structural language restrictions by enabling engineers to specify them as part of language metamodels such as OCL invariants in Ecore metamodels. The facilitation of language specifications with complex structural restrictions for automating the generation of language IDEs with built-in model validation, content assist and model repair, however, is limited or nonexistent. DrRacket and MetaEdit+ allow language specification to include complex structural restrictions, however, require engineers to provide implementations for visualizing error locations and recovery of contract validity. Xtext, JetBrains MPS, Melange, Spoofax, and SugarJ provide partial support for structural constraints. Melange and Xtext rely on the OCL interpreter provided by EMF for the validation of models of languages defined by Ecore metamodels that are augmented with OCL invariants. Therefore, errors in models are reported coarsely (e.g. by visualizing entire sub-structures as erroneous) and content-assist proposals may introduce constraint violations (e.g. by suggesting the use of illegal tokens).

Predominant flows of operation in the specification of visual textual representations presented in literature include the following three categories.

First, manual development or automated generation of domain-specific metamodels from grammar specifications such as demonstrated by Spoofax [118], SugarJ [68], and MontiCore [39, 135]. Spoofax and SugarJ require language engineers to construct and maintain (complex) bi-directional transformations. Spoofax employs host language grammar-dependent transformations to weave and unweave library-based notation-specifications in and out of Java, Haskell, and Prolog. Similarly, SugarJ specifies notations by construction and maintenance of bi-directional transformations through Java host language extension modules. MontiCore approaches the challenge of reducing redundancy and improving maintenance by employing EBNF-like grammars (i.e. acting as artifacts to capture both structure and representation) to generate metamodels and implementations. Notation that is specified for each structural concept of a domain, however, introduces redundancy among concepts following the same type of representation. The reduction of redundancy achieved by notation-specifications for types of structural concepts are not considered in MontiCore.

Second, manual development of grammar specifications based on domain-specific metamodels such as demonstrated by TCS [115], TCSSL [158], and TEF [190]. TCS employs a sequence of transformations on metamodels alongside models of a dedicated DSL to specify associations to elements in individual metamodels. TCSSL is based on the specification of bidirectional mappings between AST and CST by means of EBNF-like rules. TEF requires the implementation of complete concrete syntax specifications for metamodel elements that are intended to be instantiated.

Third, manual development of mappings between domain-specific metamodels and grammar specifications such as demonstrated by EMFText [101, 102] and Herrera et al [103, 36]. Although EMFText offers the capability to customize the appearance of languages beyond textual tokens, such as the definition of text color, it does so on the basis of individual metamodels and relying on a dedicated generator implementation to

pick up such specifications during editor generation. Herrera et al. present an approach that builds on the construction and maintenance of (complex) bridges-specifications between grammars and metamodels. Bridge-effectiveness, therefore, is challenged by the level of awareness and application of actions and tool specifications that are available on both sides of a bridge.

A more detailed analysis between related work outlined in this chapter and the contributions of this thesis is presented in Sections 4.8, 5.7, and 6.7.

CHAPTER 4

XML Schema modeling integration and assistance

A multitude of Domain-Specific Languages (DSLs) have been implemented by means of XML Schema definitions. While such DSLs are well adopted and flexible, they lack modern DSL IDE functionality. Moreover, since XML is primarily designed as a machine-processible format, XML-based artifacts lack comprehensibility and, therefore, maintainability. In order to tackle these shortcomings, a bridge between XML Schema-based languages and textual modeling languages is proposed [162]. This bridge exploits existing seams between the technical spaces of XMLware, modelware, and grammarware, and closes identified gaps. The resulting approach is able to generate Xtext-based IDEs from XML Schema definitions that provide powerful IDE functionality, basic customization options for the appearance of textual concrete syntax, and round-trip transformations which enable the exchange of data between the technical spaces in question. Further, the approach is integrated into the implementation of a modeling and metamodeling assistant to enable developers to import and query domain-specific knowledge embodied by XMLware artifacts, as well as facilitate the construction of novel modeling languages. The approach is evaluated by means of a case study on TOSCA, which is an XML-based standard for defining Cloud deployments, as well as a use case involving the development of an industry standard-conforming modeling language by facilitating the integration into a modeling and metamodeling assistant. The results show that the approach enables bridging XMLware with modelware and grammarware in several ways, which go beyond existing approaches, and allows the automated generation of IDEs that are at least equivalent to IDEs manually built for XML-based languages. Figure 4.1 recaptures the contributions of this thesis and highlights the contribution of this chapter, which is the exploitation of XML Schema-based language specifications for the automated generation of the structural components of textual modeling language implementations.

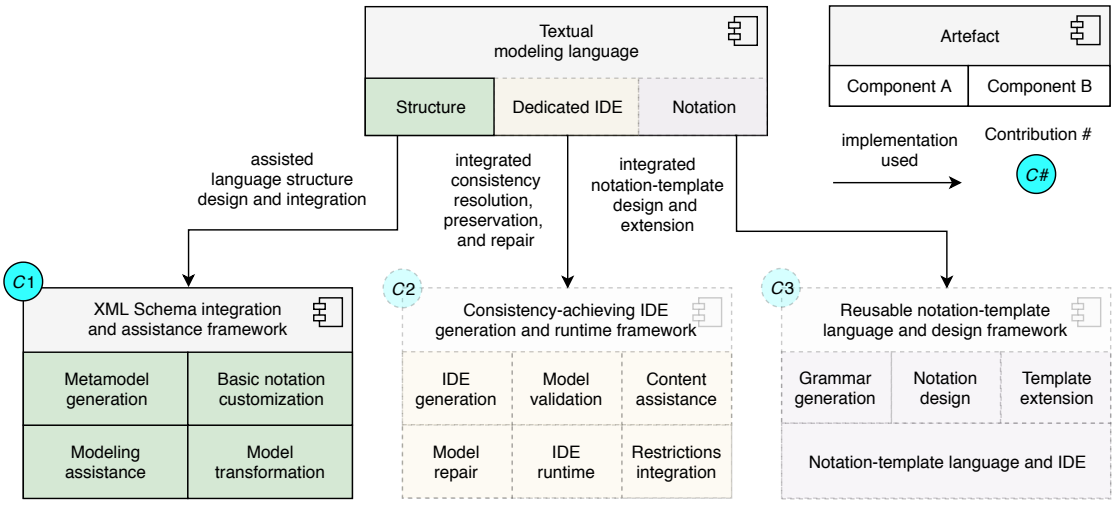

Figure 4.1: Contribution presented in this chapter.

The remainder of this chapter is organized as follows. Section 4.1 introduces the approach and its motivations. Section 4.2 briefly recaptures XML, XML Schema, and language workbench frameworks. Section 4.3 provides an overview of the challenges imposed by the existing chain of transformation from XMLware to grammarware. Section 4.4 details the gaps between XMLware, modelware, and grammarware by reclaiming the running example language introduced in Chapter 2. Section 4.5 presents the approach, its implementation and the modeling assistant integration in detail. Section 4.6 demonstrates an evaluation based on the migration of a standardized cloud modeling language. Section 4.7 presents the evaluation of the XMLText assistant integration based on the construction of a modeling language implementation for the specification of industry standard-conforming conveyor-belt systems. Section 4.8 analyzes and compares the proposed approach with related work. Finally, Section 4.9 concludes the chapter by summarizing the presented work.

4.1 Introduction

XML has been primarily designed as a machine-processible format following fixed angle-bracket syntax. More specifically, XML-based languages are bound to angle-bracket syntax that is described as verbose and complex in terms of human comprehension and are, therefore, difficult to maintain [13]. While for prominent XML-based languages, such as OASIS's TOSCA [173], advanced IDE implementations are handcrafted, for others, such as the Artificial Intelligence Markup Language (AIML) [216], no dedicated IDE is available. In the latter case, language users are bound to the application of angle-bracket syntax.

Tackling these major limitations requires breaking out of the inflexible XML syntax by providing support for the construction of modeling languages (henceforth simply

referred to as *modeling languages*). While state-of-the-art Model-Driven Language Engineering (MDLE) [113] frameworks, such as Xtext [70], enable the development of modeling languages, as well as the accompanying customized concrete syntax and rich language workbenches, handcrafting DSML implementations from XML-based languages with such frameworks is a complex, error-prone, and time-consuming task that requires advanced language-engineering skills. Additionally, Xtext-based modeling language implementations that ought to replace XML-based languages often neglect backward compatibility with the comprehensive set of applications that have been built to support these languages. Moreover, state-of-the-art MDLE frameworks allow the tailoring of modeling languages to target domain expert-familiar syntax that is rarely implemented by general-purpose notations [191].

Such modeling language implementations, however, have to be either constructed from scratch or derived from existing languages. Although the latter seems promising, it opens up a wide range of obstacles in the form of conformity and compatibility issues between the implementation of an existing language and its derived counterpart. For example, existing approaches derive metamodels from source languages for the purpose of generating grammars. Grammar is disconnected from its original metamodel and, as such, loses metamodel-conformity and the applicability of round-trip transformations within the technical spaces of XMLware and grammarware.

In order to overcome these issues, the proposed approach facilitates the modernization of XML Schema-based languages with modelware and grammarware [122] through the extraction of metamodels from existing XML Schema definitions, the adaptation of extracted metamodels to facilitate the production of effective language grammars, the generation of both customized language grammars and workbenches from adapted metamodels, and the enabling of round-trip transformations between original XML Schema-based languages and their modernized modeling language counterparts by automatically generating model serializers and parsers to ensure backward-compatibility.

The proposed framework combines the advantages of XMLware and grammarware, i.e. machine-processibility and re-use of extensive XMLware applications, and customized tool support and improved maintainability respectively, by introducing support for round-trip transformation. The XML to Xtext (XMLText) framework is evaluated based on a cloud modeling language use case in which the XML-based TOSCA language specification is employed to generate an executable modeling language implementation. Moreover, XMLText is integrated into a metamodeling and modeling assistant, and employed for the construction of a conveyor-belt system modeling language that conforms to an XML-based industry standard specification for production systems. Therefore, evaluating the framework's ability to produce modeling language implementations that conform to XML-based standard specifications, as well as feature modern editing capabilities and enable round-trip compatibility for language instances will facilitate the use of existing XML-based tools and engines. The evaluation criteria include completeness of the generated language elements and validity of XSD-conform XML instances that are round-trip-transformed to grammar-conform modeling language models and vice versa.

4.2 Background

Prior to the introduction of the Extensible Markup Language (XML), most data formats were binary and, therefore, solely machine-readable and machine parsable. Focus was primarily on processibility by machines rather than human developers and domain experts. Furthermore, since such data formats were also proprietary, they could only be read either by a single application or by a very small number of applications. As such, they are not suitable for the distributed inter-communicating systems that are in place today.

With the introduction of the fully machine-processible XML by the World Wide Web Consortium (W3C) [33] in 1998, a tremendous leap towards easing the design of software languages was achieved, which leverages the idea of having a generic editor, parser, and validation methodology. In 2004, W3C recommended the XML Schema Definition (XSD) [202] as a novel standard to specify and verify formally elements in an XML document that conforms to an XML Schema definition. Although the original design goals specify that "XML documents should be human-legible and reasonably clear" [33], the syntax of XML cannot be altered to fit the representation of a particular domain.

Many DSLs have been defined by means of XML Schema definitions. For instance, the XML Schema definition of RSS specifies a web feed format to publish frequently updated information. DSLs are software languages that are meant to target specific problem domains, rather than solving all kinds of problems like General Purpose Languages (GPLs). Fowler [84] defines a DSL as "a computer programming language of limited expressiveness focused on a particular problem domain". While XML-based DSLs are bound to the non-customizable textual concrete syntax of XML, MDLE frameworks, such as Xtext, allow the development of DSMLs, i.e. modeling languages defined by means of metamodels [205] that include fully customizable textual concrete syntaxes. Therefore, the textual concrete syntax of a modeling language can be tailored to a notation that is familiar to domain experts. For example, in order to target comprehensibility, the syntax of a modeling language may be based on notations such as OMG Human-Usable Textual Notation (HUTN) [185] or YAML Ain't Markup Language (YAML) [17], i.e. generic serialization formats that focus on human comprehensibility.

Contemporary MDLE frameworks are capable of automatically generating powerful DSL workbenches from abstract language syntaxes that include parsers, serializers, lexers, pretty printers, and sophisticated IDEs. Xtext is a prominent Eclipse-based MDLE framework that enables the development of textual modeling languages. Further, it is tightly integrated with the Eclipse Modeling Framework (EMF) and covers all aspects of a textual modeling language infrastructure, including the generation of basic implementations of lexer, parser, and an Eclipse IDE featuring syntax highlighting, background parsing, error indication, content assist, hyperlinking, quickfixing, and outline views.

Therefore, Xtext has been chosen as the underlying language workbench framework for the development of XMLText. More specifically, Xtext supports naming valida-

tion, concrete syntax semantics, composable syntax/views, and refactoring IDE service. Moreover, Xtext represents an official Eclipse project that is mature and continuously participates in the Eclipse release train with an advanced and stable release schedule [83]. Further, the extensive documentation and active community, i.e. through the Eclipse forum "TMF (Xtext)",[1] of Xtext present valuable assets for the development of third party software.

The following provides a brief overview of different types of language composition and the integration of OCL in Ecore and Xtext.

4.3 Challenges

MDLE frameworks, like Xtext, accelerate the development of modeling languages to a great extent. They cover all aspects of a textual language infrastructure, including the default generation of a lexer, parser, as well as feature-rich IDEs that offer syntax highlighting, error indication, and content assist. At the same time, they provide language engineers with the power to customize entirely the look and feel (i.e. the textual concrete syntax) of modeling languages and tailor them to specific domains and notations—a customization that is not available to XML-based languages due to their fixed concrete syntax. Although implementations of transformations and generators that are offered by Xtext and EMF may be chained together (henceforth referred to as *Default Transformation Chain* and illustrated by Figure 4.2), they expose several gaps which challenge the automated modernization of XML Schema-based languages with modeling languages. More specifically, these gaps occur between the technical spaces [140] of XMLware, modelware, and grammarware and are detailed below.

From XML Schema to Ecore Metamodel. The *EMF XSD Importer* can be employed to produce an Ecore-based metamodel (henceforth referred to as *default metamodel*) from an existing *XML Schema* (cf. ① in Figure 4.2). More specifically, a *default metamodel* is produced in which a schema maps to an `EPackage` in the Ecore language metamodel (cf. Figure 2.2), a complex type definition maps to an `EClass`, a simple type definition maps to an `EDataType`, and an attribute declaration or element declaration maps to an `EAttribute` where the type maps to an `EDataType`, or to an `EReference` where the type maps to an `EClass`. On the one hand and from a modeling perspective, XML Schema is not as expressive as Ecore. For example, the type of the target of a reference is not described in XML Schema. In addressing some of these issues, EMF provides a set of manually applicable extensions to XML Schema definitions in the form of annotations. On the other hand and from a markup perspective, the XML Schema specification enables the expression of serialization details that cannot be represented in terms of Ecore metamodels [197]. Although EMF considers such details by means of `EAnnotations` that are attached to the metamodel for the representation

[1] The Eclipse TMF (Xtext) forum is available at `https://www.eclipse.org/forums/index.php?t=thread&frm_id=27`.

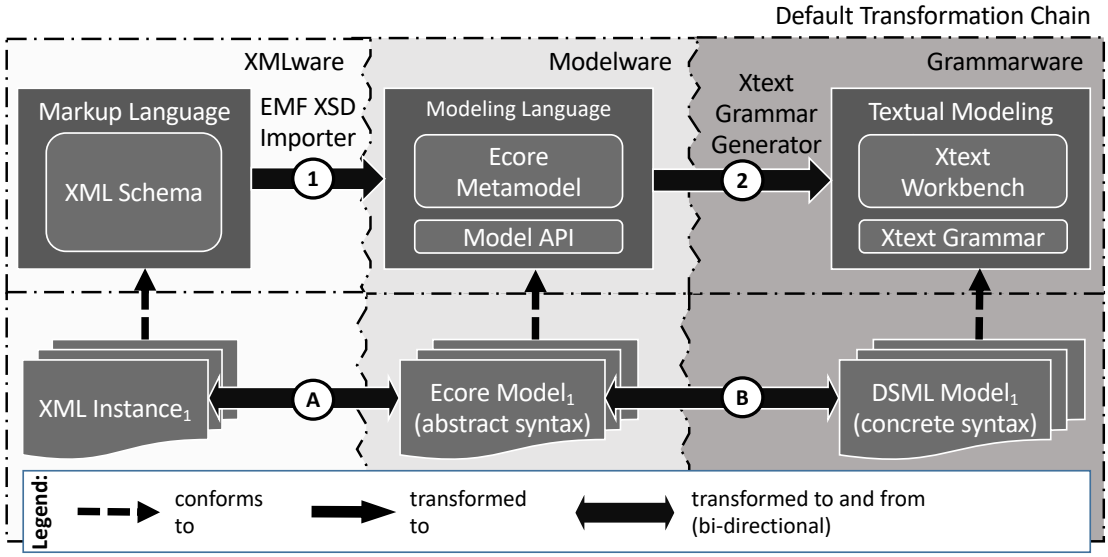

Figure 4.2: Overview of the *Default Transformation Chain*.

of extended metadata, the interpretation of such metadata depends on the capability and implementation of the subsequently employed tool or language workbench.

From Ecore Metamodel to Xtext. Alongside the generation of the *default metamodel* by the *EMF XSD Importer*, the EMF is employed to generate a Java-based *model API* that acts as in-memory representation for metamodel-conforming instances (henceforth referred to as *models*). Additionally, a *default metamodel* is used as input for the *Xtext Grammar Creator* [203] to generate a corresponding Xtext-based grammar (cf. ③ in Figure 4.3). More specifically, the Xtend-based implementation of the *Xtext Grammar Creator* is employed to generate grammar constructs for a limited set of Ecore structures and, in particular, `EClass`, `EEnum`, and `EDataType`. Consequently, the *Xtext Grammar Creator* produces grammar (henceforth referred to as *default grammar*) that typically requires adaptation and extension to enable the construction of grammar-conforming instances (henceforth referred to as *sentences*) that capture any concept that may occur in a particular domain. More specifically, this is achieved by decoupling a *grammar* from its associated *default metamodel* (henceforth referred to as *decoupled grammar*) and by the removal of the metamodel import-statement, the adaptation of grammar rules and terminal rules, and the introduction of a grammar-based metamodel, i.e. a metamodel that is automatically derived from grammar. In this case, the *model API* that has been generated from the *default metamodel*, i.e. established by the *EMF XSD Importer*, is rendered ineffective in respect to the *decoupled grammar* and is thus replaced by a *model API* that is automatically generated from the *decoupled grammar*.

It is important to note that the introduction of *decoupled grammar* causes the loss of backward-compatibility to XMLware applications due to the failure of round-trip transformations at instance level. Therefore, the tool support for round-trip transformations

(cf. Figure 4.2) that is offered by Xtext and EMF for *XML instances* (i.e. conforming to their associated *XML Schema*) and *sentences* that employ *decoupled grammar* is rendered ineffective. Consequently, existing tool support fails to transform *sentences* that conform to *decoupled grammar* into *models* that conform to the *default metamodel* that is imported and employed by the *default grammar*. Moreover, existing tool support for the transformation of *models* that conform to the *default metamodel*, i.e. established by the *EMF XSD Importer*, into valid *sentences* that conform to the *decoupled grammar* similarly fails. In other words, the instance-level transformation that is depicted by Ⓑ in Figure 4.2 fails to handle concepts that have been removed or introduced by the *decoupled grammar* (i.e. in comparison to the *grammar* generated by the *Xtext Grammar Creator* from the *default metamodel*.

Although the production of executable modeling language implementations from XML Schema definitions may be automated by instantiating a chain that employs the *EMF XSD Importer* and the *Xtext Grammar Creator* (henceforth referred to as *Default Transformation Chain*), doing so exposes several gaps between the technical spaces of XMLware, modelware, and grammarware. More specifically, instantiating the *Default Transformation Chain* entails the following steps: Firstly, a source language that is specified by an *XML Schema definition* specification feeds into the *EMF XSD Importer* to yield a corresponding *default metamodel*. Secondly, the obtained metamodel is supplied to the *Xtext Grammar Creator* to produce a *default grammar*. Thirdly, the obtained *default grammar* is employed by the Xtext framework to generate an implementation of the target *modeling language*.

4.4 Requirements

In the next paragraphs, a series of gaps are introduced by the instantiation of the *Default Transformation Chain*, which must be handled by the approach. These gaps are illustrated based on a language for space transformation services that has been introduced as a running example language in Chapter 2.

4.4.1 Identifiers and references

The *EMF XSD Importer* transforms attributes of type `xsd:ID` (exemplified by line 4 in Listing 4.1) to Ecore attributes of type `java.lang.String` that have set the `ID` property to `true`. Likewise, attributes of type `xsd:IDREF` (e.g. line 16) are transformed to Ecore attributes of type `java.lang.String`, whose instances may hold any valid character sequence. Lines 1–2 and 7–8 in Listing 4.2 illustrate the XML Schema concept for identifiers and references by means of an XML instance of the space transportation service elements *engineType* and *stage* (the latter specifies the engine type of a spacecraft *stage*) respectively. Finally, the *Xtext Grammar Creator* produces terminal rules that contain placeholder keywords for both data types `xsd:ID` and `xsd:IDREF` (cf. Listing 4.3). Thus, the *Default Transformation Chain* neglects the concept of `EReference` that is available within the technical space of modelware, and acts as a

native construct for capturing typed references, i.e. references to a particular type that is defined by a metamodel. It is important to note that the XML Schema concepts `ID` and `IDREF` act as identifiers and references that are *global* to the document in which they occur. In other words, the XML Schema specification does not define the concept of type-based referencing that is known in typical GPL implementation, such as the Java programming language.

```
<xsd:complexType name="EngineType">
  <xsd:complexContent>
    <xsd:extension base="sts:NamedElement">
      <xsd:attribute name="engineTypeId" type="xsd:ID" use="required"/>
      <xsd:attribute name="fuelKind" type="xsd:string" use="required"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="Stage">
  <xsd:complexContent>
    <xsd:extension base="sts:NamedElement">
      <xsd:sequence>
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="physicalProperty"
            type="sts:PhysicalProperty"/>
      </xsd:sequence>
      <xsd:attribute name="engineTypeId" type="xsd:IDREF" use="required"/>
      <xsd:attribute name="engineAmount" type="xsd:integer" use="required"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Listing 4.1: XML Schema identifier and identifier reference concept defined within space transportation service complex types *EngineType* and *Stage* respectively.

```
<engineType engineTypeId="M1D" fuelKind="Subcooled LOX / Chilled RP-1" name="
    Merlin1D" />
<engineType engineTypeId="M1DV" fuelKind="LOX / RP-1" name="Merlin1DVacuum" />

<spacecraft countryOfOrigin="USA" launchSite="NKSC"
   manufacturer="SpaceY" name="FalconHeavy" relaunchCostInMioUSD="90">

   <stage engineAmount="9" engineTypeId="M1D" name="FirstStage" />
   <stage engineAmount="1" engineTypeId="M1DV" name="SecondStage" />

   <function>ORBITAL_LAUNCHER</function>

   <physicalProperty type="LENGTH" unit="m" value="70.0" />
   <physicalProperty type="DIAMETER" unit="m" value="70.0" />
   <physicalProperty type="WIDTH" unit="m" value="12.2" />
   <physicalProperty type="MASS" unit="kg" value="1420788.0" />

</spacecraft>
```

Listing 4.2: XML instance of space transportation service elements *engineType* and *stage* employing the XML Schema identifier and identifier reference concept respectively.

```
1 ID0 returns type::ID:
2    'ID' /* TODO: implement this rule and an appropriate IValueConverter */;
3
4 IDREF returns type::IDREF:
5    'IDREF' /* TODO: implement this rule and an appropriate IValueConverter */
         ;
```

Listing 4.3: Grammar produced by *Xtext Grammar Creator* from XML Schema identifier and identifier reference concept.

4.4.2 Mixed content and wildcards

XML Schema defines the wildcard element type `xsd:any` that allows to specify any type of markup content in XML documents. In other words, `xsd:any` enables the extension of an XML document with an element that is not (further) restricted by an XML Schema definition. The *EMF XSD Importer* translates such types to metaclasses containing feature maps that represent ambiguous language concepts, whose handling is delegated to the underlying parser and serializer implementations. As the *Xtext Grammar Creator* neglects the support of such implicitly modeled language concepts, however, these are not represented at grammar level and thus cannot be instantiated. Furthermore, *XML Schema* defines mixed complex type elements, i.e. allowing character data to appear within the body of the element. Similarly to the identifiers and references gap, however, the *Xtext Grammar Creator* produces terminal rules containing a placeholder keyword for any occurrence of `xsd:complexType` with attribute `mixed` set to `true` (cf. line 16 in Listing 4.6), such as for the element that defines the operator of a launch site (cf. lines 7–14 in Listing 4.4). Listing 4.5 exemplifies the structure of mixed content in an XML document that conforms to the XML Schema definition depicted by Listing 4.4.

```
1  <xsd:complexType name="LaunchSite">
2    <xsd:complexContent>
3      <xsd:extension base="sts:NamedElement">
4        <xsd:sequence>
5          <xsd:element maxOccurs="1" minOccurs="1" name="launchSiteId" type="xsd:
               ID" />
6          <xsd:element maxOccurs="unbounded" minOccurs="0" name="physicalProperty"
                type="sts:PhysicalProperty" />
7          <xsd:element name="operator">
8            <xsd:complexType mixed="true">
9              <xsd:sequence>
10               <xsd:element name="operatorName" type="xsd:string" />
11               <xsd:element name="operatorService" type="xsd:string" />
12             </xsd:sequence>
13           </xsd:complexType>
14         </xsd:element>
15       </xsd:sequence>
16       <xsd:attribute name="locationLatitude" type="xsd:decimal" use="required"
             />
17       <xsd:attribute name="locationLongitude" type="xsd:decimal" use="required"
             />
```

```
18      <xsd:attribute name="numberOfLaunchpads" type="xsd:integer" use="required"
            />
19      <xsd:attribute name="operational" type="xsd:boolean" use="required" />
20    </xsd:extension>
21  </xsd:complexContent>
22 </xsd:complexType>
```

Listing 4.4: XML Schema mixed content concept defined within the space transportation service launch site *operator* element.

```
1 <launchSite launchSiteId="NKSC" locationLatitude="-80.65085" locationLongitude="
    28.524058"
2    name="KennedySpaceCenter" numberOfLaunchpads="3" operational="true"
        nextScheduledLaunchTime="16:00:00">
3    <operator>
4       The <operatorName>SpaceY</operatorName> operator offers
5       <operatorService>private-client service</operatorService>
6    </operator>
7 </launchSite>
```

Listing 4.5: XML document instantiating a space transportation service launch site *operator* conforming to an XML Schema definition that employs the concept of mixed content.

```
1 LaunchSite returns LaunchSite:
2    'LaunchSite'
3    launchSiteId=ID0
4    '{'
5       'name' name=String0
6       'locationLatitude' locationLatitude=Decimal
7       'locationLongitude' locationLongitude=Decimal
8       'numberOfLaunchpads' numberOfLaunchpads=Integer
9       'operational' operational=Boolean
10      ('physicalProperty' '{' physicalProperty+=PhysicalProperty ( ","
            physicalProperty+=PhysicalProperty)* '}' )?
11      'operator' operator=OperatorType
12   '}';
13
14 OperatorType returns OperatorType:
15   {OperatorType}
16   'OperatorType'
17   ;
```

Listing 4.6: Grammar produced by the *Xtext Grammar Creator* from an XML Schema definition that employs the XML Schema concept of mixed content.

4.4.3 Data types and restrictions

The W3C Recommendation on XML Schema data types [28] describes a set of built-in data types for different kinds of data, such as numbers, dates, character sequences, identifiers, and references. Listing 4.7 illustrates the XML Schema built-in data types

`xsd:string` and `xsd:dateTime` employed as base data types for the restriction of attributes `missionTitle` (cf. lines 7–13) and `startDateTime` (cf. lines 16–22) within the element `launchEvent` of complex type `LaunchSchedule` (i.e. an element that may occur as part of a space transportation service). More specifically, line one indicates the definition of XML namespace prefix `xsd` and lines nine and eighteen illustrate the use of XML Schema built-in data types `xsd:string` and `xsd:dateTime` respectively. Line nineteen restricts the latter data type to instances that match the defined pattern and, in particular, instances conforming to the format `YYYY-MM-DDTHH:MM:SSZ` (i.e. with `YYYY`, `MM`, `DD`, `HH`, `MM`, `SS`, and `Z` indicating year, month, day, hour, minute, second, and universal time zone respectively). The *EMF XSD Importer* transforms XML Schema data types to custom data types in modelware that are mapped to Java types. The *Xtext Grammar Creator*, however, transforms any metamodel data type to a placeholder terminal symbol that replaces the actual data type. Therefore, the *grammar* created by the *Default Transformation Chain* does not allow the construction of instances that may store values for variables of an arbitrary data type. The consequence of this limitation also impacts the instantiation of the XML Schema concept of restrictions. For example, in the case of a restricted attribute of type `xsd:string`, i.e. assuming that the type definition of the resulting grammar attribute is substituted with the Xtext built-in `STRING` terminal rule, the attribute created in the Xtext grammar is interpreted differently: a character sequence in XML Schema is interpreted with and without its surrounding quotes in Xtext and EMF respectively. Finally, Listing 4.8 illustrates an XML document that conforms to the XML Schema definition depicted in Listing 4.7, i.e. employing the XML Schema concept of data types and restrictions by instantiating a space transportation service *launchSchedule*.

```
1  <xsd:schema xmlns:sts="http://cs.york.ac.uk/ecss/examples/
       spacetransportationservice" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
       targetNamespace="http://cs.york.ac.uk/ecss/examples/
       spacetransportationservice">
2
3    <xsd:complexType name="LaunchSchedule">
4      <xsd:sequence>
5        <xsd:element maxOccurs="unbounded" minOccurs="0" name="launchEvent">
6          <xsd:complexType>
7            <xsd:attribute name="missionTitle" use="required">
8              <xsd:simpleType>
9                <xsd:restriction base="xsd:string">
10                 <xsd:maxLength value="42" />
11               </xsd:restriction>
12             </xsd:simpleType>
13           </xsd:attribute>
14           <xsd:attribute name="launchSiteId" type="xsd:IDREF" use="required"/>
15           <xsd:attribute name="spacecraftId" type="xsd:IDREF" use="required"/>
16           <xsd:attribute name="startDateTime">
17             <xsd:simpleType>
18               <xsd:restriction base="xsd:dateTime">
19                 <xsd:pattern value="\d{4}-\d\d-\d\dT\d\d:\d\d:\d\dZ"/>
20                 <!-- requires UTC as time zone (i.e. indicated by Z), e.g.:
```

```
                2004-04-12T13:20:00Z -->
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
 </xsd:complexType>

</xsd:schema>
```

Listing 4.7: XML Schema data type and restrictions concept defined within the space transportation service *launchSchedule* element.

```
<launchSchedule>
   <launchEvent missionTitle="GPS III-03 navigation satellite deployment"
       startDateTime="2020-01-31T12:00:00Z" spacecraft="FH" launchSiteId="NKSC"
       />
   <launchEvent missionTitle="AFSPC-44 payload deployment (classified)"
       startDateTime="2020-09-30T12:00:00Z" spacecraft="FH" launchSiteId="NKSC"
       />
</launchSchedule>
```

Listing 4.8: XML document instantiating a space transportation service *launchSchedule* by employing the XML Schema concept of data types and restrictions.

```
LaunchSchedule returns LaunchSchedule:
   {LaunchSchedule}
   'LaunchSchedule'
   '{'
      ('launchEvent' '{' launchEvent+=LaunchEventType ( "," launchEvent+=
          LaunchEventType)* '}' )?
   '}';

LaunchEventType returns LaunchEventType:
   'LaunchEventType'
   '{'
      'missionTitle' missionTitle=MissionTitleType
      'launchSiteId' launchSiteId=IDREF
      'spacecraftId' spacecraftId=IDREF
      ('startDateTime' startDateTime=StartDateTimeType)?
   '}';

MissionTitleType returns MissionTitleType:
   'MissionTitleType' /* TODO: implement this rule and an appropriate
       IValueConverter */;

StartDateTimeType returns StartDateTimeType:
   'StartDateTimeType' /* TODO: implement this rule and an appropriate
       IValueConverter */;
```

Listing 4.9: Grammar produced by *Xtext Grammar Creator* from an XML Schema definition that employs the concept of data types and restrictions.

4.4.4 Basic notation customization

XML has been primarily designed as a machine-processible format composed of immutable concrete syntax. Therefore, users of XML-based languages are bound to angle-bracket syntax that is described as verbose and complex in terms of human comprehension. This impedes maintainability [13]. It is important to note that the technical space of XMLware does not foresee a concept to customize concrete syntax and, thus, does not require handling during the exploitation of XML Schema-based language specifications.

4.4.5 Summary

Listing 4.10 depicts a space transportation service instance that is encoded in XML. More specifically, it describes a space transportation service that is composed of two engine types with identifier *M1D* and *M1DV* that are referenced within *FirstStage* and *SecondStage* of the spacecraft named "Falcon Heavy" and identified by *FH*. Furthermore, the modeled spacecraft is described by a set of physical properties that define the length, diameter, width, and mass of the vehicle. Additionally, the model specifies an operational launch site named "Kennedy Space Center" and identified by *NKSC*. Finally, the modeled *launchSchedule* describes two launch events with an individual mission title and start date time, as well as the spacecraft *FH* and launch site *NKSC*.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <sts:SpaceTransportationService
3     xmlns:sts="http://cs.york.ac.uk/ecss/examples/
          spacetransportationserviceXsdSource"
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5     xsi:schemaLocation="http://cs.york.ac.uk/ecss/examples/
          spacetransportationserviceXsdSource spacetransportationservice.xsd ">
6
7     <engineType engineTypeId="M1D" fuelKind="Subcooled LOX / Chilled RP-1" name="
          Merlin1D" />
8     <engineType engineTypeId="M1DV" fuelKind="LOX / RP-1" name="Merlin1DVacuum"
          />
9
10    <launchSite launchSiteId="NKSC" locationLatitude="-80.65085" locationLongitude="
          28.524058"
11       name="KennedySpaceCenter" numberOfLaunchpads="3" operational="true">
12       <operator>NASA</operator>
13    </launchSite>
14
15    <spacecraft spacecraftId="FH" countryOfOrigin="USA" launchSite="NKSC"
16       manufacturer="SpaceY" name="FalconHeavy" relaunchCostInMioUSD="90">
17
18       <stage engineAmount="9" engineTypeId="M1D" name="FirstStage"/>
19       <stage engineAmount="1" engineTypeId="M1DV" name="SecondStage"/>
20
21       <function>ORBITAL_LAUNCHER</function>
22
23       <physicalProperty type="LENGTH" unit="m" value="70.0" />
24       <physicalProperty type="DIAMETER" unit="m" value="70.0" />
```

```
25        <physicalProperty type="WIDTH" unit="m" value="12.2" />
26        <physicalProperty type="MASS" unit="kg" value="1420788.0" />
27    </spacecraft>
28
29    <launchSchedule>
30        <launchEvent missionTitle="GPS III-03 navigation satellite deployment"
              startDateTime="2020-01-31T12:00:00Z" spacecraftId="FH" launchSiteId="
              NKSC" />
31        <launchEvent missionTitle="AFSPC-44 payload deployment (classified)"
              startDateTime="2020-09-30T12:00:00Z" spacecraftId="FH" launchSiteId="
              NKSC" />
32    </launchSchedule>
33
34 </sts:SpaceTransportationService>
```

Listing 4.10: Example XML document instantiating a space transportation service (excerpt).

Table 4.1 summarizes the (lack of) support of XML Schema language concepts by the *Default Transformation Chain* (cf. Figure 4.2) and in particular the transformation of the space transportation service *XML Schema definition* to a corresponding *default metamodel* (cf. ①) and the *default metamodel* to a corresponding *default grammar* (cf. ②) by focusing on language constructs that are instantiated in the space transportation service specification. The column "Supported" denotes whether a particular source language concept (i.e. defined within the space transportation service *XML Schema definition*) is being transformed to a corresponding grammar construct (i.e. captured within the space transportation service *default grammar*) or not, which means that it has been unable to specify corresponding values.

4.5 Approach

The approach proposes bridging XMLware, modelware, and grammarware. Therefore, the goal is to provide a framework that automatically modernizes XML Schema definition to metamodel-based languages, which provide flexible syntax, rich language workbenches, and access to model-based techniques such as transformation, validation, and code generation. In order to achieve this goal, the transformations of the *Default Transformation Chain* are adapted and extended by the introduction of new transformations that transcend the gaps presented in Section 4.4. Figure 4.3 depicts the conceptual overview of the XMLText framework.

Similar to the *Default Transformation Chain*, the first step is to transform a given *XML Schema* to an *Ecore Metamodel* by facilitating the *EMF XSD Importer* ①. In order to tackle the gaps associated with Ecore-based feature maps, which cause the production of empty grammar rules, the *default metamodel* is refactored (henceforth referred to as *adapted metamodel*) by replacing feature maps with generic concrete constructs (cf. ② in Figure 4.3). Next, the *adapted metamodel* is used as input for generating the *grammar*. In order to store actual values for attributes, however, the *Xtext Grammar Creator* (cf. ③

XML Schema			
Concept	**Definition**	**Supported by DTC**	**Notes**
Element	xsd:element	✓	Grammar rule is created
Attribute	xsd:attribute	✓	Feature in grammar rule is created
Containment	(through nesting)	✓	Grammar rule is created and rule call stated
Mixed content	mixed="true"	✗	Ecore feature map is neglected in grammar generation
Wildcard	xsd:any, xsd:anyAttribute	✗	Ecore feature map is neglected in grammar generation
Restriction	xsd:restriction	✗	Non-conforming interpretation
Data type	type="xsd:string" (example)	✗	Placeholder terminal and a TODO-comment replaces data types
Identifier and reference	type="xsd:ID" and type="xsd:IDREF"	✗	Placeholder terminal replaces identifier value

Table 4.1: Overview of *XML Schema* language concepts and their (lack of) support by the *Default Transformation Chain*.

in Figure 4.3) is enhanced by creating, importing, and referencing a library of data types. Moreover, basic customization of the textual concrete syntax of the target modeling language is achieved by the supply of a configurable grammar rule template.

In order to restore forward and backward compatibility of the *adapted metamodel*, introduced by the XMLText framework, it is necessary to refactor existing transformations (cf. (A) in Figure 4.3) to act upon the execution of round-trip transformations on instance level. Therefore, the deserializer (i.e. reading *XML Instance*s and creating in-memory *model* representations that conform to the *adapted metamodel*), as well as the serializer (i.e. storing *model*s as *XML Instance*s), are refactored. As a result of evading the process of decoupling generated grammar from its metamodel, transformation (B) can be reused. With the introduction of transformation (2) and the adaptation of

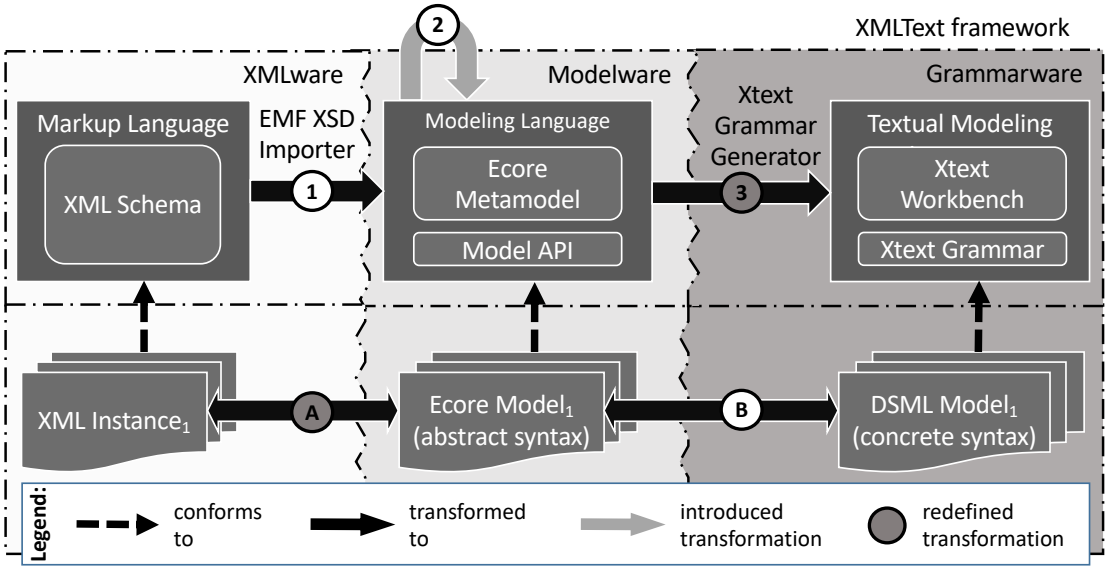

Figure 4.3: Overview of the XMLText framework

transformations ③ and (A), the XMLText framework overcomes the limitations of existing bridges between XMLware and grammarware, thus enabling the automated modernization of XML-based languages with modeling languages as detailed below.

4.5.1 Metamodel generation

The aforementioned *adapted metamodel* addresses the following gaps.

Identifiers and references. The gap associated with global identifiers and references is tackled by transformation ② and, in particular, by replacing model attributes of type `xsd:IDREF` with references to the metaclass `EObject`, which represents the uppermost class in the hierarchy of the Ecore language. Although such references allow objects to reference any kind of object, an `xsd:IDREF` attribute may only reference elements owning an `xsd:ID` attribute. Thus, transformation ③ introduces a necessary refinement in the generation of a grammar rule for every attribute of type `xsd:IDREF`, which (only) allows to reference objects that own an attribute of type `xsd:ID`. Therefore, a new set of terminal rules is defined (cf. the example in Listing 4.11) that enables the subsequently generated IDE to provide support for the referencing of objects through content assist. Listing 4.12 illustrates grammar that is produced by XMLText from the XML Schema concept of identifiers and identifier references. More specifically, the specification of an XML Schema identifier attribute is exemplified by `engineTypeId=ID0` in line five and defined by rule `ID0` in lines 20–21, referring to the import in line one. Further, the specification of XML Schema attributes of type `IDREF` is exemplified by `engineTypeId=SourceElementType` in line fifteen and defined by rule `SourceElementType` in lines 23–24. In both the cases of `ID` and `IDREF`, the

rule `QualifiedName` applies and, thus, enables definition and referencing by means of dot-separated identifier names (cf. lines 29–30).

```
1 IDREFS returns ecore::EString:
2   IDREF (',' IDREF)*;
3
4 IDREF returns ecore::EString:
5   QualifiedName;
6
7 QualifiedName returns ecore::EString:
8   ID (=>'.' ID)*;
```

Listing 4.11: Grammar rules enabling the XML Schema concept of identifiers and identifier references (excerpt).

```
1  import "http://www.eclipse.org/emf/2003/XMLType" as type
2
3  EngineType returns EngineType:
4    'EngineType'
5    engineTypeId=ID0
6    '{'
7      'name' name=STRING
8      'fuelKind' fuelKind=STRING
9    '}';
10
11 Stage returns Stage:
12   'Stage'
13   '{'
14     'name' name=STRING
15     'engineTypeId' engineTypeId=SourceElementType
16     'engineAmount' engineAmount=Integer
17     ('physicalProperty' '{' physicalProperty+=PhysicalProperty ( ","
           physicalProperty+=PhysicalProperty)* '}' )?
18   '}';
19
20 ID0 returns type::ID:
21   QualifiedName;
22
23 SourceElementType returns SourceElementType:
24   '(' referencingAttribute=[ecore::EObject|IDREF])? ')';
25
26 IDREF returns ecore::EString:
27   QualifiedName;
28
29 QualifiedName returns ecore::EString:
30   ID (=>'.' ID)*;
```

Listing 4.12: Grammar produced by *XMLText* from the XML Schema identifier and reference concepts (excerpt).

```
1 SpaceTransportationService {
2
3   engineTypes {
```

```
4     EngineType M1D {
5       name "Merlin 1D"
6       fuelKind "Subcooled LOX / Chilled RP-1"
7     },
8     EngineType M1DV {
9       name "Merlin 1D Vacuum"
10      fuelKind "LOX / RP-1"
11    }
12  }// engineTypes
13
14  spacecrafts {
15    Spacecraft FalconHeavy {
16      manufacturer "SpaceY"
17      countryOfOrigin "USA"
18      relaunchCostInMioUSD 90
19
20      stages {
21        Stage FirstStage {
22          engineAmount 9
23          engineType ( M1D )
24        },// FirstStage stage
25        Stage SecondStage {
26          engineAmount 1
27          engineType ( M1DV )
28        }// SecondStage stage
29      }// stages
30
31    }// FalconHeavy spacecraft
32
33  }// spacecrafts
34
35}// SpaceTransportationService
```

Listing 4.13: Example of a sentence employing the XML Schema identifier and reference concepts.

Mixed content and wildcards. The definition of mixed content and wildcards is rendered by the *EMF XSD Importer* through the creation of attributes of type `EFeatureMapEntry`. Since the *Xtext Grammar Creator* neglects feature maps, however, XML Schema concepts that are rendered as feature maps in modelware are not represented in the resulting *grammar*. In order to cope with the occurrence of feature maps, the metamodel adaptation step (cf. ② in Figure 4.3) replaces feature maps with generic concrete constructs for which grammar rules are generated. Figure 4.4 illustrates the abstract class `AnyGenericConstruct`, extended by the classes `AnyGenericElement` and `AnyGenericText`, that is introduced to enable the support for wildcards, i.e. elements that are not (further) restricted by an associated XML Schema definition. In other words, the former represents the notion of wildcards in terms of `xsd:any`, while the latter represents mixed content appearing either prior to or after an XML tag.

Listing 4.14 depicts the *grammar* generated from the *adapted metamodel* by the

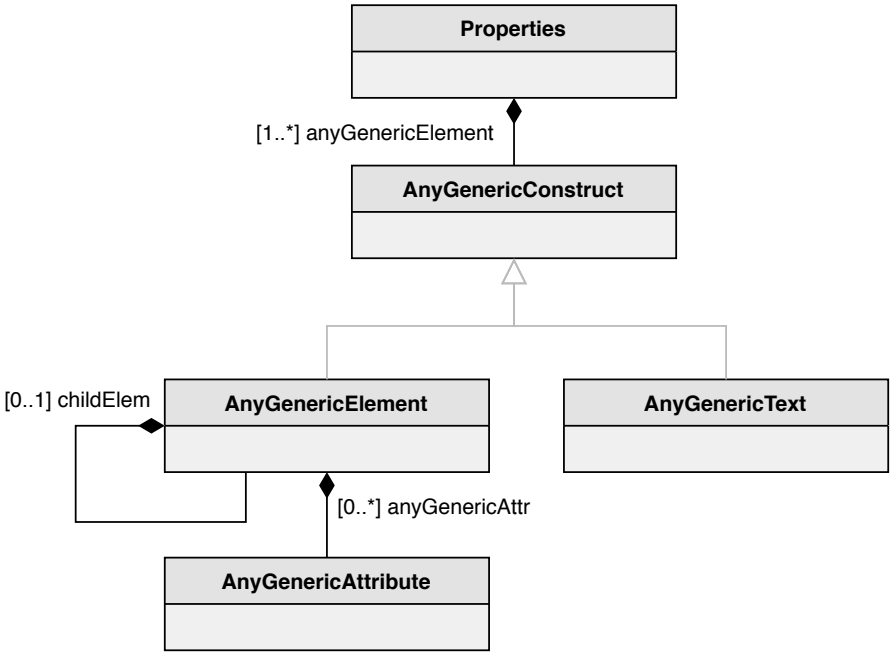

Figure 4.4: Explicit modeling structures replacing feature maps

Xtext Grammar Creator. The abstract class `AnyGenericConstruct` is translated to an equally named grammar rule delegating either to the grammar rule `AnyGenericElement` generated for the metaclass `AnyGenericElement` or to the grammar rule `AnyGenericText` generated for the homonymous metaclass.

In turn, the attributes `elemName` and `elemValue` of `AnyGenericElement` represent the name and value, i.e. any valid character sequence, of the XML tag (cf. lines 12–13 in Listing 4.14) respectively. Moreover, `AnyGenericElement` may contain any number of attributes represented by the `anyGenericAttr` attribute of type `AnyGenericAttribute` (cf. line 15). More specifically, the latter attribute is composed of a name and value that is represented by `attrName` and `attrValue` respectively (cf. lines 25–27). Additionally, an `AnyGenericElement` may contain a number of generically typed element children, as represented by the `childElem` of type `AnyGenericElement` (cf. line 16). Furthermore, the rule `AnyGenericText` (cf. line 18) is capable of holding any character sequence by means of the attribute `textValue` and, thus, the representation of mixed content that might occur in a sentence. Finally, based on the grammar in Listing 4.14, sentences with a structure as depicted in Listing 4.15 can be constructed.

```
1 Property returns Property:
2    'Property'
3    '{'
4       anyGenericElem+=AnyGenericConstruct ( ',' anyGenericElem+=AnyGenericConstruct)
            *
5    '}';
6
```

```
7  AnyGenericConstruct returns AnyGenericConstruct:
8     AnyGenericElement | AnyGenericText;
9
10 AnyGenericElement returns AnyGenericElement:
11    {AnyGenericElement}
12    elemName=STRING
13    (':' elemValue=STRING)?
14    '{'
15    (anyGenericAttr+=AnyGenericAttribute ( ',' anyGenericAttr+=AnyGenericAttribute)*)
          ?
16    ('{' childElem+=AnyGenericElement ( ',' childElem+=AnyGenericElement)* '}' )?;
17
18 AnyGenericText returns AnyGenericText:
19    {AnyGenericText}
20    '{'
21       (textValue=STRING)?
22    '}';
23
24 AnyGenericAttribute returns AnyGenericAttribute:
25    attrName=STRING
26    ':'
27    attrValue=STRING;
```

Listing 4.14: Grammar generated from `Property` class containing an attribute of type `AnyGenericConstruct` as depicted in Figure 4.4.

```
1 Property {
2   "operator": "The " {
3     operatorName: "SpaceY" {
4       " operator offers " {
5         operatorService: "private-client service"
6       }
7     }
8   }
9 }
```

Listing 4.15: Example of a sentence instantiating the XML Schema concept of mixed content within the technical space of grammarware.

Data types. As mentioned above, the *Xtext Grammar Creator* does not create rules for metamodel data types. Therefore, both the specification of terminal rules, as well as calls to these rules, are missing. In order to overcome this limitation, a library for Xtext data types (henceforth referred to as *type library*) is constructed, which defines terminal rules for XML Schema data types. Further, transformation ③ in Figure 4.3 is refactored so that the terminal rules defined by the *type library* are employed by the grammar of the modeling language being constructed. Listing 4.16 depicts an excerpt of the *type library* defining a terminal rule for a valid time value, as outlined in the XML Schema specification [202].

```
1 import "http://www.eclipse.org/emf/2003/XMLType" as type
2
```

```
3  terminal DATETIME returns type::DateTime:
4    (
5      ('1'..'9')('0'..'9')('0'..'9')('0'..'9')
6      '-'
7      ('0'..'1')('0'..'9')'-'('0'..'3')('0'..'9')
8      'T'
9      ('1'..'0')('0'..'9')':'('1'..'0')('0'..'9')':'('1'..'0')('0'..'9')
10     '-'
11     ('1'..'0')('0'..'9')':'('1'..'0')('0'..'9')
12     'Z'
13   );
```

Listing 4.16: *Type library* terminal rule for the representation of date and time in universal time zone.

XML Schema restrictions. The handling of restrictions in XML Schema definitions is presented in Section 5.5.6 (i.e. illustrating the integration of XMLText with the approach for consistency-achieving IDEs, outlined in Chapter 5). In brief, the support for XML Schema restrictions is extended by the transformation of restrictions into formal constraints and the facilitation of language definitions that are composed of metamodels and formal constraints for the automated generation of modeling language implementations with enhanced IDEs (i.e. enabling precise model validation, consistency-preserving content assistance, and consistency-restoring model repair).

4.5.2 Basic notation customization

In the following, an initial approach towards the customization of notation that overcomes rigid XML syntax is illustrated. The notation that is used to specify a model of the textual modeling language depends on the definition of terminal symbols in the language grammar. The *Default Transformation Chain* employs the *Xtext Grammar Creator* and, thus, creates a generic concrete syntax. Although the necessity to change symbols in language implementations has been raised [84], the Xtext framework language engineers manually refactor either the generated grammar rules or the transformation represented by the *Xtext Grammar Creator*. In the former case, however, the grammar rules of every generated grammar have to be refactored manually, i.e. a process that neglects reuse. Although the latter case, i.e. the refactoring of the *Xtext Grammar Creator* transformation, is applicable to arbitrary Ecore-based metamodels, it has to be repeated for every style of customized concrete syntax.

XMLText enables the modification and exchange of different concrete syntax specifications through the introduction of a template mechanism that is composed of exchangeable template files that define a set of textual concrete symbols that are to be employed by the target language. For example, the notation of variable value specifications is defined by the Human-Usable Textual Notation (HUTN) [185] and YAML Ain't Markup Language (YAML) [17] in terms of equal signs and colons respectively. Hence, the character or character sequence that is defined in line two of Listing 4.17 renders the terminal symbol for

variable value specification (cf. `VariableValueSpecificationTerminalSymbol`) to be represented by a colon.

```
1 InterPackageReferenceTerminalSymbol = '.'
2 VariableValueSpecificationTerminalSymbol = ':'
3 PropertyMemberOpenTerminalSymbol = '{'
4 PropertyMemberCloseTerminalSymbol = '}'
```

Listing 4.17: Example of basic notation customization template.

The result of employing the basic notation customization template defined by Listing 4.17 is illustrated by the language grammar in Listing 4.18. For example, lines 5–8 and 11 employ the terminal symbol for variable value specification.

```
1  LaunchSite returns LaunchSite:
2     'LaunchSite'
3     launchSiteId=ID0
4     '{'
5        'name' VariableValueSpecificationTerminalSymbol name=STRING
6        'locationLatitude' VariableValueSpecificationTerminalSymbol locationLatitude=
             Decimal
7        'locationLongitude' VariableValueSpecificationTerminalSymbol
             locationLongitude=Decimal
8        'numberOfLaunchpads' VariableValueSpecificationTerminalSymbol
             numberOfLaunchpads=Integer
9        'operational' operational=Boolean
10       ('physicalProperty' PropertyMemberOpenTerminalSymbol physicalProperty+=
             PhysicalProperty ( "," physicalProperty+=PhysicalProperty)*
             PropertyMemberCloseTerminalSymbol )?
11       'operator' VariableValueSpecificationTerminalSymbol operator=OperatorType
12    '}';
13
14 QualifiedName returns ecore::EString:
15   ID (=>InterPackageReferenceTerminalSymbol ID)*;
16
17 terminal VariableValueSpecificationTerminalSymbol:
18   ':';
19
20 terminal PropertyMemberOpenTerminalSymbol:
21   '{';
22
23 terminal PropertyMemberCloseTerminalSymbol:
24   '}';
25
26 terminal InterPackageReferenceTerminalSymbol:
27   '.';
```

Listing 4.18: Grammar of space transportation service *LaunchSite* generated by *XMLText* by applying the basic notation customization template depicted in Listing 4.17.

Note that the implementation of the basic notation customization in XMLText is substituted by a more powerful approach, i.e. the Ecore Concrete Syntax Specification Language (ECSS) language that is presented in Chapter 6.

4.5.3 Modeling assistance

This section presents work on the integration of XMLTEXT and the implementation of a modeling and metamodeling assistance approach [192] that is henceforth referred to as *modeling assistant*. More specifically, XMLTEXT is integrated into the modeling assistant to enable developers of modeling languages to import and map language concepts from XML-based artifacts to shared domain-specific concept repositories during the phase of language development. In other words, this section illustrates an approach for bridging the gap between language engineering and domain expertise modeling by empowering the process of ingraining domain-specific knowledge embodied by artifacts originating from the technical space of XMLware.

The remainder of this section is structured as follows. First, the heterogeneous modeling assistance approach is introduced within the context of MDE. Second, an overview of the approach and its motivation is provided. Third, the handling of heterogeneous sources, such as XML Schema definitions, by the employment of a common data scheme and the management of constraints is presented. Forth, the architecture of the assistant as well as its integration with the XMLTEXT framework is described in a nutshell. Fifth, the core components of the assistant are presented briefly. Finally, the work on the modeling assistant integration is summarized.

Section 4.7 presents an evaluation of XMLTEXT within the context of modeling assistance and in particular the development of an industry standard-conforming modeling language. The approach is evaluated in more detail in three different areas, which include the usefulness of language engineering, the capability to capture information mapped by the assistant for the technical spaces of XMLware and modelware, and the proficiency of integration of the XMLTEXT framework.

Introduction

The field of MDE advocates the active use of models throughout the life cycle of software development, thus, encouraging the application of models for the definition, analysis, testing, simulation, execution, code generation, and maintenance of software [31, 191, 54]. In general, models may be constructed with the use of GPMLs, such as the UML [94], or modeling languages [152], i.e. containing tailored domain-specific primitives and concepts that accurately abstract a domain and may lead to simpler and more intentional models. The abstract syntax of a modeling language is described by a metamodel, i.e. also a model, typically described by means of a class diagram that acts as a description for any set of models that are considered valid. Thus, the construction of models and metamodels is a recurrent and central activity in MDE projects [106]. Moreover, high-quality models and metamodels are pivotal for the success of MDE projects, as they capture the most important concepts of a domain or describe the features of a system.

Nevertheless, metamodels are commonly constructed from the ground up, with no mechanisms for the reuse of existing domain knowledge. This situation comes into contrast with modern programming IDEs that offer support for code completion or the use of a

given API [182]. In the field of MDE, however, modelers carry the burden of creating metamodels from scratch. For this reason, modelers would greatly benefit from flexible access and reuse of existing knowledge that is available in a domain and, in particular, from various technological spaces such as XMLware, modelware, and ontologies.

In order to improve this situation for the technical space of XMLware, the XMLText framework is integrated into a modeling assistant approach, which attains information from the technical space of XMLware by means of extraction and mapping of XML Schema definitions and XML documents to a common data scheme. Subsequently, the modeling assistant provides a platform to visualize uniformly and query heterogeneous information, as well as prioritize and aggregate query results in order to facilitate the construction of novel (meta)models. The modeling assistant approach is manifested in the Extremo tool and extended with the XMLText framework by means of an Eclipse extension point for the integration of information sources that originate from the technical space of XMLware. The usefulness of the integration of XMLText in the modeling assistant (henceforth referred to as *XMLText assistant integration*) has been evaluated during the creation of a modeling language for industry standard-conforming production systems and, in particular, by employing the eCl@ss standard [63]. Moreover, the XMLText assistant integration is evaluated according to its ability to be integrated with the modeling assistant in order to exploit domain knowledge from XML Schema definitions and XML instances, as well as to employ domain knowledge for the construction of an industry standard-conforming conveyor-belt system.

Motivation and overview

Many technical tasks in software engineering require access to knowledge found in a variety of formats, ranging from documents in natural language, to semi-structured and structured data. There is a current trend to make such information readily available and easily embedded in different types of artifacts generated during the software construction process [150]. For example, in the programming community, efforts are made to profit from code repositories, such as Github, and Q&A sites, such as StackOverflow, in order to automate tasks associated with coding and documentation [14, 177]. Some of these approaches are based on artifact collection, for example by distributed mining and processing of software repository resources [188, 128]. Following this trend, the objective of the modeling assistant is to enable the engineer to gather and query heterogeneous resources and make them available for the construction of novel (meta)models. In general, the task of creating high-quality metamodels is complex due to the involvement of two distinctive roles: a domain expert, i.e. having in-depth knowledge of a particular domain, and a metamodeling expert, i.e. experienced in the design of class-oriented metamodels. Nevertheless, the metamodeling expert is frequently unsupported in the construction of metamodels and, in particular, during the process of decision-making, which is based on tacit domain knowledge or under-specified language requirements. In this scenario, the metamodeling expert also assumes the role of domain expert and, thus, may misinterpret or omit domain concepts that will subsequently compromise the quality of the resulting

metamodel.

Metamodels within a domain are not entirely different from one another due to recurring patterns and the application of common idioms in the representation of concepts [15]. For example, during the construction of a language to describe behavior, designers typically resort to accepted specification styles, which include variants of languages such as state machines, workflows, and rule-based or data-flow languages, enriched with the concepts of a particular domain. The metamodeling expert or designer of a language can obtain this information from sources such as metamodels, XML Schema definitions, and XML documents. Moreover, having access to a variety of information sources helps to obtain the necessary domain knowledge, vocabulary, and technical terms required to build a metamodel for a particular domain. This situation also applies to the construction of metamodel-conforming instances. In this case, it may be helpful to support the querying of information sources and knowledge bases, e.g. by enabling the filtering of relevant information from a metamodel based on the use of a specific data type.

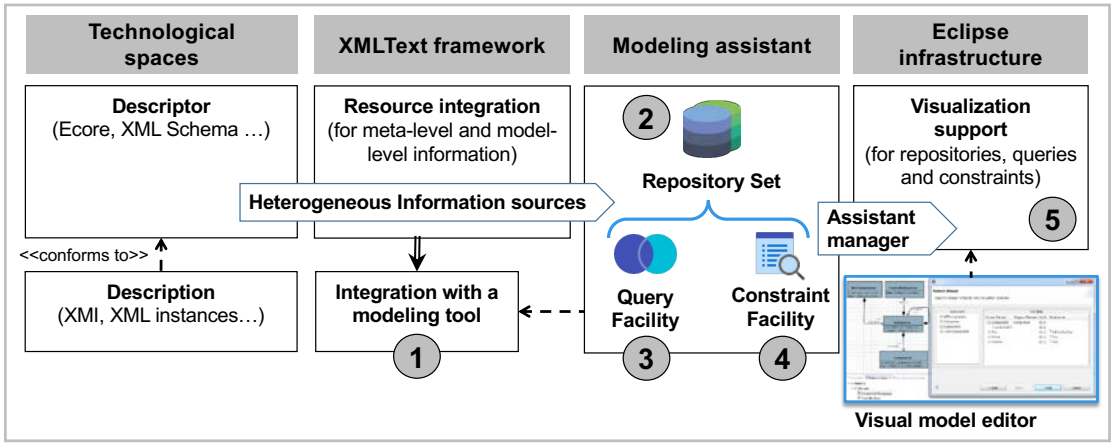

Figure 4.5: Approach overview

For this purpose, the modeling assistance approach illustrated in Figure 4.5 may be useful for the creation of models at any meta-level. The XMLText assistant integration is based on the integration of XMLText with the modeling assistant tool. Although XMLText is independent from modeling tools, such as the modeling assistant, it has been designed to integrate easily with a wide range of tools (label 1). The modeling assistant is based on the creation of a set of repositories (label 2), in which heterogeneous data descriptions, such as XML Schema definitions and Ecore metamodels, and heterogeneous data sources, such as XMI-based models and XML documents, are injected. More specifically, heterogeneous information is represented in a uniform way through the application of a common data scheme. Further, extensible facilities for the uniform and flexible query of the repository (label 3) are provided alongside basic services for synonym search and word sense analysis. Heterogeneous constraints may also be persisted and evaluated in the repository by the use of an external facility (label 4). The results of

the queries for each source in the repository may be aggregated and ranked according to individual suitability and relevance. Subsequently, information sources and query results may be visualized, e.g. by means of a visual model IDE or viewer (label 5).

Several scenarios have been identified in which the integration of XMLText in the modeling assistant may be useful. They can be generally classified in three categories: firstly, in creating models and metamodels; secondly, in creating artifacts describing a set of other artifacts, such as model-based product lines [53, 178] or model transformation reuse [52]; finally, the integration of XMLText in the modeling assistant may be useful:

- as support for the development of new domain metamodels. This way, domain concepts and vocabulary can be sought in external sources specified by means of XML Schema definitions and XML documents;
- in creating models for a particular domain. In this case, model elements conforming to the same or similar metamodel can be incorporated into the model being built, and heterogeneous information sources can be queried in order to extract concrete data values;
- in designing a "concept" metamodel [52], i.e. a minimal metamodel that gathers the core primitives, such as workflow specifications, within a domain. Furthermore, concepts can be used as the source metamodel of a model transformation, and reused, so they can be bound to a particular metamodel. This task implies the querying and understanding of a variety of metamodels for a particular domain and, as such, the assistant becomes useful;
- in aggregating multiple existing models into a model-based product line [53, 178]. In this approach, a description of the space of possible features of a software system is created, typically through a feature model. The choice of features implies the selection of a software model variant. This way, there is a need for understanding an existing family of models and for describing their variability. One possible approach is to merge or superimpose all model variants (leading to a so-called 150% model) and use "negative variability", i.e. the removal of deselected artifacts [53, 178];
- in detecting "bad smells" [154] or signs of bad modeling practices [2], such as isolated nodes and abstract classes with no children in a set of resources, through the execution of technology-specific or domain-specific queries over a repository.

Common data scheme

The common data scheme (cf. Figure 4.6) focuses on the need to capture information from several data sources stored in a variety of formats, and is arranged by the provision of a mechanism to organize and classify information from an arbitrary number of meta-levels. Heterogeneous sources, such as XML Schema definitions and XML documents, are then integrated by establishing transformations into the common data scheme. Section 4.5.3

presents the architecture design with the support for the integration of heterogeneous information sources.

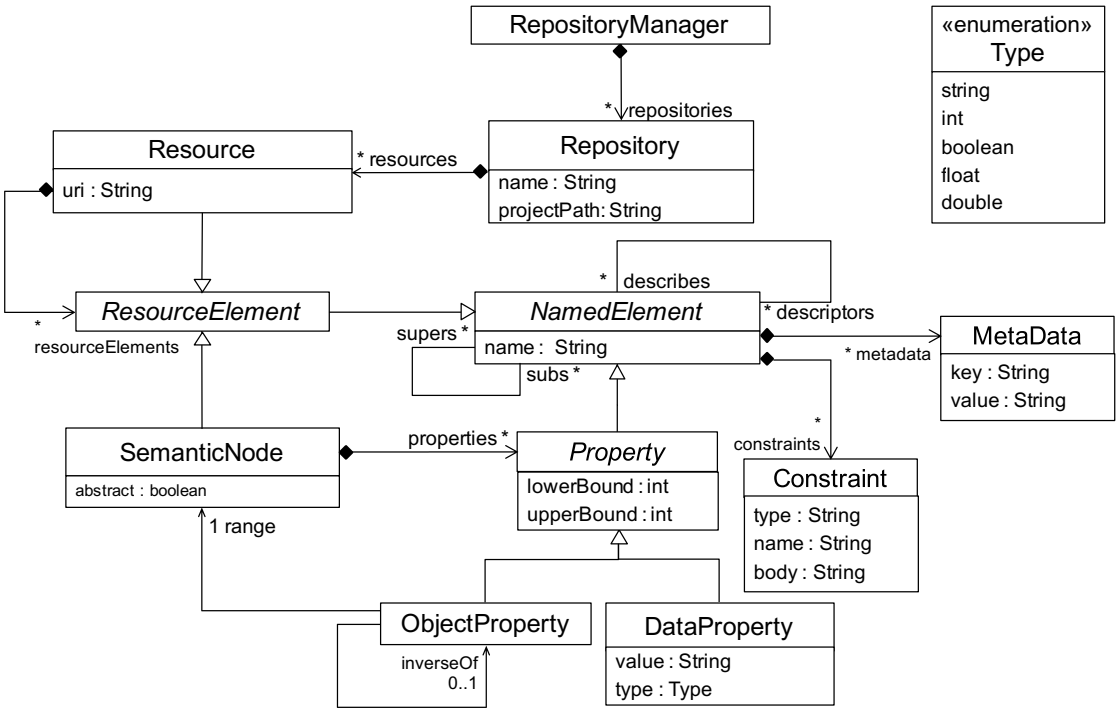

Figure 4.6: The common data scheme (package *dataModel*)

More specifically, each file or information source is represented by a *Resource*, which can be aggregated into *Repository* objects. Each resource contains a collection of *SemanticNode*s, i.e. entities that are added to account for different technical spaces [140]. In other words, semantic nodes are elements that gather knowledge from (original) source elements and, hence, serve as an abstraction for managing heterogeneous information. *Resource*s can be nested to account for hierarchical organization. For example, in the Ecore language, models and packages are nested.

Resources, nodes, and properties are *NamedElement*s that are identified by their name and can act both as descriptors, i.e. be a type or class, and be described by other elements, i.e. be instances of other elements. Further, the common data scheme may accommodate instance-relations found in heterogeneous technical spaces, which include descriptor-only elements, such as meta-classes in metamodels; described-only elements, such as objects in models; and elements that are both descriptors of other elements and at the same time are described by others, such as clabjects[2] applied in multi-level modeling [11]. Hence, the common data scheme is meta-level agnostic, as the same

[2]A clabject is a model element that has both type and instance facets and, hence, holds both attributes, i.e. field types or classes, and slots, i.e. field values or instances.

concepts may be represented by both models and metamodels, both classes and objects, and both attributes and slots, which may increase simplicity and generality [9].

*NamedElement*s may be associated with *MetaData* to account for technology-specific details that do not fit into the common data scheme. For example, when reading an Ecore metamodel, it may be necessary to store whether an object property represents a composition or the potency[3] of an element respectively. Additionally, a *Resource*, which is also a *NamedElement*, may manifest conformance relations between artifacts, such as models and metamodels, XML Schema descriptions and XML documents, thus permitting the representation of a simple mega-model [61].

*SemanticNode*s can take part in generalization hierarchies with the support of multiple inheritance and may be tagged as *abstract*. Generalization hierarchies are supported at any meta-level, such as class-level and object-level, to account for approaches where inheritance may occur at the level of objects [56]. A node is made of a set of properties (*DataProperty*) and a set of links to other nodes (*ObjectProperty*), both of which define cardinality intervals. Similar to nodes, properties unify the concepts of *attribute*, i.e. a specification of required properties in instances, and *slot*, i.e. a holder for values. The common data scheme supports a range of neutral basic data types (*Type* enumeration), such as *string*, *int*, *boolean* and *double*. For generality, the value of the property is stored as a *String*. Finally, any element may have *Constraint*s attached. The handling of heterogeneous constraints is presented in Section 4.5.3.

Table 4.2 illustrates how individual technologies and in particular elements at different meta-levels are mapped to the common data scheme. More specifically, meta-levels of the technical spaces of modelware and XMLware are represented by Ecore metamodels, i.e. a widely used implementation of the Meta Object Facility (MOF) OMG standard [168] distributed as a core component of the EMF [197], and XML Schema definitions, i.e. XML-based documents conforming to the W3C XML Schema specification [202].

EMF supports two meta-levels and the mapping is direct. Figure 4.7 shows a schema of how the translation from Ecore into the common data scheme is performed. At the top, the figure shows that both a metamodel (i.e. named *ecore* model in Figure 4.7) and a model (i.e. serialized in XMI format) are transformed into a *Resource*. In this case, a *Resource* object from a model is described by a *Resource* of a metamodel. Similarly, elements within both the metamodel and model follow the latter translation scheme. Both *EClass* and *EObject* are transformed into *SemanticNode* with a suitable *descriptor* relation, and the same applies for references and attributes. At the model-level (i.e. EMF compiled mode), links and slots are represented as Java fields, whose value is obtained via getter methods.

Figure 4.8 depicts the translation of XML Schema definitions, i.e. acting as language definitions, and XML documents. This case is conceptually similar to EMF. At the top, the figure shows that an XML Schema definition, i.e. serialized as XML Schema

[3] The potency of an element is represented by zero or a positive number that accounts for the instantiation depth of an element at subsequent meta-levels [11].

Common data scheme	Modelware	XMLware
Meta-level (types)		
Resource	Ecore file/EPackage	XML Schema file
SemanticNode	EClass	xsd:element
Property (abstract)	EStructuralFeature	xsd:complexType xsd:element
ObjectProperty	EReference	Nested xsd:element IDREF attribute
DataProperty	EAttribute	xsd:attribute
Property.supers	EClass.eSuperTypes	xsd:element type attribute
Constraint	OCL EAnnotation	xsd:restriction
Model/data level (instances)		
Resource	XMI file	XML file
SemanticNode	EObject	XML element
ObjectProperty	Java reference	Nested xsd:element IDREF attribute
DataProperty	Java attribute	XML attribute

Table 4.2: Mapping different representation technologies to the common data scheme

definition (XSD) file, is transformed into a *Resource* in the common data scheme. Then, as a result of XML Schema definitions being described in the XML format, the *Resource* object from the document is described by the *Resource* of the schema as indicated by the *descriptor* association from *Resource* to *Resource*.

The elements within an XML Schema definition and an XML document follow a similar translation. For example, an XML element is transformed into a *SemanticNode* with a *descriptor* relation to the respective *xsd:element*. Moreover, an XML element or XML attribute is transformed either into an *ObjectProperty* or a *DataProperty* depending on the type it specifies. More specifically, in the case of type *xsd:IDREF* and any other type, an *ObjectProperty* and *DataProperty* is created respectively. The former and latter are indicated by a *descriptor* association from *ObjectProperty* to *ObjectProperty* and a *descriptor* association from *DataProperty* to *DataProperty* respectively.

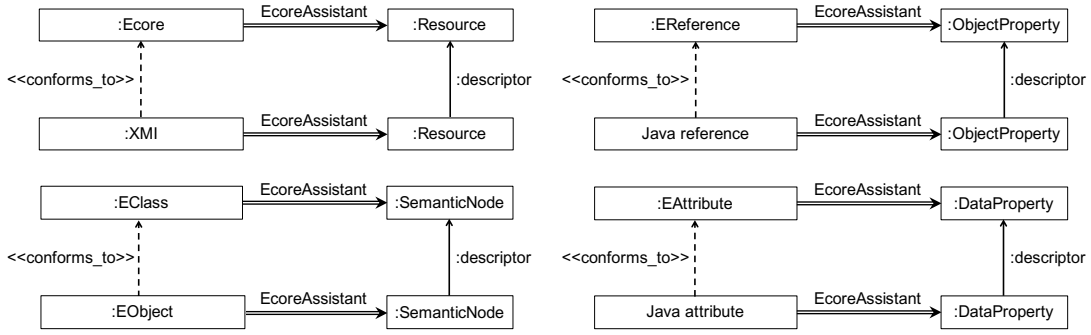

Figure 4.7: Injecting Ecore (meta)models into the common data scheme.

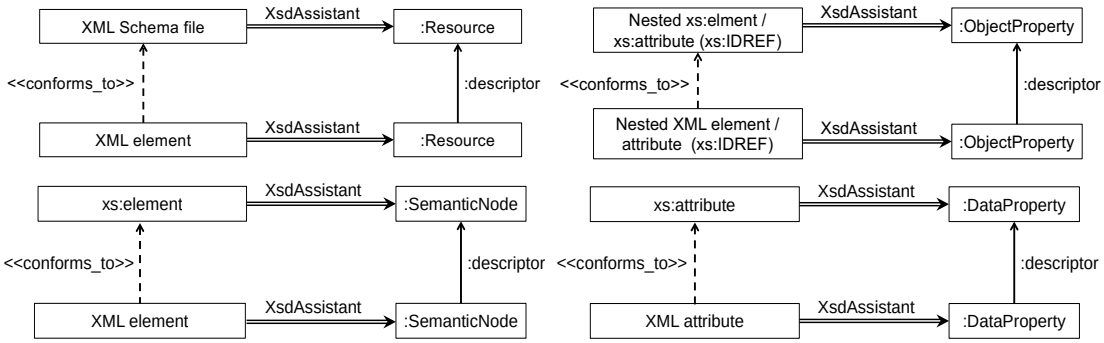

Figure 4.8: Injecting XML Schema descriptions into the common data scheme.

Heterogeneous constraints

As described in the previous section, the common data scheme creates *SemanticNodes* to store the knowledge found in elements, properties, and links to other nodes.

In addition, restrictions may be specified that limit the set of possible instantiations. For example, a class in a metamodel may contain OCL invariants that all valid objects of that class must obey. Similarly, elements in XML Schema definitions may contain restrictions on properties of a given data type. These may be used to define patterns on character sequences (regular expressions), restrictions on the length of character sequences (min/max/exact length), and handling of white spaces, among others.

Figure 4.9 depicts the facility to store heterogeneous constraints in the common data scheme, as well as the mechanism for constraint interpretation. Constraints may be attached to any *NamedElement*, i.e. extended by *Resource*, *SemanticNode* and *Property*, and thus cover the basic restriction scenarios that may occur in a technical space. Moreover, a *Constraint* has a *type* that identifies the type of the constraint, such as OCL, as well as the name and body, i.e. capturing the actual constraint specification. The support for the evaluation of heterogeneous constraints, such as OCL invariants and XML Schema restrictions, may be provided by extending the modeling assistant.

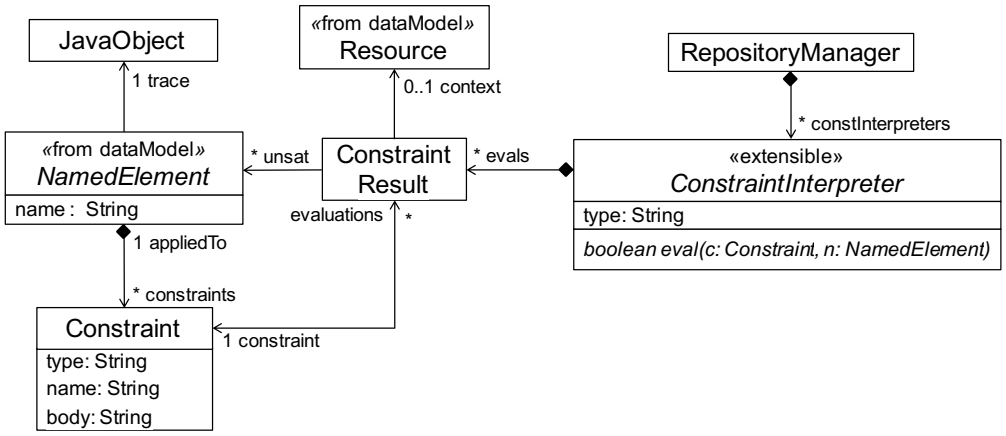

Figure 4.9: Constraint storage and interpretation mechanism.

Chapter 5 presents the contribution of this thesis in regards to the automated generation of consistency-achieving language implementations from formally constrained language structures and, in particular, the implementation of a constraint interpreter, which may extend the modeling assistant with the capability to evaluate OCL-based constraint specifications. In general, constraint interpreters extend the class *ConstraintInterpreter*, declare support for specific constraint types, and implement the *evaluate* method. This method receives a constraint and an instance of the element to which the constraint is attached, and returns a boolean indicating whether the element satisfies the constraint. Similar to query results, constraint results are reified using *ConstraintResult* objects, i.e. holding elements that do not satisfy the constraint, and organized in the context of the enclosing *Resource*. Similar to format assistants, the addition of constraint interpreters is realized by means of Eclipse extension points, i.e. interfaces that may be implemented by external components that provide customized functionality.

Architecture and tool support

The modeling and metamodeling assistance tool is made of a *Core* component, which provides support for the common data scheme and includes subcomponents for the integration assistance, search, and constraint interpretation, as depicted in Figure 4.10. The XMLText framework contributes to the *XsdAssistant* by providing access to information from XML Schema descriptions and XML documents through the employment of the *XMLware to modelware facility* and the *XMLware resource handler*. Further, the *XsdAssistant* contributes data to the *Repository manager* in the *Core* component of the modeling assistant. In order to allow for scalability, the repository is persisted using NeoEMF [18], a model persistence solution designed to store models in NoSQL datastores. NeoEMF is based on a lazy-loading mechanism that transparently brings model elements into memory only when they are accessed, and removes them when they are no longer needed. The *UI* component permits visualization and interaction with the resources

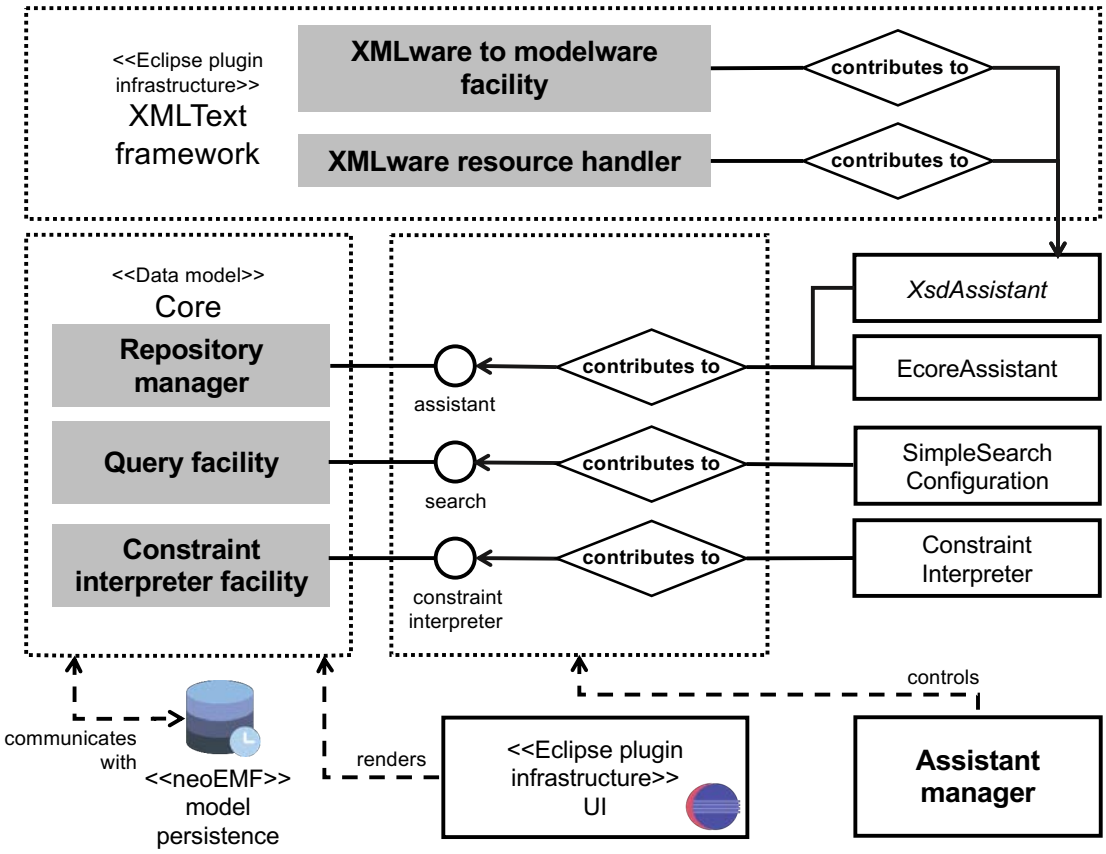

Figure 4.10: Architecture of modeling assistant with XMLText integration.

and query results. A set of views, commands, wizards, and actions has been defined to control access to the repository, the list of query results, and the constraint validation facilities. By extending this component, it is possible to integrate the modeling assistant with external modeling tools.

Core components and integration mechanism

Figure 4.11 shows the component model of the modeling assistant with XMLText integration. The *Core* components of the modeling assistant consist of the *Repository manager*, i.e. controlling access to the common data scheme and managing the process of integrating information sources, such as those provided by XMLText assistant integration for the technical space of XMLware; the *Query facility*, i.e. enabling the extension of the query mechanism; and a *Constraint interpreter facility*, i.e. enabling the support and extension of different constraint formats and interpreters.

For each part, a set of extension points have been defined. For example, the *assistant* extension point enables support for new data formats, i.e. realized by the implementation

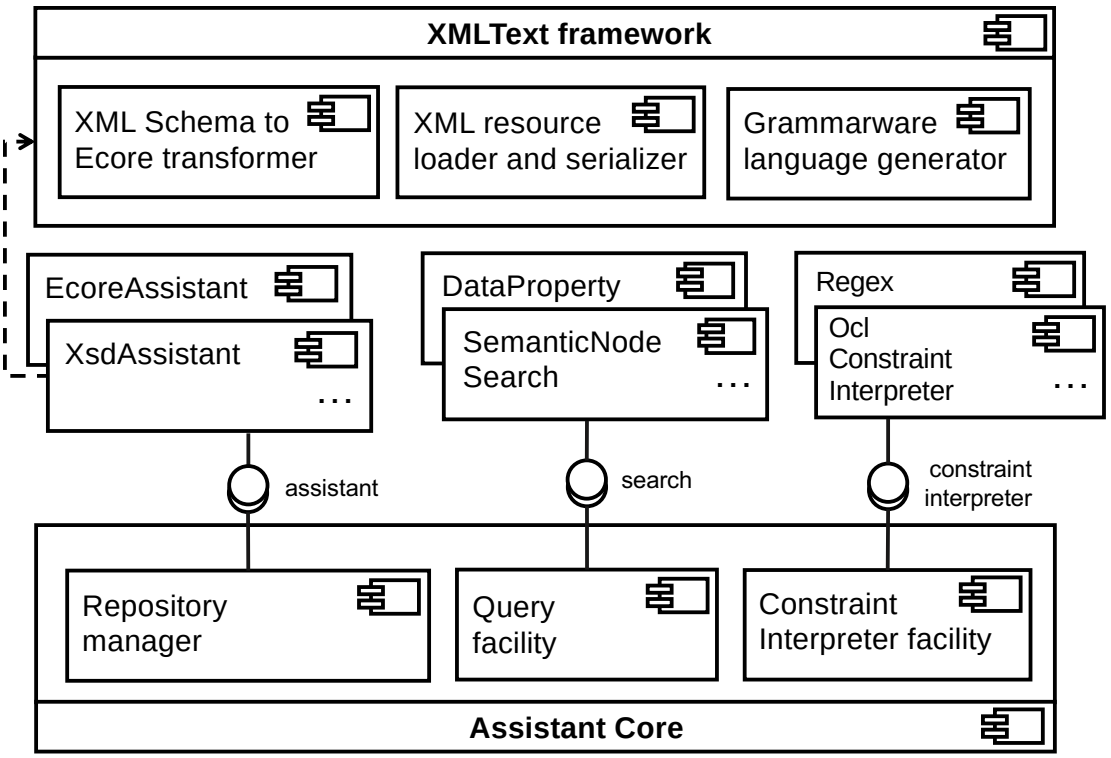

Figure 4.11: Component model of the modeling assistant with XMLText integration.

of a mapping from the format-specific structure, such as XML Schema definitions and XML documents, to the common data scheme, as described in Section 4.5.3. A predefined set of assistants, such as for EMF-based information sources, as well as a framework for the creation of assistants, i.e. permitting the conceptual organization of assistants as model-to-model transformations, has been created. Hence, an abstract class with a number of methods is provided and must be overridden by a particular assistant to act as rules in a transformation. With these methods, it is possible to create one or more elements in the common data scheme, supporting one-to-many and many-to-one mappings. In order to facilitate the construction of assistants, it is possible to define class hierarchies and, thus, reuse import functionality.

In addition, transformations between two individual technical spaces may be reused. For example, information that originates from the technical space of XMLware may be imported into the modeling assistant by instructing the XMLText framework to contribute resources to the technical space of modelware, as opposed to the specification of a customized mapping to the common data scheme within an assistant implementation, such as the *XsdAssistant*. More specifically, XML Schema elements and attributes may be transformed into respective Ecore-based metamodel classes and attributes, and subsequently processed by the *EcoreAssistant*. In this manner, a chain process is constructed, which translates XML Schema definitions into instances that conform to

the presented common data scheme.

The *search* extension point provides extensibility for queries. Conceptually, user-defined queries extend the corresponding metamodel. The extensions allow the definition of custom, predicate-based, and composite queries by subclassing corresponding classes. Finally, the *constraint interpreter* extension point permits the contribution of support for the evaluation of new types of constraints, extending the metamodel in Figure 4.9.

Summary

This chapter presented the integration of the XMLText framework in an assistant for modeling and metamodeling. The XMLText assistant integration enables the modeling assistant to gather information from the technological space of XMLware by facilitating the exploitation of XMLware artifacts and, in particular, by importing XML Schema definitions and XML documents, as well as their respective mapping to the common data scheme. The latter has been created to enable unified persistence, querying, and visualization of information originating from heterogeneous sources. The results of the presented evaluation, i.e. based on the eCl@ss standard and SigPML, illustrate the usability of the XMLText assistant integration alongside the modeling assistant in constructing a production system modeling language for the specification of industry standard-conforming conveyor-belt systems. Further, the results indicate that the common data scheme is able to capture heterogeneous information and, in particular, information that originates from XMLware and modelware. Finally, the integration of the XMLText framework within third-party modeling tools through the implementation of the *XsdAssistant* has been achieved with few lines of code, whilst significantly more effort was required for the implementation of the mapping of modelware artifacts to a respective third-party data format, such as the common data scheme of the modeling assistant at hand.

4.5.4 Model transformation

Listing 4.19 shows the result of employing the XMLware to grammarware instance-level forward transformation provided by XMLText on the example of the space transportation service XML document in Listing 4.10.

```
1  SpaceTransportationService {
2
3     engineTypes {
4        EngineType M1D {
5           name "Merlin 1D"
6           fuelKind "Subcooled LOX / Chilled RP-1"
7        },
8        EngineType M1DV {
9           name "Merlin 1D Vacuum"
10          fuelKind "LOX / RP-1"
11       }
12    }// engineTypes
13
```

```
14    launchSites {
15       operational LaunchSite NKSC {
16          name "Kennedy Space Center"
17          locationLatitude 28.524058
18          locationLongitude -80.65085
19          operator "NASA"
20          numberOfLaunchpads 3
21       }
22    }// launchSites
23
24    spacecraft {
25
26       Spacecraft FH {
27
28          functions {
29             ORBITAL_LAUNCHER
30          }
31
32          name "Falcon Heavy"
33          manufacturer "SpaceY"
34          countryOfOrigin "USA"
35          relaunchCostInMioUSD 90
36          launchSites ( NKSC )
37
38          stages {
39
40             Stage FirstStage {
41                engineAmount 9
42                engineType M1D
43             },
44
45             Stage SecondStage {
46                engineAmount 1
47                engineType M1DV
48             }
49
50          }// stages
51
52          physicalProperties {
53
54             PhysicalProperty {
55                type LENGTH
56                unit m
57                value 70.0
58             },
59             PhysicalProperty {
60                type DIAMETER
61                unit m
62                value 3.66
63             },
64             PhysicalProperty {
65                type WIDTH
66                unit m
```

```
67                 value 12.2
68              },
69              PhysicalProperty {
70                 type MASS
71                 unit kg
72                 value 1420788.0
73              }
74
75           }// FalconHeavy physicalProperties
76
77        }// FalconHeavy spacecraft
78
79     }// spacecrafts
80
81     launchSchedule {
82        LaunchEvent {
83           missionTitle "GPS III-03 navigation satellite deployment"
84           startDateTime "2020-01-31T12:00:00Z"
85           spacecraftId FH
86           launchSiteId NKSC
87        },
88        LaunchEvent {
89           missionTitle "AFSPC-44 payload deployment (classified)"
90           startDateTime "2020-09-30T12:00:00Z"
91           spacecraftId FH
92           launchSiteId NKSC
93        }
94     }
95
96 }// SpaceTransportationService
```

Listing 4.19: Example space transportation service model established by XMLware to grammarware instance-level forward transformation.

4.5.5 Prototype implementations

The approach introduced in this chapter has been prototypically realized using the EMF [197] and Xtext [70]. Additional information, such as slides, source code, and examples, of the XMLText and modeling assistant prototypes are provided at dedicated web pages: https://xmltext.big.tuwien.ac.at and http://angel539.github.io/extremo/.

4.6 Evaluation based on cloud topology and orchestration modeling

This section describes the evaluation of the XMLText framework based on TOSCA [173], a cloud topology and orchestration language standard that has initially been released by the Organization for the Advancement of Structured Information Standards (OASIS) in 2013. TOSCA enables the description of the structure of composite applications,

such as cloud-based web services, as topologies that contain components and relationships, as well as plans that capture component-exposed management tasks, such as the creation or modification of web services. This use case emerged within the ARTIST research project [207] and is motivated by establishing interoperability between the Cloud Application Modeling Language (CAML) [19], i.e. a UML-based language and library of UML stereotypes for the modeling of cloud application deployment specifications developed within the ARTIST project, and the TOSCA language. More specifically, it aims to establish interoperability between CAML and TOSCA to enable the reuse of the TOSCA-based ecosystem [26], which includes the implementation of an execution engine for TOSCA-conforming cloud application deployment descriptions. The TOSCA language is formally defined by the TOSCA XML Schema definition, which in version 1.0 [173] contains 791 lines of code, 99 complex types, 11 simple types, 54 global types, ten wildcards, two abstract types, and two global elements.

The evaluation of XMLText within the standardized TOSCA cloud topology and orchestration specification is twofold. First, the modeling language that is produced for TOSCA by the XMLText framework is compared with the *Cloudify* language, i.e. the only available handcrafted textual modeling language for TOSCA at the point in time when the evaluation was conducted. Second, the ability to transform XML documents to models and sentences that conform to the TOSCA XML Schema definition, the Ecore-based TOSCA metamodel, and the grammar of the TOSCA modeling language is evaluated. Furthermore, the results that are produced by the transformation chain of the opposite direction, i.e. from sentence to model and XML document, are evaluated similarly.

4.6.1 Research questions and evaluation criteria

The common objectives to evaluate the approach and, in particular, the implementation of the XMLText framework include establishing a verdict in regards to the *completeness* and *forward and backward compatibility* of language implementations that are either generated or handcrafted from XML Schema definitions. More specifically, the *completeness* of an individual language implementation, i.e. acting as an executable manifestation of a conceptual language, is defined by its coverage in regards to language concepts that occur in a given XML Schema-based source language definition. Further, *forward and backward compatibility* of an individual language implementation is defined by its ability to employ existing language instances for the generation of respective language instances for XMLware, modelware, and grammarware without the loss of conformance to associated language definitions. Accordingly, the following research questions are formulated:

RQ1: *Is the set of language concepts that are made available by the modeling language that is generated from the TOSCA XML Schema definition at least as comprehensive as the sets of language concepts that are made available by the modeling language that is generated by the Default Transformation Chain, as well as the handcrafted TOSCA language implementation represented by Cloudify?*

RQ2: *Is the execution of sentence-level transformations conformance-preserving with respect to associated language definitions?*

In regards to ***RQ1***, the criteria for comparing the set of language concepts that are made available by the involved languages, which are the result of employing different approaches (i.e. the DTC, XMLText, and handcrafted development), include the language concepts reflected in the specification of the source language, as represented by the XML Schema definition of TOSCA. In other words, a source language specification acts as a container of the complete set of language concepts that are available in a given source language, i.e. represented by the TOSCA standard [173]. Moreover, the capability of an individual language implementation in regards to the instantiation of language concepts that are specified within the Moodle reference example [25], i.e. the definition of topology and orchestration for the Moodle open source course management system, represents the instance-based validation criteria for the *completeness* of an individual language implementation.

The criteria to evaluate ***RQ2*** include the preservation of the conformance of instances, as produced by the execution of round-trip transformations, in regards to their respective language definitions, i.e. represented by the XML Schema definition, the Ecore metamodel, and the Xtext grammar specification that are required to ensure *forward and backward compatibility* of language instances. The units of analysis include source XML instances and target XML instances, i.e. created by the transformation of source XML instances to modeling language sentences that are subsequently serialized to target XML instances. If the source XML instance, as well as the target XML instance, i.e. the result of instantiating a round-trip transformation, conform to the (same) XML Schema definition that has been employed to generate the respective modeling language implementation, then sentence-level transformations preserve the conformance to respective language definitions.

4.6.2 Procedure

In order to compare individual languages in regards to the occurrence of concepts that appear in the TOSCA standard, individual implementations that are based on the TOSCA XML Schema specification are taken into account. In brief, the following steps are performed. First, the DTC is operated with the TOSCA XSD version 1.0 to generate the $TOSCA_{DTC}$ modeling language. Secondly, the XMLText framework is employed by supplying the same TOSCA XSD to produce the $TOSCA_{XMLText}$ modeling language. Third, the language concepts and features that appear in the TOSCA XSD-conforming Moodle reference example are gathered and correlated with language concepts and features that are reflected in the respective language implementations $TOSCA_{DTC}$, *Cloudify*, and $TOSCA_{XMLText}$.

More specifically, in order to construct the set of language concepts that are available in the *Cloudify* language, the respective language parser implementation[4] is analyzed based

[4]The employed *Cloudify* language parser is available at `https://goo.gl/JzPL7U`.

on a comparison with the language concepts that are reflected in the TOSCA standard specification. Furthermore, the unit of analysis in $TOSCA_{DTC}$, i.e. the modeling language that is established by employing the DTC, is represented by the language constructs that are depicted in the grammar of the $TOSCA_{DTC}$ modeling language. Similarly, the unit of analysis in $TOSCA_{XMLText}$, i.e. the modeling language that is established by employing the XMLText framework, is represented by the language constructs that are depicted in the grammar of the $TOSCA_{XMLText}$ modeling language.

The answer to ***RQ1*** is established by comparing TOSCA language concepts that appear in the TOSCA Moodle[5] [62] reference example, i.e. defined by the University of Stuttgart, with the TOSCA language concepts that are available within the respective modeling languages $TOSCA_{DTC}$, $TOSCA_{XMLText}$, and *Cloudify*. The Moodle reference example has been chosen, as it defines the topology details, as well as input and output specifications for an orchestration workflow that is complete, i.e. it may be supplied to the OpenTOSCA [25] container—an open source deployment and orchestration tool for TOSCA—to perform the orchestration of the Moodle application to a cloud provider.

In order to evaluate ***RQ2***, the equivalent Moodle reference example is employed as used in the context of ***RQ1***. More specifically, in order to establish compatibility with existing TOSCA tools, round-trip engineering tests are performed that employ instance-level transformations to transform the (source) Moodle XML document to a sentence, subsequently transform the resulting sentence to a (target) Moodle XML document, and then evaluate the resulting pair of Moodle XML documents in terms of their conformance to the TOSCA XML Schema specification.

4.6.3 Results

In total, the Moodle reference example employs 19 different language concepts that are specified in the TOSCA XML Schema definition. First, operating the *Default Transformation Chain* and the XMLText framework with the TOSCA XML Schema produced a set of two and 19 language concepts respectively (approximately 11% and 100%),which occur in the TOSCA standard and are instantiated by the Moodle reference example. Second, *Cloudify* contains 11 language concepts (approximately 58%) that occur in the TOSCA standard and are instantiated by the Moodle reference example. Therefore, in regards to ***RQ1***, the produced $TOSCA_{XMLText}$ modeling language contains a set of TOSCA language concepts that is approximately 89% and 42% more comprehensive than respective sets of TOSCA language concepts made available by the $TOSCA_{DTC}$ modeling language and the handcrafted *Cloudify* language respectively.

Table 4.3 illustrates the availability of TOSCA language concepts, as instantiated by the Moodle reference example, in the languages $TOSCA_{DTC}$, *Cloudify*, and $TOSCA_{XMLText}$. Therefore, the *Cloudify* language (only) permits the definition of a subset of TOSCA language concepts and, hence, for example, lacks support for the concepts

[5] Moodle refers to an open source course management system.

TOSCA language concept availability			
Moodle	**$TOSCA_{DTC}$**	**Cloudify**	**$TOSCA_{XMLText}$**
TDefinitions	✓	✗	✓
TImport	✓	✓	✓
TServiceTemplate	✗	✗	✓
TTopologyTemplate	✗	✗	✓
TNodeTemplate	✗	✓	✓
TRelationshipTemplate	✗	✓	✓
SourceElementType	✗	✓	✓
TargetElementType	✗	✓	✓
Property	✗	✓	✓
RequirementsType	✗	✗	✓
TRequirement	✗	✗	✓
CapabilitiesType	✗	✗	✓
TCapability	✗	✗	✓
TPlan	✗	✓	✓
TInputParameters	✗	✓	✓
TInputParameter	✗	✓	✓
TOutputParameters	✗	✓	✓
TOutputParameter	✗	✓	✓
TPlanModelReference	✗	✗	✓

Table 4.3: TOSCA concepts instantiated by the Moodle reference example and their availability in $TOSCA_{DTC}$, *Cloudify*, and $TOSCA_{XMLText}$.

RequirementsType, CapabilitiesType, TServiceTemplate, and TTopology-Template. Furthermore, although the *Cloudify* language allows the specification of TPlan, the concepts TPlan and TDefinitions are mixed up. For example, a TPlan in the *Cloudify* language may contain instances of TPolicy, defined as members of TDefinitions and, thus, reflect a concept that is not foreseen by the TOSCA standard. Moreover, the *Cloudify* language implementation does not offer round-trip transformations and, thus, neglects the compatibility of *Cloudify* instances with third-party tools, such as the OpenTOSCA [25] container that consumes TOSCA XML Schema-conforming instances to execute the orchestration of applications to cloud providers.

```
<?xml version="1.0" encoding="ASCII"?>
<tosca:DocumentRoot xmlns:tosca="http://docs.oasis-open.org/tosca/ns/2011/12">
  <definitions id="MyCloudAppDefinition" name="MyMoodleApp Definitions"
      targetNamespace="http://mytargetnamespace.com">
    <serviceTemplate id="MyMoodleAppService" name="My Moodle App Service">
      <topologyTemplate>
        <nodeTemplate id="ApacheWebServer" type="ApacheWebServerType" name="
            Apache Web Server">
          <properties id="ApacheWebServerProperties">
            <numCpus>1</numCpus>
            <memory>1024</memory>
          </properties>
          <requirements>
            <requirement id="ApacheWebServer_container" type="
                SoftwareContainerRequirementType" name="container"/>
          </requirements>
          <capabilities>
            <capability id="ApacheWebServer_webapps" type="
                ApacheWebAppContainerCapabilityType" name="webapps"/>
          </capabilities>
        </nodeTemplate>
        <nodeTemplate id="MyMoodleApp" type="CloudApplicationType" name="My
            Moodle App">
        <requirements>
          <requirement id="MyMoodleApp_container" type="
              ApacheWebApplicationContainerRequirement" name="container"/>
        </requirements>
        </nodeTemplate>
        <relationshipTemplate id="MyMoodleApp_HostedOn_Apache" type="HostedOnType
            " name="hosted on">
          <sourceElement ref="MyMoodleApp_container"/>
          <targetElement ref="ApacheWebServer_webapps"/>
        </relationshipTemplate>
      </topologyTemplate>
    </serviceTemplate>
  </definitions>
</tosca:DocumentRoot>
```

Listing 4.20: XML TOSCA Moodle instance (miniaturized).

```
TDefinitions MyMoodleAppDefinition {
  name: "MyMoodleApp Definitions"
```

```
3  targetNamespace: "http://mytargetnamespace.com"
4  TServiceTemplate MyMoodleAppService {
5    name: "My Moodle App Service"
6    TTopologyTemplate {
7      TNodeTemplate ApacheWebServer {
8        name: "Apache Web Server"
9        type: ApacheWebServerType
10       Property ApacheWebServerProperties {
11          NumCpus: "1"
12          Memory: "1024"
13       }
14       RequirementsType {
15         TRequirement ApacheWebServer_container {
16           name: "container"
17           type: SoftwareContainerRequirementType
18         }
19       }
20       CapabilitiesType {
21         TCapability ApacheWebServer_webapps {
22           name: "webapps"
23           type: ApacheWebAppContainerCapabilityType
24         }
25       }
26     }
27     TNodeTemplate MyMoodleApp {
28       name: "My Moodle App"
29       type: CloudApplicationType
30       RequirementsType {
31         TRequirement MyMoodleApp_container {
32           name: "container"
33           type: ApacheWebAppContainerRequirement
34         }
35       }
36     }
37     TRelationshipTemplate MyMoodleApp_HostedOn_Apache {
38       name: "hosted on"
39       type: HostedOnType
40       SourceElementType ref: MyMoodleApp_container
41       TargetElementType ref: ApacheWebServer_webapps,
42     }
43   }
44 }
45 }
```

Listing 4.21: TOSCA modeling language Moodle model (miniaturized).

Figure 4.12 shows an excerpt of the Ecore-based TOSCA metamodel and Listing 4.10 depicts a miniaturized Moodle topology, as defined by a TOSCA Schema-conforming XML document that specifies an application named `MyMoodleApp`, hosted on an `Apache-WebServer`. Listing 4.19 displays the same Moodle topology definition, serialized in a $TOSCA_{XMLText}$ modeling language grammar-conforming sentence. For example, the XML element that is specified by `xsd:element` and named `serviceTemplate` (cf.

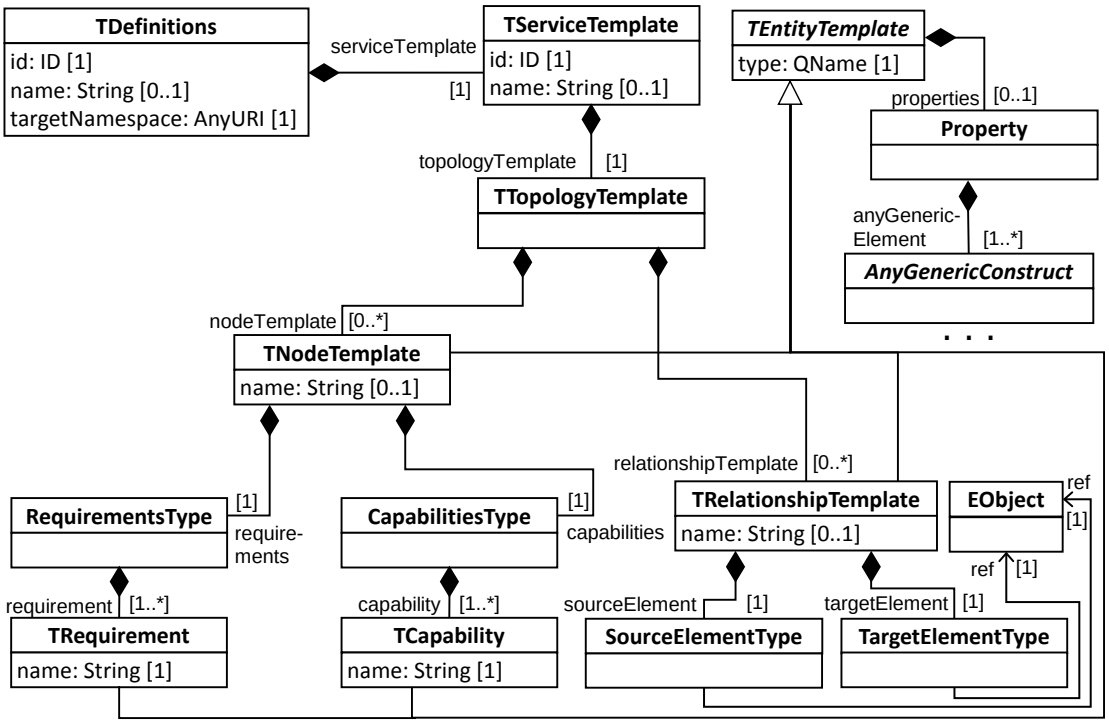

Figure 4.12: TOSCA language metamodel (excerpt).

line 4 in Listing 4.20) is transformed to the `EClass TServiceTemplate` in the Ecore metamodel (cf. Figure 4.12), for which its instance appears in line 4 of Listing 4.21. Furthermore, identifying attributes such as those specified by `requirement` in line 12 of Listing 4.20 are transformed to the identifying attributes – `TRequirement` in line 15 of Listing 4.21 in this case. Identifier references, represented by `xsd:IDREF` and depicted in line 24 of Listing 4.20, are transformed to references to `EObject` in the Ecore metamodel and appear as shown in line 40 of Listing 4.21. Moreover, the XML element named `properties` refers to an `xsd:any` construct defined in the TOSCA XML Schema. Therefore, this XML element is transformed to an `AnyGenericElement` in the Ecore metamodel and appears as `Property` in line 10 of Listing 4.21. The `id` attribute with value `ApacheWebServerProperties`, defined alongside the XML element, is determined as the identifying `AnyGenericAttribute` of its container and, therefore, placed between `Property` and the opening curly bracket.

In summary, the results on ***RQ2*** indicate that the XMLTEXT framework is forward- and backward-compatible, i.e. XML documents that conform to the TOSCA XML Schema definition are transformed to sentences that conform to the grammar of the $TOSCA_{XMLText}$ modeling language and vice versa.

4.6.4 Threats to validity

Internal validity A threat to validity is the fact that the grammar of $TOSCA_{XMLText}$ is manually compared with the language parser implementation of the *Cloudify* language. The rationale behind this manual approach is that several differences between the *Cloudify* language and the TOSCA language specification exist that concern the naming of language concepts. As a result, some concept-correspondences between the two different language implementations may have been missed. In order to mitigate this risk, the *Cloudify* language documentation has been studied alongside the *Cloudify* parser implementation. A second threat to validity is that the grammar of $TOSCA_{XMLText}$ contains a set of TOSCA language concepts that is more comprehensive than those instantiated by the Moodle reference example. Consequently, in order to confirm that TOSCA language concepts, which are not employed by the Moodle reference example, are represented correctly, additional TOSCA-based applications, covering the remaining set of language concepts, must be considered.

External validity Concerning the external validity of the studies, the scope of the presented evaluation may be too narrow, as it is based on a single language and a single example (i.e. the TOSCA standard at the metamodel level and the TOSCA Moodle reference example at the model level). Although the TOSCA standard specification represents a relatively complex language, which imposes a variety of challenges during the exploitation of its XML Schema-based formal language specification for establishing a textual modeling language, although the Moodle example has a working reference application, and although the validity of the presented approach is illustrated based on a real-world case at both the level of metamodel and model, one cannot claim that similar results may be achieved based on use cases that employ different sets of languages and instances. Consequently, additional studies that cover different sets of language constructs, made available by the XML Schema language specification [202], may reveal gaps between XMLware, modelware, and grammar that eventually need to be tackled in order to extend the applicability of the approach.

4.6.5 Summary and discussion

Table 4.4 presents a summary of the TOSCA language concepts and features employed in the Moodle reference example and their availability in the language implementations $TOSCA_{DTC}$, *Cloudify*, and $TOSCA_{XMLText}$. In total, the Moodle reference example uses 19 different language concepts and 35 features defined in the TOSCA XML Schema definition. Thus, the language implementations $TOSCA_{DTC}$, *Cloudify*, and $TOSCA_{XMLText}$ reflect sets of concepts and features that respectively cover a total of 17%, 37%, and 98% of the TOSCA standard concepts and features that are captured by the TOSCA-conforming Moodle reference example.

In summary, the grammar of the $TOSCA_{DTC}$ modeling language lacks essential concepts, such as nodes and relationships, and is therefore not sufficient to represent the Moodle reference example. Furthermore, although the *Cloudify* language parser contains

TOSCA language concept and feature availability				
	Moodle	**DTC**	**Cloudify**	**XMLText**
Concepts	19	2 / ~11%	11 / ~58%	19 / 100%
Features	35	7 / 20%	9 / ~26%	34 / ~97%
Combined	54	9 / ~17%	20 / ~37%	53 / ~98%

Table 4.4: Availability of TOSCA standard concepts and features in different languages based on the Moodle reference example.

a more comprehensive set of language concepts and features than those available in the $TOSCA_{DTC}$ modeling language, it still lacks certain concepts, such as requirements and capabilities. Moreover, for some of these missing concepts, such as `TDefinitions`, features are scattered throughout different concepts in the *Cloudify* language. For example, the parser rule `models.Plan` contains policies and relationships that are originally located in `TDefinitions`. Therefore, the *Cloudify* language does not fully conform to the TOSCA standard and, hence, requires the user to map TOSCA-conforming instances to *Cloudify*-conforming instances manually. Finally, the $TOSCA_{XMLText}$ modeling language allows to represent almost entirely the same information as depicted in the Moodle reference example. In more detail, $TOSCA_{XMLText}$ lacks the representation of the `xmlns` feature, which is represented in the root element of the metamodel. Therefore, except for the occurrence of `xmlns`, the XMLTEXT framework is able to perform round-trip transformations between TOSCA-conforming XML instances and $TOSCA_{XMLText}$-conforming models facilitating the re-use of existing XMLware applications, as well as the advanced capabilities of modern modeling languages. Moreover, three threats to validity have been identified: (*i*) misinterpretation of language concepts and features due to their naming differences in the *Cloudify* language and the TOSCA standard; (*ii*) consideration of a subset of the TOSCA language represented by the TOSCA Moodle reference example, i.e. representing a subset of possible TOSCA language concepts and features; and (*iii*) the consideration of TOSCA as a representative for an XML Schema-based language, i.e. considering a subset of language concepts and features that are made available by the XML Schema language specification. As a countermeasure to (*i*), both the language concepts and features that appear in the *Cloudify* language parser, as well as in the *Cloudify* language documentation have been examined. In order to mitigate (*ii*), several TOSCA-based examples may be employed. Due to the lack of available open source TOSCA-based examples, however, the evaluation is limited to the Moodle reference example. In dealing with (*iii*), TOSCA has been selected due to its relatively complex language structure, which imposes several challenges when transforming the TOSCA XML Schema definition into a textual modeling language implementation. One cannot claim, however, that the established results in regards to the applicability of the approach can be extended to languages that instantiate a different set of concepts that appear in

the XML Schema language specification [202].

*Concept-coverage in generated and handcrafted languages (**RQ1**)*: The $TOSCA_{XMLText}$ modeling language contains a set of TOSCA language concepts that is approximately 89% and 42% more comprehensive than respective sets of TOSCA language concepts made available by the $TOSCA_{DTC}$ modeling language and the handcrafted *Cloudify* language respectively.

*Conformance-preservation in sentence-level transformations (**RQ2**)*: The produced results indicate that the XMLText framework is forward- and backward-compatible, i.e. XML documents that conform to the TOSCA XML Schema definition are transformed to sentences that conform to the grammar of the $TOSCA_{XMLText}$ modeling language and vice versa.

4.7 Evaluation based on industrial conveyor-belt system modeling

This section presents the evaluation of the XMLText framework integration in the modeling assistant within the context of constructing an industry standard-conforming conveyor-belt production system modeling language. The aim is to answer the following research questions.

4.7.1 Research questions and evaluation criteria

The overarching objective to evaluate the approach includes reaching a verdict on the usefulness of the XMLText assistant integration and, in particular, its applicability in the context of language engineering.

RQ1: *How useful is the* XMLText *assistant integration in solving practical problems in language engineering?*

RQ2: *How effective is the common data scheme in capturing heterogeneous information mapped by the assistant integrations for XMLware and modelware?*

RQ3: *How much effort is involved in the integration of* XMLText *in third-party tools?*

In order to assess the *usefulness* of the XMLText assistant integration in solving practical problems in language engineering, a use case is presented that employs the XMLText assistant integration for the development of an industry standard-conforming modeling language implementation, as well as its application for the modeling of a conveyor-belt production system. The industry standard being employed in this case is represented by the eCl@ss standard [63], an ISO/IEC-compliant international standard for the unified and consistent definition and classification of products, materials, and services alongside typical supply chains through the use of commodity codes.

In order to determine the *effectiveness* of the common data scheme (cf. Section 4.5.3) in representing information that originates from the technological spaces of XMLware and modelware, an analytical evaluation is conducted that computes the ability to capture heterogeneous information provided by the XMLTEXT assistant integration from XML Schema descriptions and XML instances, as well as the Ecore assistant from Ecore metamodels.

One of the salient features of the XMLTEXT framework is the ability to be *integrated* with third-party tools. Therefore, the proficiency of XMLTEXT to be integrated by modeling tools is evaluated by assessing the reach and limitations of the XMLTEXT framework integration in the modeling assistant.

4.7.2 Procedure

The procedure for the evaluation includes the following sequence of phases: outlining the scope of the language, the process of collecting required resources, the import of information into the common data scheme, the querying of information from the common data scheme, and the construction of the target language metamodel and model. More specifically, a modeling language for industrial production systems will be developed that targets conveyor-belt system-modelers and, thus, aims to reduce the number of components available in the eCl@ss standard to components for conveyor-belt systems.

Language scope. The scope of the development of the eCl@ss Process Modeling Language (EPML) modeling language is to enable the construction of production system models that conform to the eCl@ss standard. Moreover, the language includes elements from conveyor-belt systems that must fulfill the constraints imposed by the eCl@ss standard and the Signal Process modeling Language (SigPML)—a modeling language dedicated to data flow processing presented as part of the GEMOC initiative [46].

Example model. Figure 4.13 and Figure 4.14 respectively illustrate the graphical and tree-based representation of an example eCl@ss standard-conforming conveyor-belt production system model adopted from the AutomationML and eCl@ss integration white paper[6]. More specifically, the model contains electrical drives (i.e. instances of *DC Engine*), communication cables or ready-made data cables that represent SigPML connectors, PC-based controls or field buses (i.e. decentralized peripherals, which represent controls), controls in terms of inductive proximity switches, and sets of rectangular industrial connectors, which represent connector systems. A *DC Engine* is connected to a *Fieldbus* through a communication cable and a data cable with a set of industrial connectors. The *Fieldbus* also has a connection with a *Proximity Switch* through a communication cable and a data cable that use the industrial connectors that are supported by the *Proximity Switch*.

[6]The AutomationML and eCl@ss integration white paper is available online at https://www.automationml.org/o.red/uploads/dateien/1417686950-AutomationML%20Whitepaper%20Part%201%20-%20AutomationML%20Architecture%20v2_Oct2014.pdf.

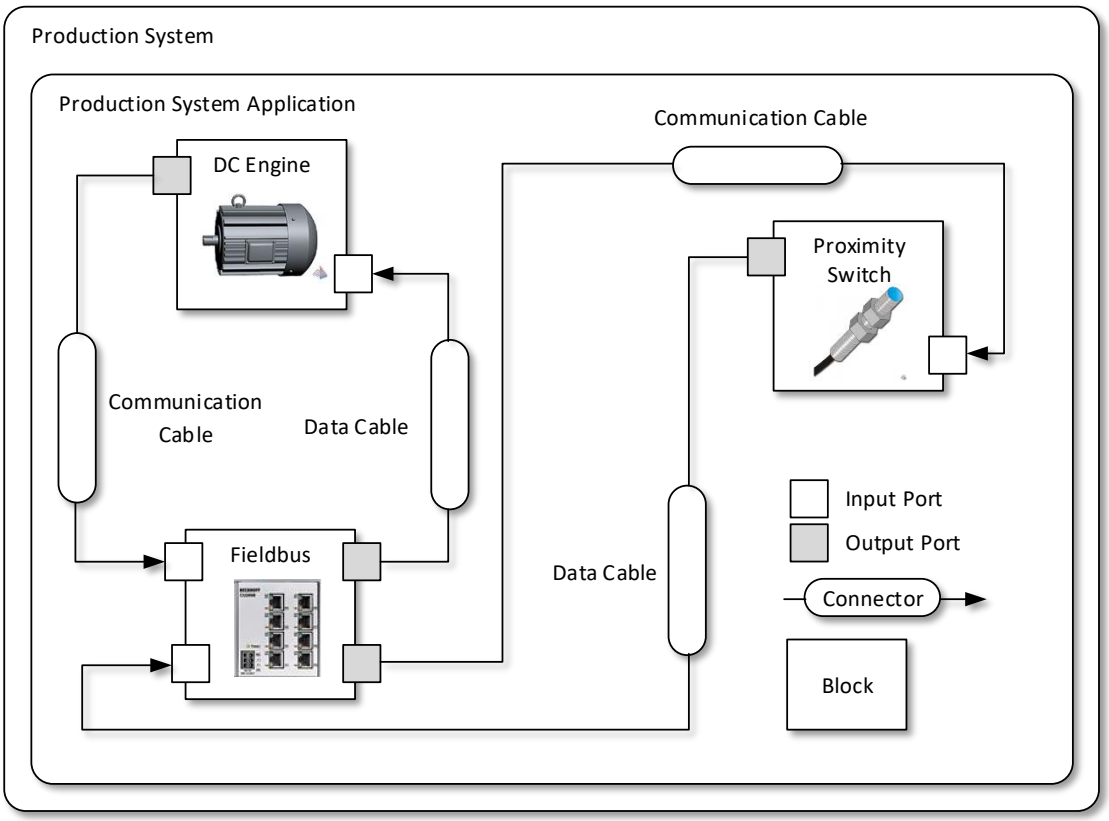

Figure 4.13: Graphical representation of an example eCl@ss standard-conforming conveyor-belt production system model.

Resource collection. In order to gather all the required elements, a set of repositories and resources will be defined according to the common data scheme shown in Figure 4.6. The resources are gathered from the GEMOC initiative, such as the SigPML Ecore metamodel[7] and the eCl@ss standard [63], i.e. defined in the form of several XML Schema definitions and XML instances.[8] Table 4.5 summarizes the number of instances of the eCl@ss standard, as well as those of SigPML that are available on a meta-level. It is important to note that the domain-specific concepts that are represented in the eCl@ss standard are substantial in size. The basic and advanced specifications in the English language alone consist of 41,000 product classes and 17,000 properties or approximately 15.5 gigabytes of data. Consequently, the extraction of desired concepts for conveyor-belt system modeling requires the manual examination of a vast amount of resources, as well as the eventual re-implementation of a target modeling language. Moreover, any update

[7]The SigPML metamodel is available online at `http://bit.ly/32NqLgp`.

[8]The resources of the eCl@ss standard are proprietary and have been made available by the eCl@ss e.V. on request. A summary of available eCl@ss language resources may be found online at `https://www.eclassdownload.com/catalog/eclass_releases.php`.

Figure 4.14: Tree-based representation of an example eCl@ss standard-conforming conveyor-belt production system model.

that is performed on XML Schema definitions and XML instances that represent the eCl@ss standard may impact implementation in the target system and, thus, may involve complex and time-consuming maintenance. In an effort to counteract these limitations, the XMLText assistant integration will be employed.

Resource import. First, resources will be collected by employing the import functionality offered by the *EcoreAssistant* and the *XsdAssistant*, building on the XMLText framework [165]. More specifically, the XMLText framework transforms XMLware artifacts, i.e. XML Schema definitions and XML instances, to corresponding modelware-artifacts, i.e. Ecore metamodels and conforming models. Next, the *EcoreAssistant* will be employed to map modelware-artifacts to the common data scheme in order to enable their use by the modeling assistant.

Metamodel construction. Next, the Eclipse perspective of the assistant will be integrated with the Sample Reflective Ecore Model Editor, as described by the architecture of the assistant in Section 4.5.3.

XML Schema concept in eCl@ss standard	Number of instances
XML Schema file	30
Element	960
Element with IDREF attribute	110
Attribute	104
Restriction	84

Ecore concept in SigPML	Number of instances
Ecore file/EPackage	1
EClass	14
EReference	18
EAttribute	10
OCL EAnnotation	0

Table 4.5: Number of individual XML Schema definitions and Ecore concepts collected from input resources.

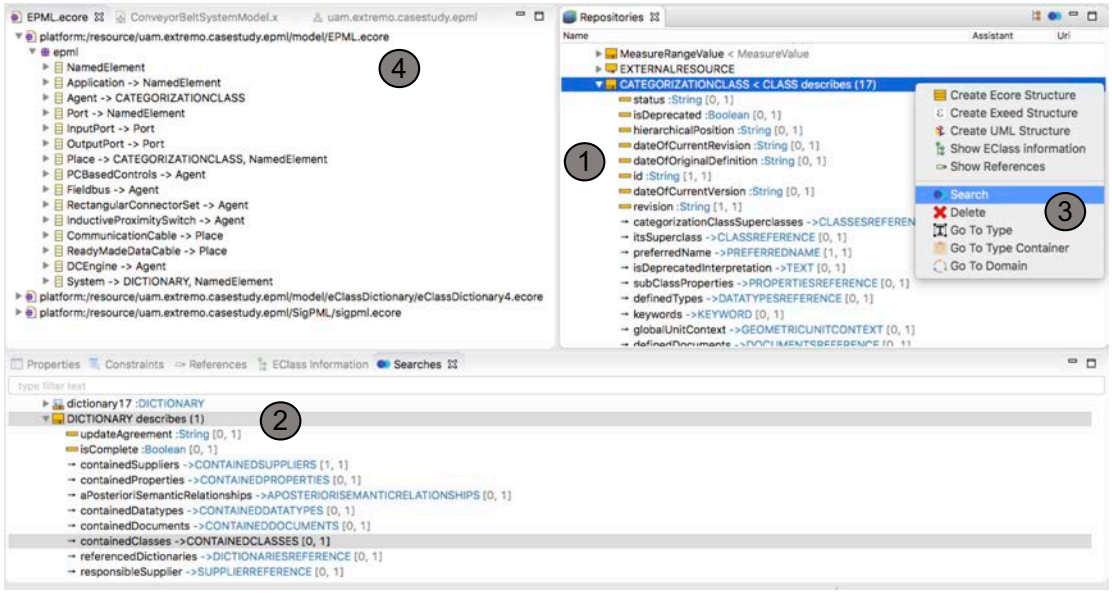

Figure 4.15: XMLText assistant integration with the Sample Reflective Ecore Model IDE during EPML metamodel construction.

4.7.3 Results

Figure 4.15 is a screenshot from the construction of the EPML metamodel. More specifically, (1) visualizes a set of resources in the repository view; (2) depicts the result of querying the Ecore metamodel that has been constructed from the eCl@ss standard by the XMLText assistant integration; (3) shows a context menu with the selection of the functionality that triggers the traversion of class *CATEGORIZATIONCLASS*; and (4) illustrates the application of desired concepts from the imported repositories in the

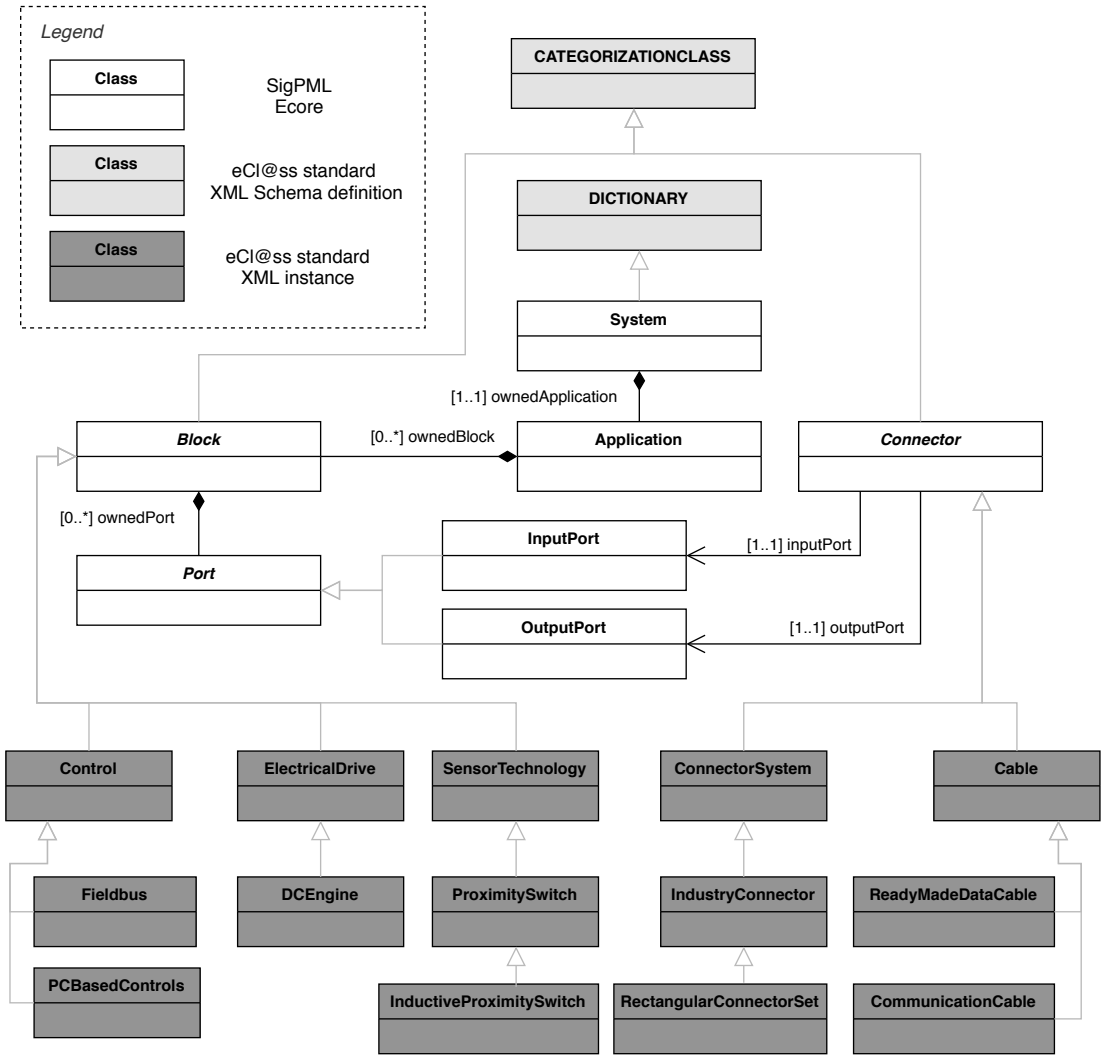

Figure 4.16: Excerpt of EPML metamodel based on SigPML and the eCl@ss standard.

target language. For example, available concepts that represent an electrical drive in the eCl@ss standard have been gathered by issuing a query for retrieving semantic nodes that are named *engine*, which then facilitated the creation of corresponding concepts in the metamodel of EPML. Further, several traversal functionalities, such as *Reveal On Repository*, *Go To Type*, and *Go To Domain*, have been employed to gather respective classes that are referenced as super-types and, thus, force EPML models to conform to the eCl@ss standard.

The final outcome of the EPML metamodel construction process is shown in Figure 4.16. Rectangles colored in dark gray represent data flow process elements that originate in the SigPML and include *System*, *Application*, *Block*, *Connector*, and *Port*. Rectangles colored in light gray indicate eCl@ss standard concepts, such as electrical

drives, cables, controls, binary sensors,[9] and connector systems. For example, the eCl@ss standard class named *CATEGORIZATIONCLASS* represents the supertypes *Block* and *Connector* in EPML. Additionally, subtypes of *Block* and *Connector*, such as *Control* and *Cable*, are also instances of *CATEGORIZATIONCLASS* in the eCl@ss standard. The distinction of individual instances of categorization classes introduces semantics in EPML that enable the representation of concepts originating in SigPML and the eCl@ss standard.

Table 4.6 summarizes the XMLText assistant integration transforming eCl@ss standard resources from XMLware to modelware and, in particular, XML files to XMI files to enable their handling by the modeling assistant. The meta-level contains *EPML.ecore* as *Resource*, and references 18 individual XML Schema definitions that are supplied by the eCl@ss standard and instantiated by the XML instance *eClass9_1_BASIC_EN_SG_27.xml* (55.6 MB). More specifically, the meta-level is represented by 525 types of semantic nodes, 500 types of object properties, 26 types of data properties, and 88 types of constraints. Moreover, the model-level instantiates 487,746 semantic nodes, 805,097 object properties, 487,745 data properties, and 820,356 constraints.[10] Note that SigPML contains thirteen types of semantic nodes that are not instantiated and are, therefore, neglected in Table 4.6.

	Number of instances	
common data scheme concept	**Meta-level (types)**	**Model-level (instances)**
Resource	1	1
SemanticNode	525	487,746 (4,071)
ObjectProperty	500	805,097 (23,752)
DataProperty	26	487,745
Constraint	88	820,356

Table 4.6: Instances of conveyor belt system components imported from the eCl@ss standard to the common data scheme.

As a result, the modeling language constructed by employing the assistant reduces the number of potential candidates for the conveyor-belt system components in the eCl@ss standard by approximately 99.17% (97.05%), i.e. from 487,746 (805,097) to 4,071 (23,752) semantic nodes (object properties) that represent instances (references) of *CATEGORIZATIONCLASS* and, thus, potential candidates for instances (references) of (to) *Block* and *Connector* in SigPML.

[9] Binary sensors represent safety-related sensors in the eCl@ss standard.

[10] Note that the *number of instances* of constraints refers to the number of constraints defined at meta-level and validated at model-level.

Finally, the XMLTEXT framework is supplied with the file *EPML.ecore*—the metamodel of the industry standard-conforming conveyor-belt system modeling language that has been established by facilitating the modeling assistant with the XMLTEXT assistant integration—for the generation of EPMLText, i.e. an Xtext-based modeling language that enables the construction of eCl@ss standard-conforming models, such as those depicted by the concrete textual representation of the example EPML model in Listing 4.22.

```
System ConveyorBeltProductionSystem {
  containedSuppliers CONTAINEDSUPPLIERS {
    suppliers {
      SUPPLIER {
        org ORGANIZATION "eCl@ss" {
          id "0173-1"
        }
      }
    }
  }
  ownedApplication Application ConveyorBeltProductionSystemApplication {
    ownedAgents {
      DCEngine {
        dateOfCurrentRevision "2004-09-27Z"
        ownedPorts {
          InputPort DC_Engine_Data,
          OutputPort DC_Engine_Comm
        }
      },
      InductiveProximitySwitch {
        dateOfCurrentRevision "2015-11-13Z"
        ownedPorts {
          InputPort Inductive_Proximity_Switch_Comm
          OutputPort Inductive_Proximity_Switch_Data
        }
      },
      Fieldbus {
        dateOfCurrentRevision "2013-11-28Z"
        ownedPorts {
          InputPort Fieldbus_Comm_IN, Fieldbus_Data_IN
          OutputPort Fieldbus_Comm_OUT, Fieldbus_Data_OUT
        }
      }
    }
    ownedPlaces {
      CommunicationCable Fieldbus_DCEngine_Communication_Cable {
        dateOfCurrentRevision "2014-11-30Z"
        itsInputPort DC_Engine_Comm
        itsOutputPort Fieldbus_Comm_IN
      },
      ReadyMadeDataCable Fieldbus_DCEngine_Data_Cable {
        dateOfCurrentRevision "2014-11-30Z"
        itsInputPort Fieldbus_Data_OUT
        itsOutputPort DC_Engine_Data
      },
      CommunicationCable Fieldbus_ProximitySwitch_Communication_Cable {
```

```
47        dateOfCurrentRevision "2014-11-30Z"
48        itsInputPort Fieldbus_Comm_OUT
49        itsOutputPort Inductive_Proximity_Switch_Comm
50      },
51      ReadyMadeDataCable Fieldbus_ProximitySwitch_Data_Cable {
52        dateOfCurrentRevision "2014-11-30Z"
53        itsInputPort Inductive_Proximity_Switch_Data
54        itsOutputPort Fieldbus_Data_IN
55      }
56    }
57  }
58 }
```

Listing 4.22: Concrete textual representation of an example eCl@ss standard-conforming conveyor-belt production system model (simplified).

Moreover, EPML may be extended by either adding further eCl@ss standard specific concepts, which represent instances of *CATEGORIZATIONCLASS*, to the metamodel or by expressing the concept of blocks and connectors as concrete (instead of abstract) classes. Consequently, the latter option moves the decision-making process for choosing the desired eCl@ss standard elements from the metamodel level to the model level. Although the assistant supports such cases by means of level-agnostic data handling, the chosen option constrains EPML at metamodel level to limit the set of possible types that may be instantiated at model-level in order to fit the purpose of modeling conveyor-belt production systems. The XMLText assistant integration enables importing XML Schema definitions that offer enumerations and extensible data types, as well as cardinalities, on features with different configurations for the semantics of *many*. Furthermore, XML Schema definitions offer the following three possibilities for element types: single inheritance, features, and nesting of element types (i.e. hierarchies). XML Schema definitions only allow binary unidirectional references and strict typing, with the exception of open points that allow for any valid XML structure. XML Schema follows the classic two-level approach and allows for local constraints. Without the reuse of the (existing) mapping implementation of the *EcoreAssistant* (cf. Section 4.5.3), however, the mapping of the resulting modelware artifact to the common data scheme requires a total of approximately 500 lines of code. The full version of the *XsdAssistant* is available in the Github repository of the modeling assistant[11].

4.7.4 Threats to validity

As described in the three preceding subsections, the assistant has been used to support the construction of a modeling language by reusing heterogeneous artifacts and, in particular, an Ecore metamodel, a set of XML Schema definitions, and an XML instance. More specifically, the degree to which the common data scheme of the assistant is able to

[11] The source code of the XMLText assistant integration Java class is available online at https://github.com/angel539/extremo/blob/master/uam.extremo.extensions.assistants/src/uam/extremo/extensions/assistants/XsdAssistant.java.

accommodate information from heterogeneous sources and, in particular, XML-based resources has been analyzed. Further, the proficiency of XMLText to be integrated has been evaluated by the implementation of the *XsdAssistant*, namely by presenting the integration of the XMLText framework with the modeling assistant. Although the results are positive, a set of potential threats to the validity of the experiments were identified and categorized according to the four basic types of validity threats [220].

Threats to construct validity. Construct validity is concerned with the relationship between theory and observation. The presented case focuses on the evaluation of the use of the XMLText assistant integration for XML Schema definitions and XML instances, as employed by the eCl@ss e.V. and other third parties. Although the eCl@ss specification represents an industrial standard and, thus, reflects realistic requirements, the constructed modeling language for conveyor-belt systems has been constructed by developers with limited knowledge on industrial production systems. Therefore, further studies need to be performed in order to evaluate the capability of the modeling assistant to support domain experts during the construction of modeling languages that fulfill requirements that may supersede those covered by the presented use case.

Threats to conclusion validity. Conclusion validity is concerned with the relationship between treatment and outcome. The considered use case is composed of a set of 18 individual XML Schema definitions, an XML document, and an Ecore metamodel as input, and an Ecore metamodel and Xtext grammar as output. As such, it covers the three technical spaces of XMLware, modelware, and grammarware. Although these numbers are limited, they provide an indication of the usefulness, applicability, and proficiency of XMLText, and its integration in the modeling assistant.

Threats to internal validity. Internal validity establishes a causal relationship between the cause and effect of an event. Although the development of the XMLText assistant integration was performed by the subject that developed the XMLText framework, this was not involved in the development of the modeling assistant. Although the integration of the XMLText in the assistant requires a small number of lines of code, it may be more demanding for engineers that have not been involved in the development of XMLText, the assistant or any third-party tool.

Threats to external validity. External validity refers to the domain to which the findings of the evaluation can be generalized. On the one hand, the presented findings illustrate the ability of the XMLText framework to be integrated with a modeling assistant in order to facilitate the exploitation of XMLware artifacts and support the construction of an industry standard-conforming modeling language for conveyor-belt systems. On the other hand, the presented findings may be limited, and challenge the ability of the XMLText framework to be integrated with third-party tools, such as the metamodel repository tool MDEForge [183], and language engineering use cases that are different to the one presented.

4.7.5 Summary

*Usefulness of assistant integration (**RQ1**)*: The XMLText assistant integration is useful in solving practical problems in language engineering as shown by its capability to exploit XMLware resources and, in particular, by importing and querying information to support the construction of a conveyor-belt system modeling language, which conforms to the industrial eCl@ss standard specification [63].

*Effectiveness in capturing heterogeneous information (**RQ2**)*: The effectiveness of the common data scheme in capturing heterogeneous information has been shown by the assistant integrations for XMLware and modelware and, in particular, by the import of both the SigPML metamodel and the formal XML-based specifications of the eCl@ss standard.

*Third-party tool integration (**RQ3**)*: The amount of effort involved in the integration of the XMLText framework with third-party tools has been illustrated by the implementation of the *XsdAssistant* requiring five lines of code to import and instantiate the XMLware to modelware transformer, supply an XML Schema definition and XML instance, and retrieve the resulting modelware artifact.

4.8 Analysis

This section describes the differences and similarities of the presented approach with the bridging of the technical spaces of XMLware, grammarware, and modelware available in the literature and, in particular, work outlined in Section 3.1.

On a general level, the presented approach applies the ModelGen operator of Atzeni et al. [12]. This operator defines a general pattern, which uses bridges on the meta-language level to derive transformations on the language and instance levels. This pattern also fits the architecture presented in Figure 4.3. Traditionally, this pattern is proposed and used in the database field for schema-independent transformations. It is also applicable, however, in the field of language engineering.

With respect to the transformation chain presented in this chapter, a set of related approaches exists, which covers certain aspects of this chain by focusing on the transitions between the technical spaces involved. In what follows, bridges between the technical spaces of XMLware, grammarware, and modelware (cf. Section 3.1) are compared with the presented approach.

Contrary to the presented approach, users of modelware and grammarware approaches typically handcraft transformation rules between either individual grammar rules or terminal rules and metamodel elements, as opposed to relying on a generic and automated transformation of XML Schema definitions.

Several approaches for realizing either forward engineering from modelware to XMLware [27, 47] or reverse engineering from XMLware to modelware [189, 147] exist (cf.

Section 3.1.1). Previous work presents an approach for the generation of MOF-based metamodels from DTDs [189]. More specifically, changing the applied source type, i.e. DTDs as opposed to XML Schema definitions, as well as the transformation chain, applies to modelware as opposed to grammarware. Neglecting the transformation of modelware to grammarware makes it difficult for language engineers to prototype textual modeling language implementations from XML Schema definitions quickly.

In regards to XMLware and grammarware, Eysholdt and Rupprecht present a report on the migration of a modeling environment from XML/UML to Xtext/GMF [71] (cf. paragraph "From XML and UML to Xtext and GMF" in Section 3.1.1). Their goal is to modernize legacy modeling environments to overcome inefficiencies of XML caused by verbose syntax and lack of tool support. In summary, their approach creates Ecore metamodels from XML Schema definitions and then performs changes and customizations of Xtext features to complete the implementation of the desired modeling language and workbench. The completeness and correctness of textual model serialization and XML document persistence is similarly evaluated by employing round-trip testing. Although partial automation is reported through the use of facilities provided by the EMF, manual customizations to imported XML Schema constructs have been made to address "glitches in XML files" not further elaborated in their work. Thus, changes to the XML Schema definitions represented by Ecore metamodels, as well as customizations of Xtext features, are performed manually rather than through a generic and automated transformation chain from XMLware to grammarware. For example, manual restructuring has been employed where reasonable, such as in the merging of separate listings for table columns and class attributes, into class attributes containing optional column specifications in the presented language for the persistence of objects.

Chodarev [43] presents an approach for the development of a translator between XML and a DSL with customized concrete syntax. This approach differs from the one presented primarily in the following. First, abstract syntaxes are represented by Java classes that are generated by the JAXB XML binding compiler [169] and annotated with Java annotations, as opposed to metamodels that are generated by the EMF and refactored with model-to-model transformations. Second, parsers are generated by the use of the YAJCo parser generator, as opposed to Xtext and the underlying ANTLR tool. Third, although the expressiveness of notation specifications in Chodarev's approach is higher than those offered by XMLText, i.e. without considering the option that involves the (direct) adaptation of grammar to fit a particular notation, the example in his work only employs annotations that are already offered by the out-of-the-box YAJCo implementation, such as *Before*, *After*, *Indent*, *Token*, and *NewLine* (cf. Figure 5 in [43] and respective Java classes of YAJCo implementation [179]). Moreover, the contribution presented in Chapter 6 addresses this shortcoming of XMLText through the implementation of a dedicated framework for the specification of textual notations in the form of style models that are employed alongside structural language specifications, such as Ecore metamodels, in order to generate grammars that follow the notation outlined by the respective style model. Fourth, both approaches evaluate the validity

and completeness of the round-trip transformation implementation by comparing original instances and instances produced by the execution of round-trip transformations.

Izquierdo et al. present Gra2MoL [111], in which users define transformations based on language grammar. In comparison, Gra2MoL also employs an automated enrichment process, referred to as "preprocessing step to render ANTLR grammar compatible with Gra2MoL" and "metamodel adaptation" in Gra2MoL and the approach presented in this chapter (cf. Section 4.5) respectively. The source language type in Gra2MoL, however, is represented by ANTLR grammars, as opposed to XML Schema definitions. Therefore, at instance-level, Gra2MoL focuses on the evolution of ANTLR-grammar source code, as opposed to XML Schema-conforming XML documents. Moreover, the approach presented in this chapter does not require the specification of individual transformations for the migration of XML Schema-based languages to modelware and grammarware.

Compared to the approach presented by Wimmer et al. [218], automated metamodel refinement is employed rather than a combination of manual metamodel annotations and model transformations. Further, XML-based languages are transformed to grammarware, as opposed to the transformation of EBNF-based languages to MOF-based languages, i.e. grammarware to modelware.

Similarly to the approach presented by Kunert et al. [138], the work presented in this chapter also employs a fully automated migration. As opposed to automating the migration of XML-based languages, however, their approach migrates EBNF-based languages and eliminates syntactic information that is depicted in the generated metamodel. As a result, their approach neglects backward compatibility and, in particular, the ability to parse and serialize XML documents.

An approach for the translation of MontiCore [135] grammars, i.e. EBNF-like grammars, to restricted Ecore metamodels is presented by Butting et al. [39]. Their work represents a unidirectional transformation approach and is, thus, limited to the translation of grammars to constrained metamodels. In other words, the translation of constrained metamodels to grammars is not investigated and, thus, modifications that are performed on such metamodels are not translated to the corresponding grammar modifications.

4.9 Summary

This chapter presented the first contribution of this dissertation, namely the automated exploitation of XML Schema-based language specifications for the generation of textual modeling language implementations (cf. *C1* in Figure 1.1). More specifically, individual gaps between the technical spaces of XMLware, modelware, and grammarware have been highlighted, and include individual XML Schema concepts such as mixed content, wildcards, restrictions, identifiers and identifier references, data types, and rigid concrete syntax. Next, an approach was developed, which fills identified gaps in the literature through the construction of a transformation chain from XMLware to modelware and

modelware to grammarware by facilitating metamodel adaptation for the automated generation of textual modeling language implementations from XML Schema definitions. Moreover, the approach was integrated into a modeling assistant for the exploitation of XMLware artifacts and, in particular, the import and querying of domain-specific knowledge that is embodied by XML Schema definitions and XML documents, as well as their application for the construction of (meta)models.

The implementation of the approach was evaluated in a case study and a use case. The former involves the migration of a standardized cloud modeling language and a respective reference model depicting the topology and orchestration of an open source course management system. The latter involves the application of the modeling assistant integration to the construction of an industry standard-conforming conveyor-belt production system. More specifically, the assistant integration was evaluated in terms of its usefulness for solving practical problems in language engineering, the effectiveness of the employed common data scheme in capturing heterogeneous information and, in particular, sources that are mapped from the technical space of XMLware, and the ability of the XMLText framework to be integrated with the modeling assistant. In summary, both evaluations indicate the ability of the presented approach to exploit XML Schema-based language specifications for the automated migration of backward-compatible XMLware to modelware and grammarware, as well as the assisted construction of modeling languages in the form of metamodels, and their subsequent application by the developed framework for the automated generation of grammars and textual modeling language implementations. Its usability for integrated modeling assistance and the automated migration of backward-compatible modeling languages from XML Schema-based language specifications has thus been demonstrated.

CHAPTER 5

Consistency-achieving integrated development environment

The previous chapter presented an approach to exploit XML Schema-based language specifications for modeling assistance and automated generation of textual modeling language implementations by adapting metamodels from XML Schema definitions. This chapter focuses on the automated generation of consistency-achieving IDEs through the facilitation of formal constraints that are specified alongside the metamodel of a domain-specific modeling language (i.e. henceforth simply referred to as *modeling language*). The advances of modeling languages and dedicated IDEs, i.e. created with modern language workbenches, are recognized by domain experts as important and powerful means to capture domain-specific information. Despite the fact that such IDEs are proficient in retaining syntactical model correctness, they present major drawbacks in preserving consistency in models which require language engineers to implement manually IDEs facilitating language definitions with elaborated language-specific constraints. Consequently, there is a demand for automating procedures to support language designers in the construction of enhanced modeling language implementations and, ultimately, to assist IDE users in both model construction, as well as model consistency violations resolution. This chapter presents an approach to automate the creation of IntellEdit-generated IDEs which offer automated validation, content-assistance, and quick fix resolution capabilities that are beyond those created by state-of-the-art language workbenches and are therefore capable to support domain experts in retaining and achieving the state of model consistency. For validation, the causes of potential errors for violated constraints are visualized, instead of only the context in which constraints are violated. The state-space explosion problem is mitigated by the approach resolving constraint violations by increasing the neighborhood scope in a three-stage process, seeking constraint repair solutions presented as quick fix solutions to IDE users. The approach is evaluated based on a modeling language for modeling service clusters. Figure 5.1 recaptures the contributions

of this thesis and highlights the contribution presented in this chapter.

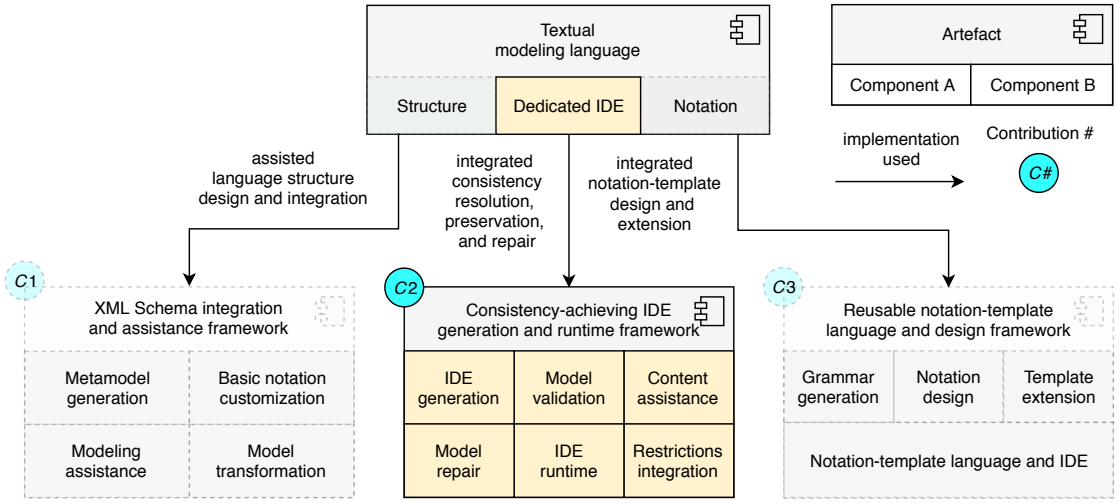

Figure 5.1: Contribution presented in this chapter.

The remainder of this chapter is organized as follows. Section 5.1 introduces and the approach and its motivations. Section 5.2 briefly describes the integration of the formal constraint language OCL in Ecore and Xtext. Section 5.3 and Section 5.4 present the challenges and requirements of the approach presented in this chapter respectively. Section 5.5 presents the approach and in particular the generation of enhanced IDEs, as well as the facilitation of XML Schema restrictions in order to limit the set of valid model instances. Section 5.6 demonstrates an evaluation based on a language for modeling service clusters. Section 5.7 analyzes and compares the proposed approach with related work and, in particular, in a comparison with state-of-the-art language workbench frameworks. Finally, Section 5.8 concludes the chapter by summarizing the presented work.

5.1 Introduction

Domain-Specific Modeling Languages (DSMLs) [119] encode the expertise of domain experts [90], e.g. hospital process managers and mechatronics engineers, and are, hence, languages designed for a specific class of problems allowing domain experts to describe their problem on a higher level of abstraction than what is possible with General-Purpose Modeling Languages (GPMLs) [87]. DSMLs are known to leverage domain expertise and improve usability, comprehensibility, and maintainability compared to alternatives. On the one hand, the benefits of modeling languages have long been recognized, but on the other hand the development of advanced modeling language implementations still requires extensive manual effort and the combined knowledge of domain and language engineers, resulting in slow industrial adoption. Therefore, in order to advance the latest developments in the maintenance and evolution of languages and systems created with

them, the goal is to reduce significantly the cost associated with the creation and, thus, the adoption of advanced modeling language implementations required by the currently available approaches [84].

Model-Driven Language Engineering (MDLE) frameworks (i.e. also referred to as *language workbenches*), such as Xtext [70], ease the out-of-the-box creation of powerful modeling language implementations, including associated IDEs [84]. Typically, an MDLE framework allows the creation of a language either by developing a metamodel representing the language's abstract syntax, which is subsequently used to generate a modeling language implementation including language grammar, or by developing a language grammar, which is automatically used to generate an associated metamodel. Thus, MDLE frameworks provide the necessary means for automating the creation of modeling language implementations with advanced IDE capabilities. The generated modeling language implementations, however, require extensive manual implementation for enabling precise validation, content-assistance, and quick fix capabilities. For example, a metamodel containing formally specified consistency constraints, which restrict the amount of stages of orbital launch vehicles, dictates that these must be less than or equal to two in order to meet safety regulations. In this example, models that define vehicles with a number of stages that is larger than two are considered invalid. In order to take formal constraint specifications into account, however, the language developer has to manually implement validation, content-assistance, and quick fix resolution using a general-purpose programming language, such as Java, on top of the generated modeling language IDE. Although enhanced IDEs, which offer precise validation, consistency-maintaining content assist, and consistency-restoring model repair, facilitate the creation and modification of consistency-achieving models, therefore increasing productivity in language use, the effort and expertise that is required for their initial development and ongoing maintenance often challenges their introduction in the first place. Moreover, models that are created with modeling languages as well as IDE implementations themselves, evolve [74, 153]. Each time a metamodel or its associated formal constraint specifications are adapted, handcrafted implementations of model validation, content assist, and model repair typically require refactoring as well.

This chapter presents an approach to generate automatically precise model validation, content assist, and model repair capabilities for modeling languages based on formal constraints that are defined alongside language metamodels. Hence, the approach facilitates language definitions composed of metamodels and formal constraint specifications for the automated manifestation of modeling language implementations that include the aforementioned capabilities and is, thus, proficient in the reduction of the cost associated with establishing and maintaining IDE implementations, as well as the growth in efficiency associated with the creation and preservation of consistency-achieving models. Formal constraint specifications may be added to modeling languages in terms of Object Constraint Language (OCL) [40] constraints and Ecore annotations added to metamodel elements. This approach extracts OCL invariants and Ecore annotations from metamodels and analyzes them for the manifestation of enhanced modeling language

IDEs. Consequently, the following challenges must be resolved. First, the scope of infinite possible paths to repair a given set of inconsistencies must be reduced to a level that is practically applicable, e.g. as shown by Nentwich et al. in [160]. Second, the impact of model repair solutions, i.e. represented by sequences of change actions, must be measured to enable ranking according to cost [151]. The hypothesis is that the application of the approach presented in this chapter increases both the effectiveness of the manifestation of modeling language implementations, as well as the creation and editing of consistency-achieving models. The approach is evaluated based on a modeling language for service clusters and, in particular, by comparing the capabilities of the dedicated IDE generated by Xtext and IntellEdit, i.e. an implementation of the approach, respectively. The goal of the approach is to enable MDLE frameworks to be extended with the capability to automate fully the generation of backward-compatible modeling languages with enhanced IDE implementations from existing language specifications, such as XML Schema-based languages [161]. Hence, the approach [164, 165] presented in this chapter focuses on the particular aspect of increasing the effectiveness, i.e. in terms of quality and usability, of modeling language IDEs by facilitating metamodels with formal constraint specifications and apply them to reject statically semantically invalid models, as well as offering precise model validation, consistency-preserving content assistance, and consistency-restoring model repair solutions. More specifically, the latter are generated at runtime by facilitating Search-based Software Engineering (SBSE) techniques that entail the application of search-based optimization, i.e. originally introduced in metaheuristic computation and operations research. The implementation and evaluation of the approach has been made available in the form of the Intelligent Editing (IntellEdit) framework. A ready-to-use virtual machine image and Eclipse instance of the framework. Evaluation results may be retrieved from `https://intelledit.big.tuwien.ac.at`.

5.2 Background

In what follows, a brief overview of the integration of OCL in Ecore and Xtext is provided.

OCL and Ecore Theisz et al. present a multi-level meta-modeling technique enabling the uniform self-contained interpretation of constraint semantics by encapsulating constraint modeling constructs into their Dynamic Multi-Layer Algebra (DMLA) framework [201], originally presented in 2015. In other words, as opposed to the combination of EMF Ecore meta-models and OCL constraints, in which OCL expressions are treated as external constraint notation or independent layers, their framework enables the interpretation of meta-model semantics including constraints in a uniform fashion. They define the role of the XML schema-based network management configuration language YANG [29] for data modeling purposes as being very similar to the Emfatic[1] textual syntax for EMF Ecore models. They argue that although meta-type definitions work very similarly in both these languages, the introduction of additional type constraints is different. What

[1]The Emfactic language reference is available online at `https://www.eclipse.org/epsilon/doc/articles/emfatic/`.

is more, MOF [168] does not provide a genuine concept for pattern constraints. More specifically, the EMF, i.e. the quasi-reference implementation of MOF, and in particular EMF Ecore, does not provide an integrated solution for the specification of constraints except lower-bound and upper-bound for cardinality in ETypeElements. For all other purposes, OCL constraints are used to introduce restrictions in Ecore. As an alternative to the use of OCL constraints, an ad-hoc solution for Ecore to incorporate additional constraints in the meta-metamodel is to introduce a new type for each constraint type. Doing so, however, does not solve the problem of validation. Their goal is to make constraint definition an integral part of multi-level meta-modeling, as opposed to the external definition of OCL in EMF Ecore, for the purpose of avoiding patchwork caused by the separation of OCL and EMF Ecore and to enable advanced integrated constraint definitions. Their approach builds on the idea of "interchangeable bootstraps" allowing to handle both type and cardinality constraints through validation formulae, as well as model constraints like all other elements in their framework.

OCL and Xtext Willink [217] presents work on the re-engineering of Eclipse OCL that exploits the characteristics of the Xtext framework for enhancing tool support and, in particular, the parsing and evaluation of OCL constraints. This work highlights the limitation of the Eclipse MDT/OCL project[2], which does not require significant semantic analysis in order to create the AST for the OCL specification, resulting in repetitive action code increasing the possibility for the introduction of errors. The resulting implementation reduces the amount of manual input by 80%. Other limitations illustrated by this work include overlapping syntaxes caused by ambiguities in the specification of OCL version 2.2 [167]; the 64 kilobyte size limit that is exceeded by the ANTLR grammar generated for the complete OCL metamodel; and the ambiguity on how to establish Xtext grammar that produces predictable and efficient results.

5.3 Challenges

The challenges addressed by the contribution presented in this chapter include automating the generation of modeling language IDEs with increased accuracy of model validation and the proposal of consistency-maintaining content assist suggestions, as well as consistency-restoring model repair solutions. Consequently, as opposed to the visualization of consistency violations with coarse granularity by indicating the entire model structures as erroneous, the challenge is to isolate inconsistencies and narrow the display to locations in a model where violations actually occur. A further challenge is to produce modeling language IDEs that feature content assist functionality, which proposes suggestions to IDE users that preserve model consistency, as opposed to introducing constraint violations. Additionally, it is necessary to identify how to re-establish the consistency of models which violate constraints. More specifically, model repair solutions which are automatically generated must take complex dependencies between a set of models into

[2]The Eclipse MDT/OCL Project is available online at `https://www.eclipse.org/modeling/mdt/?project=ocl`.

account, as opposed to requiring the manual specification of model repair rules as part of an individual modeling language implementation. As a result, the challenge is to develop an approach that addresses the limitations of search-based optimization approaches, such as MOMoT [81], and in particular address limitations in regards to the delivery of local optimal solutions. More specifically, an approach must be developed that enables the specification of multi-objective search problems (i.e. a result of the manifestation of modeling language definitions consisting of a multitude of constraints), as well as the procurement of neighborhood populations, while aiming to escape local optima solutions.

5.4 Requirements

The requirements to automate the generation of modeling language IDEs that enable increased precision in model validation, consistency-maintaining content assist suggestions, and consistency-restoring model repair solutions include the following.

5.4.1 Precise model validation

First, model inconsistencies must be isolated to facilitate fine-grained error localization. Therefore, the result of the model validation process must deliver precise locations of model inconsistencies. In other words, precise model validation must only include parts of a model that are affected by a consistency violation or a set of consistency violations and, thus, exclude parts of a model which are not affected by an inconsistency or a set of inconsistencies.

5.4.2 Consistency-preserving content assistance

Second, content assist proposals that are suggested by a modeling language IDE must retain model consistency. Therefore, the preservation of model consistency in regards to the application of potential content-assistance proposals are required to be evaluated beforehand. Moreover, sets of consistency-preserving content assist proposals must be sorted according to individual quality and cost, in order to enable the visualization of ranked proposals to IDE users.

5.4.3 Runtime model repair

Third, similar to the requirements associated with content-assistance, sets of model repair solutions must be computed and ranked according to solution quality and cost. Although MOMoT [81] allows the use of model constraints, their applicability is limited. For example, constraint evaluation must be performed externally (i.e. by instantiating a constraint solver), constraints may only be applied globally (i.e. constraint-evaluation based on the entire model, as opposed to objects of a model), and solutions with higher fitness must be indicated. More specifically, the latter behavior (i.e. the selection of neighbors with higher fitness) renders the MOMoT tool unable to produce solutions such as the simultaneous change of multiple features of a model. Therefore, a custom search

approach must be developed that addresses the limitations of search-based software engineering approaches and, in particular, in regards to the generation of change actions that maintain, improve, or restore model consistency.

In summary, it is necessary to develop an approach that enables both multi-objective search, as well as escapes from local optima solutions. As such, it will be capable of increasing model validation accuracy, i.e. in comparison to the generated default model validator; maintenance of model-consistency when applying content assist suggestions; and improvement or recovery of model consistency when applying model repair solutions.

5.5 Approach

The approach has been realized in the IntellEdit framework. The functionality of the framework is applied both for the manifestation of IntellEdit-generated IDEs, as well as during their execution. In what follows, the developed framework is illustrated in terms of the generation of enhanced modeling language implementations and, more specifically, a generic approach for more precise validation of constraints, as well as the production of content assist suggestions and model repair solutions during runtime.

5.5.1 Overview and IDE generation

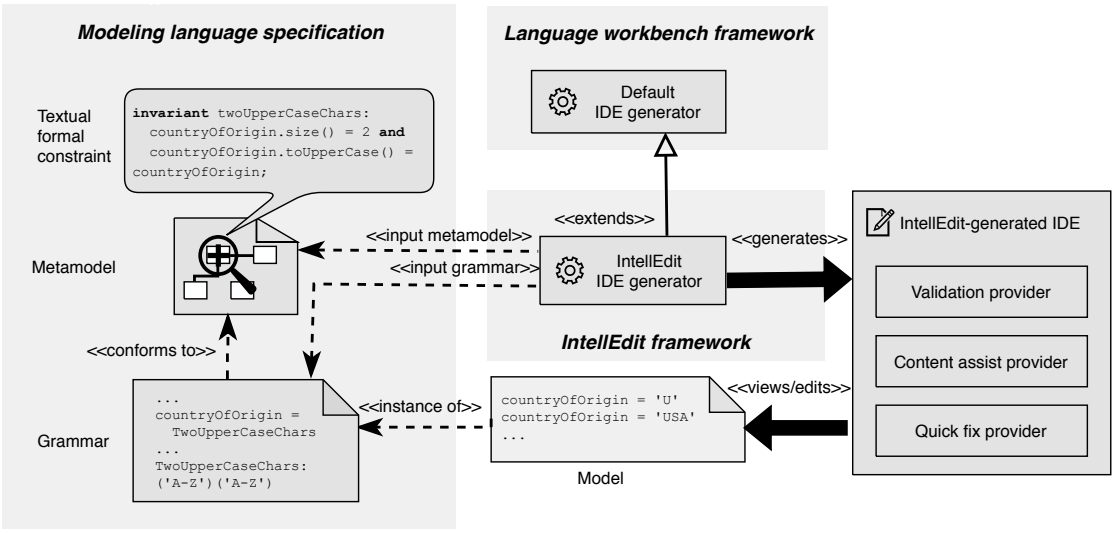

Figure 5.2: Manifestation of IntellEdit-generated IDEs.

IntellEdit represents the reference implementation of the approach that is built on top of the Eclipse Modeling Framework (EMF) [197], Xtext [70], Multiobjective Evolutionary Algorithm (MOEA) framework[3], and OCL [40]. It has been specifically

[3]The MOEA framework is available online at http://moeaframework.org.

created to leverage the power of comprehensive language definitions. As indicated in Figure 5.2, the language engineer first applies the facilities of EMF to specify an Ecore-based *metamodel*, i.e. a common standard for the specification of structural components of a modeling language. Next, the language engineer augments the *metamodel* with *textual formal constraints* in terms of OCL invariants, which enables the definition of restrictions on the structural components captured in a metamodel (henceforth referred to as *constrained metamodel*). In a nutshell, a model is considered valid if it fulfills all the constraints defined by its metamodel. For example, an invariant in the *constrained metamodel* of the space transportation service language restricts the value of the *stages* attribute of *Spacecraft* instances which function as *ORBITAL_LAUNCHER* to an integer number with a value that is less than or equal to *4*. In this case, as opposed to visualizing the entire *Spacecraft* element as erroneous, the IDE will reduce the set of invalid parts of the model and thus provide more precise validation results.

Having completed the definition of the *constrained metamodel*, a conforming *IntellEdit-generated IDE* provides a customized *Validation provider*, *Content assist provider*, and *Quick fix provider*. In order to do so, both EMF default facilities, as well as the *IntellEdit Generator* are employed. Moreover, by separating classes used for validation, content-assistance, and quick fix resolution from other editing classes, *IntellEdit* enables the straightforward injection into automatically generated files.

During the execution of an *IntellEdit-generated IDE*, which is used to construct and manipulate models that represent instances of the modeling language definition, the IDE issues runtime requests to the framework. Subsequently, *IntellEdit*, which employs the MOEA framework during this step, establishes results for validation, content assistance, and model repair, and relays them to the *IntellEdit-generated IDE*.

5.5.2 Model validation

IDEs that are generated by state-of-the-art tools tend to visualize single inconsistencies in terms of the context in which the violation occurs instead of a more exact location, e.g. the union of all minimal error causes. This is problematic, as the language modeler is presented with imprecise information on the correctness and incorrectness of the model. Proper visualization of error causes can contribute to a better modeling process that avoids subsequent errors [181].

Definition of change action requirement types. The approach considers change action requirement types that are constructed and evaluated in evaluation trees. For the purpose of acquiring potentially relevant error locations, a runtime analysis of the expression evaluation is performed by comparing *expected result* with *actual result*, and the return locations where deviations occur. In order to do so, the expected result value is stored as one or more of the following eleven change action requirement types that provide a good basis for finding corrective changes with adequate performance (i.e. without imposing noticeable waiting times to IDE users) in the first two layers of the search algorithm. These change action requirement types may be seen as a specialization of the

```
1  name: fixgen
2  input: f ⊆ ChangeActionRequirements, (e, v) ∈ {(Expression, Result)}
3  output:SURE m : (Expression, Result) → Car // result fixes
4    f ← f \ check(f, v)
5    m ← (e, v) ↦ f else ∀a, b : (a, b) ↦ {}
6    u ← f //unhandled fixes
7    for p ∈ availableGenerators(e.type())
8      (h, m_s) ← p.get(e, v, f)
9      u ← u \ h
10     m ← (a, b) ↦ (m(a, b) ∪ m_s(a, b))
11   if u ≠ ∅
12       for (e_s, v_s) ∈ subresults(e, v)
13           m ← (a, b) ↦ (m(a, b) ∪ ((e_s, v_s), Change))
14   for (e_s, v_s) ∈ subresults(e, v)
15       m_s ← fixgen(m(e_s, v_s), (e_s, v_s)
16       m ← (a, b) ↦ (m(a, b) ∪ m_s(a, b))
```

Listing 5.1: Generation of change action requirements.

general model constraints to basic constraints on individual model element properties. First, different types of expected results, generally corresponding to a simple set of conditions or single values conditions, are represented by equal(v) and different(v) and denote that expressions should have the same value or a different value than the one given. Second, *true* and *false* correspond to equal(*true*) and equal(*false*) respectively. Next, the expected results increase(v) and increaseexcl(v), as well as decrease(v) and decreaseexcl(v) for integer numbers denote that a value should be at least or at most a specific value. Moreover, contains(v) and excludes(v) denote that an expected set of results should or should not contain the value v. Likewise, the anticipated size of an expected set of results is indicated by minsize(c) and maxsize(c). Finally, the definition of change on the expected result indicates that the criteria of a particular expression are not known. In this way, the deviation of common change action requirements from OCL constraints by the application of basic requirement types that include setting properties, performing various collection operations, simple inequalities, boolean operations, and single-parameter functions with a computable inverse set is achieved.

Evaluation of change action requirement types. The basic generation of change action requirements, as defined above, is illustrated by *ChangeActionRequirements* in Listing 5.1. In case the evaluation of a change action requirement yields a positive result, i.e. the requirement is fulfilled, changes of existing sub-evaluations are not required and, hence, propagation is skipped. On the one hand, expression types for which expected results are known may be specified accordingly using a change action requirement generator. On the other hand, expression types for which expected results are not known have their most general expected result, i.e. change, propagated to all sub-expressions.

There are several change action requirement generators which can be employed by the approach. For operations with a finite operation table and a certain target value, all

```
input f : (x_1, ..., x_n) → y Function, c_1, ..., c_n current assignment,
      vexpected value
output r : P((x_1, ..., x_n)) //setminimal changes
  b = f^{-1}(v)
  b_h ← {{(i, k_i)|k_i ≠ x_i}|(k_1, ..., k_n) ∈ b}
  b_h ← b_h \ {A ∈ b_h, ∃B ∈ b_h : B ⊊ A}
  r = {(r_1, ..., r_n)|A ∈ b_h, r_i =if ∃(i, z) ∈ A then z else x_i end}
```

Listing 5.2: Preparation of change action requirement generator operation for finite functions.

```
input: (e, v) ∈ {(Expression, Result)}, fChangeActionRequirements
output: (h, m_s) ∈ {ChangeActionRequirements, Reqtable}
h ← {f_s|f_s ∈ f, ∃x : f_s = equal(t)}
m_s ← (a, b) ↦ {}
for equal(t) ∈ h
    s_r ← {(e_i, x_i)|(e_i, x_i) ∈ subresults(e, v))
    s ← generateChangeReq(e, x_1, ..., x_n, t)
    if s = ∅ //No fix, so don't handle!
        h = h \ equal(t)
    else
     for (e_i, x_i) ∈ s_r //Add requirement values
       m_s = (a, b) ↦ m_s ∪ {equal(t_i)|(t_1, ..., t_n) ∈ s}
```

Listing 5.3: Application of change action requirement generator operation for finite functions.

possibly expected results of a sub-expression are determined from the operation table, i.e. set of operation results for input values. More specifically, assuming an operation $f(x_1, ..., x_n) = v$ with current sub-expression evaluations $y_1, ..., y_n$ for all sub-expressions and expected value v, a suitable change that fulfills $f(y_1, ..., y'_k, ...y_n) = v$ is $\{y_k \to y'_k, ...\}$. The computation of all suitable set-minimal changes is computationally intensive and, thus, performed once for every operation on application startup, as depicted in Listing 5.2. Listing 5.3 illustrates an algorithm that employs a change action to an existing evaluation. More specifically, it handles all equal(x) requirements by computing base values, which render the function to return the expected value x and recursively apply change action requirements.

$\vee$	**A:false**	**A:true**
B:false	A $\to$ *true*, B $\to$ *true*	-
B:true	-	-

Table 5.1: Reduced table which renders the value of the logical disjunction to `true`.

Table 5.1 illustrates an example of a reduction table, which renders the result of the expression containing a logical disjunction (i.e. denoted by $\vee$ and represented by *or* in

```
– Regulation
MAX_STAGE_COUNT := 2;
MAX_STAGE_ENGINE_COUNT := 4;
– Spacecraft
invariant regulationsFulfilled:
  functions → includes(Function::MULTIPLANETARY_TRANSPORT)
  or
  stages → includes(self.engineAmount ≤ MAX_STAGE_ENGINE_COUNT) and
    (if functions → includes(Function::INTERCONTINENTAL_TRANSPORT)
      then stages → size() ≤ MAX_STAGE_COUNT
      else functions → includes(Function::ORBITAL_LAUNCHER) endif);
```

Listing 5.4: Selected constraints in the space transportation service modeling language.

OCL) to *true*. Only in the case of both A and B being false, is propagation triggered. In that case, both A and B may become *true*.

Further, for the boolean conditions $v_1{=}v_2$ and $v_1{\neq}v_2$, and expected values true and false, the expected values can be propagated to equal(v_2) and equal(v_1) for the first and second sub-expressions, i.e. $v_1{=}|{\neq}v_2 := true|false$ respectively. For the rest of the combinations, different can be applied. Similarly, increase and decrease are utilized for the inequalities $<$ and $>$ respectively. Likewise, for inclusion and exclusion in set memberships of expected boolean values, contains and excludes may be used respectively. Furthermore, expected integer number values of set sizes are converted with minsize and maxsize. Inclusions and exclusions of set selection operators, i.e. select, $\{x \in S|\text{cond}(x)\}$, are mapped to inclusions and exclusions of the source set S and an expected value of *true* for every object that must be contained in the set, and *false* for every object that must not be contained in the set. Set collections, i.e. collect, $\bigcup_{y \in S} x \in f(y)$, propagate their excluded elements to $f(y)$ and to remove y from S, where an unwanted value is calculated, and their included elements yield an inclusion in $f(y)$ as well as any additions to S. The monotone operations $\cup$ and $\cap$ allow to propagate the expected results directly. Likewise, additions allow propagating requests that increase and decrease values.

Following the propagation of the expected result, potentially erroneous sub-expressions indicate cases in which the expected result does not match the actual result. As the applied grammar reflects most model access operations, particularly object feature access operations and object instance selection operations, error markers and annotations are placed at locations where sub-expressions are violated and, therefore, the expected result is not valid. By fulfilling change requirements for a sub-expression, this sub-expression may evaluate to true and, thus, contribute to rendering the expression value to true. Note that it may be sufficient to fulfill (only) a subset of available change requirements to (re-)establish model validity.

Listing 5.4 illustrates selected constraints in the space transportation service modeling language, which must be fulfilled to conform to the regulations imposed by the aviation administration. More specifically, the number of engines and stages of a spacecraft are

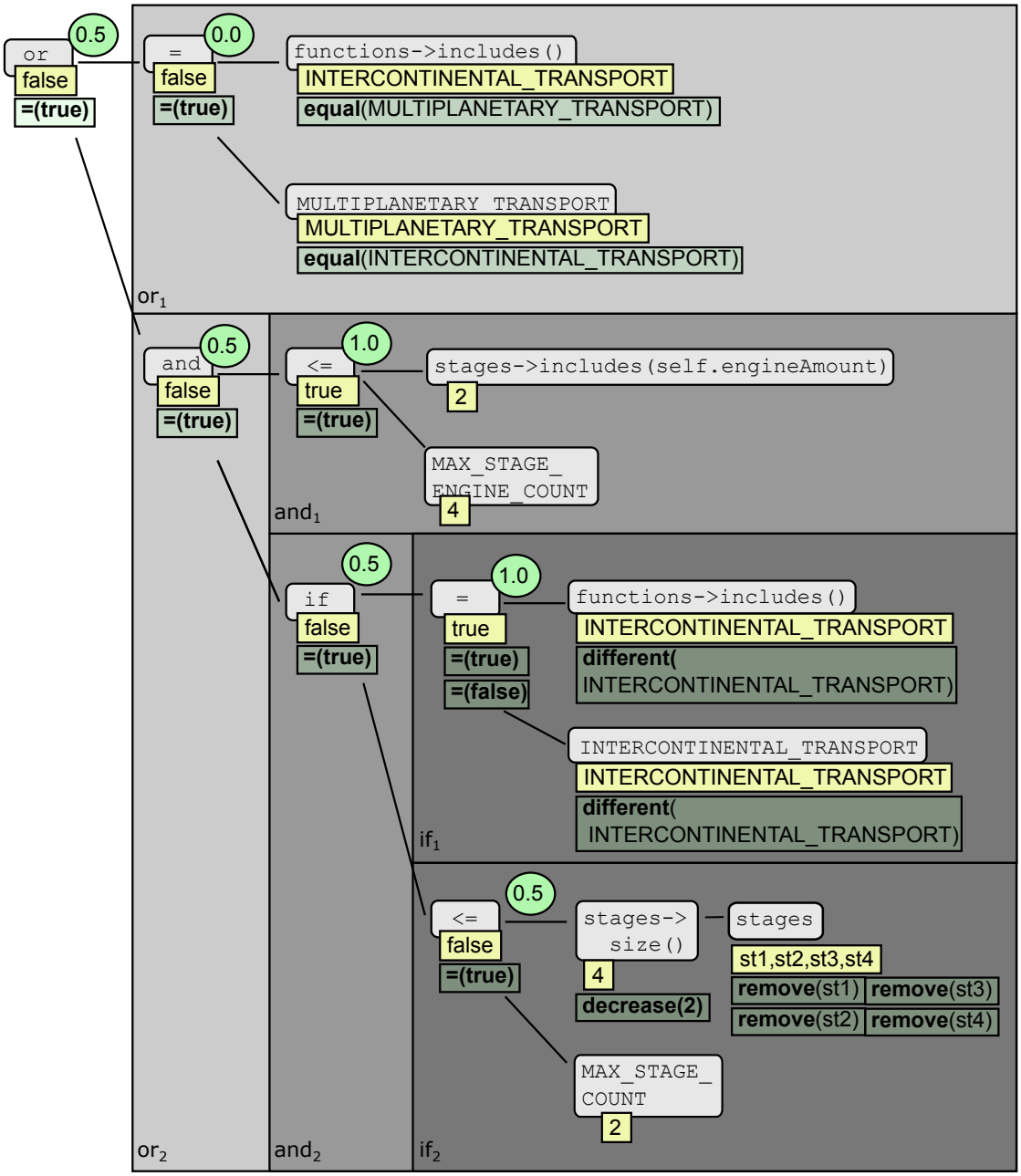

Figure 5.3: Failed evaluation tree for the running example expression (cf. Listing 5.4).

restricted based on the functions of a spacecraft. The launchpads of spacecrafts that function as *MULTIPLANETARY_TRANSPORT* are situated away from populated areas and, thus, impose no restrictions on the spacecrafts themselves in terms of numbers of stages and stage engines. Spacecrafts, however, that function as *ORBITAL_LAUNCHER*, i.e. launching payloads into earth-orbit, or *INTERCONTINENTAL_TRANSPORT*, i.e.

offering transport between cities on earth, are restricted by a maximum of four engines per stage. Moreover, spacecrafts that function as *INTERCONTINENTAL_TRANSPORT* are further constrained by a maximum of two stages. Note that the assignment of functions to spacecrafts follows a hierarchical order. For example, a spacecraft with non-restrictive functionality *MULTIPLANETARY_TRANSPORT* may not (also) function as a more restricted spacecraft, such as *ORBITAL_LAUNCHER*. Likewise, a spacecraft which functions as an orbital launcher may not (also) function as an intercontinental transport vehicle.

The evaluation tree of the failed constraint in Listing 5.4 is shown in Figure 5.11. The syntactic tree is shown in a monospace font. Below each subexpression node, the current result is listed first, followed by the expected result. The numbers in the top right indicate the equality of actual result versus expected result. The constraint failed as a result of the combination of spacecraft function being *INTERCONTINENTAL_TRANSPORT* and the number of *stages* being too high.In other words, the number of *stages* of the spacecraft must be equal to or smaller than *4*, according to aviation administration regulations for spacecrafts that function as orbital launchers or intercontinental transport vehicles and as indicated by *MAX_STAGE_ENGINE_COUNT*. Solutions to restore the validity of a constraint are computed by setting the expected value to *true* and subsequently propagating the result. In the case where at least one subexpression of the expression or evaluates to *true*, the expected value *true* is propagated to both subexpressions. The first subexpression is an *equal*, which is *true* if the value of *functions* includes *MULTIPLANETARY_TRANSPORT*, so the expected value of *functions* is set to *MULTIPLANETARY_TRANSPORT* and the expected value of *MULTIPLANETARY_TRANSPORT* to *INTERCONTINENTAL_TRANSPORT*. In order to make the and boolean expression *true*, both subexpressions must return *true*. The first subexpression returns *true*, such that no further propagation is performed. Similarly, the feature set *functions* does not need further propagation. The second subexpression, however, does not yet return *true*. Hence, if will evaluate to *true* for cases in which the condition is fulfilled and the then-subtree is *true*, or the condition is not fulfilled and the else-subtree is *true*; thus, both cases in the conditional equals are considered. This equals expression is currently *true* and rendered to *false* by setting either subexpression to a different value. The then-part is evaluated to *true* if the feature set *stages* of a spacecraft is lower than or equal to *2*, as dictated by the regulations of the aviation administration and indicated by *MAX_STAGE_COUNT*. This proposes to decrease the feature set *stages* of a spacecraft that functions as an intercontinental transport vehicle.

5.5.3 Content assistance

The approach ranks the discovered suggestions for content assistance so that the most favorable are placed first. In principle, suggestions violating the lowest number of constraints are considered favorable. Constraints are recursively evaluated by matching the closest equivalent of a set of expected values with actual values of subexpressions. The *closeness measure* in the approach is established by computing distance metrics for

types. More specifically, metrics for Strings and numbers are established by computing the Levenshtein distance and the numeric equality $e^{-|v_1-v_2|}$ respectively. Moreover, boolean operators are mapped to double operators, where *true* and *false* map to *0* and *1* respectively. Additionally, and and max are mapped to min and max respectively.

Considering the example in Figure 5.4, the only part of the expression which is not completely true or false is the rightmost inequality, where the inequality is nearly fulfilled (measure 0.5). This value is propagated to the top so the constraint is considered mostly fulfilled.

5.5.4 Model repair

Model repair refers to the action of applying a quick fix solution—placed at locations where model values are changed—that can be executed to increase model quality by decreasing the number of constraints that are violated by a model. Zeller defines the *cause of an effect* as the minimal difference between the world where the effect (i.e. in the case of the approach, a constraint violation) occurs and an alternate world where the effect does not occur [221]. Hence, in order to find the actual cause for a constraint violation, the closest possible world in which the violation does not occur, is searched. Thus, quick fix solutions are established by ranking them according to the least change principle, i.e. favoring solutions that cause minimal differences in the manifestation of their effect.

The approach employs different SBSE techniques for the discovery of quick fix solutions through the application of a three-stage search process. More specifically, the search process involves two types, with varying neighborhood search [95] for the exploration of the search space: local search and global search. Both types of search processes are executed in the background and continuously deliver solutions that are eventually displayed to the IDE user. In case a model change occurs, previously found model repair solutions are re-evaluated based on the current model. In general, new values and value changes are randomly sampled for each specific data type. Moreover, changes are sampled in a way that small changes are more likely than large ones. For example, the likelihood of changing $n+1$ characters is only ten percent of the likelihood of changing n characters.

Local search. The generic approach performs local search in two stages, i.e. a small local search and a large local search. In general, model changes based on a single violated constraint are explored, which either resolve the violation, i.e. the constraint becomes true, or guide the constraint resolution closer to true, i.e. the involved model change has a high score of resolving the violation. A simple hill climbing algorithm with backtracking is used for the local search [8]. As expected, locally resolving an error may yield to new violations in other model areas. Therefore, as a countermeasure, the approach sorts quick fix solutions in terms of their score in resolving the single violation and how many violations exist, i.e. including existing violations and new violations, in the model after applying the solution. The *small local search* explores model changes where the expression analyzer may find concrete possible changes for resolving a single

expression. More specifically, model changes include the application of equal, contains, excludes, increase, and decrease on object features to reach the expected results. The attempted value of model change operations increase and decrease refers to the extent of what is applicable. The *large local search* considers changes for single violated expressions as well. It considers, however, model changes involving the manipulation of all feature values, as well as all object instances that occur as part of a constraint expression in which the evaluated result does not match the expected result.

Global search. The generic algorithm for global search applies genetic search techniques allowing arbitrary model changes on the entire model in order to discover a broad range of model repair solutions that may not have been found yet by the execution of the local search algorithm. Hence, as a short-cut, model repair solutions that result from the execution of the global search algorithm may be applied directly as a result of being selected by the IDE user. Practical implementations may consider multi-objective search algorithms, such as the NSGA-II algorithm, i.e. also applied in *IntellEdit*, which allows the consideration of the change amount, as well as the number of fixed and violated constraints. Hence, the results that are displayed to the IDE user are based on the set of changes that are made available by the current state of the Pareto-front.

The exemplary model (cf. Figure 5.11) may be repaired by a variety of repair actions that directly fulfill the expression and may, thus, be proposed to the IDE user. For example, changing the function of the modeled spacecraft from *INTERCONTINENTAL_TRANSPORT* to *MULTIPLANETARY_TRANSPORT* or decreasing the number of stages from *4* to *2*.

5.5.5 IDE runtime

During the creation and manipulation of models by the IDE user (cf. Figure 5.4), several interactions between the *Advanced modeling language IDE* and *IntellEdit* occur to provide IDE assistance. Moreover, to make the most appropriate use of resources consumed by the framework, any change made to the model is immediately communicated to the framework. The two major components of the framework are represented by the *IntellEdit constraint interpreter* and the *IntellEdit content-assist and quick fix builder*, which contains a *Local search engine* as well as a *Global search engine*.

The general workflow entails the delivery of identified error location details by the OCL interpreter employed by the approach to both the internal content-assist and quick fix builder, as well as the external *Validation provider* in the IDE implementation. As a result, the IDE user is provided with the validation results of the model currently created or manipulated in the editor. Next, the editor's *Content assist provider* requests and retrieves any available results from the content-assist and quick fix builder. Therefore, in the model used here as an example, a line stating "`countryOfOrigin = 'U'`" for the spacecraft will result in the content-assist builder to prefer to insert character "`A`" at the current cursor location. Finally, the editor's *Quick fix provider* is continuously supplied with new quick fix solutions by the content-assist and quick fix builder. Hence, as soon

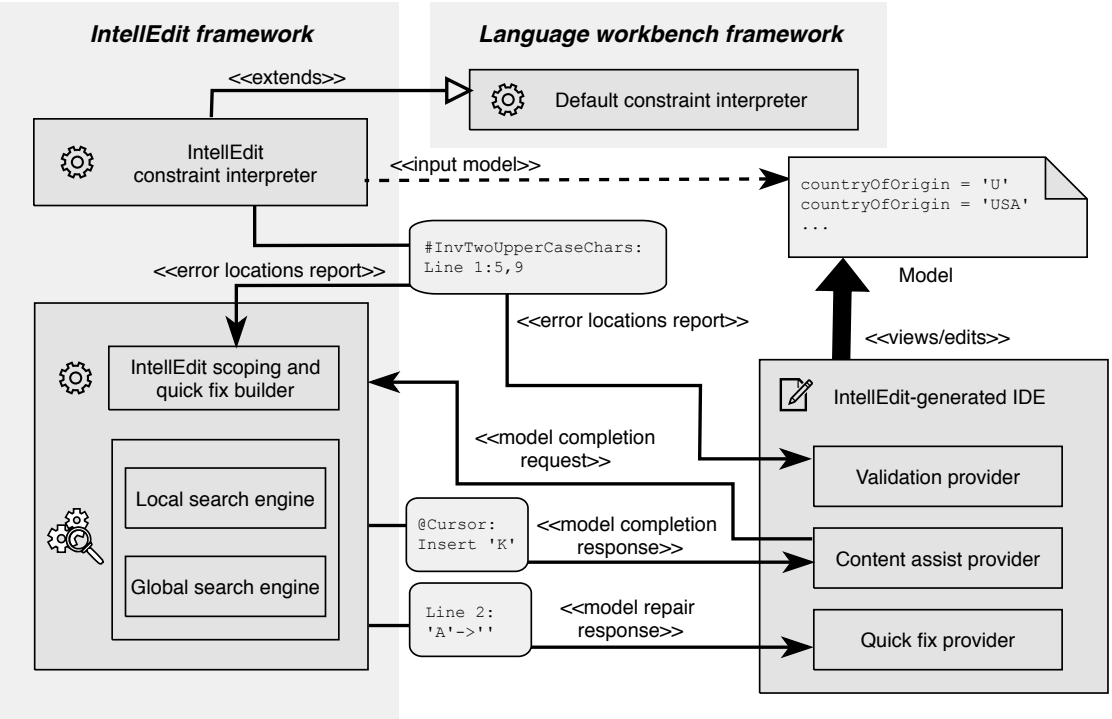

Figure 5.4: Runtime of IntellEdit-generated IDEs.

as solutions, such as inserting the character "A" in line one or removing the character "A" in line two, are found, they are made available to the quick fix provider and can, therefore, be applied by the IDE user operating the editor.

5.5.6 Restriction integration and application example

The following presents the integration of the approach presented in this chapter with the XML Schema modeling integration and assistance approach presented in Chapter 4 and, in particular, the integration of IntellEdit with XMLText in the Intelligent XML to Xtext Editing (XMLIntellEdit) framework [165]. More specifically, the following illustrates the application of a language for libraries of books and customers, i.e. defined by means of an XML Schema definition, with XMLIntellEdit by focusing on the facilitation of XML Schema restrictions for the generation of enhanced modeling language IDEs.

Figure 5.5 presents an overview of the XMLIntellEdit framework; a more detailed illustration is provided by Appendix 2. XMLText realizes the modernization of XMLware-based languages with modelware and grammarware by the transformation of XML Schema definitions to metamodels (cf. 1a); the refactoring of metamodels to facilitate the production of modeling language grammars (cf. 2); the generation of modeling language implementations from metamodels (cf. 3); and the ability to perform

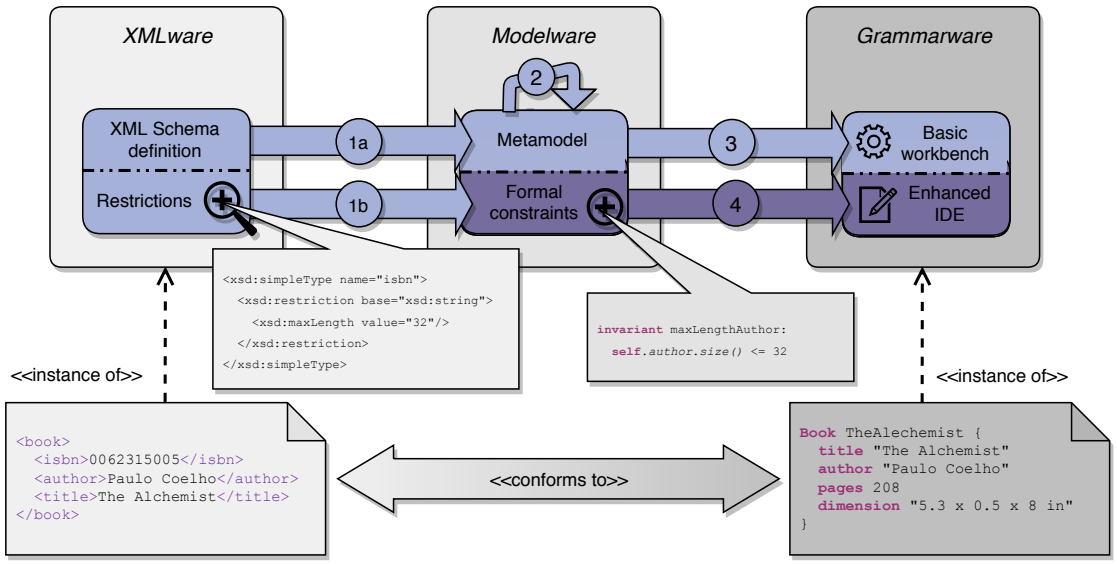

Figure 5.5: Overview of the XMLIntellEdit framework.

round-trip transformations for language-conforming instances. IntellEdit functions as a framework for the automated generation of consistency-achieving modeling language IDEs by addressing the limitation of state-of-the-art language workbench frameworks in the preservation of model consistency for elaborated language-specific constraints. As a result, the necessity to handcraft the implementation of dedicated *basic workbenches* and *enhanced IDEs* (i.e. offering the capability of precise model validation, consistency-preserving content assistance, and consistency-restoring model repair) is made redundant for languages that are defined by a *metamodel* and *formal constraints*.

Modeling language IDEs that are generated by IntellEdit, however, employ a runtime plugin which facilitates formal constraints in metamodels for computing change actions that are eventually displayed as model repair solutions to IDE users (cf. Section 5.5.5). Thus, in order to integrate XMLText with IntellEdit and subsequently enable the use of the IDE runtime plugin (i.e. performing a three-stage neighborhood search process for the assembly of change actions), it is necessary to supply formal constraints alongside language definitions. More specifically, *restrictions* in XML Schema definitions are facilitated for the generation of *formal constraints* and *enhanced IDEs* (cf. 1a and 4 respectively). As a result, the developed three-stage neighborhood search process is executed and, thus, computes the following kinds of concrete change actions. First, the small local search algorithm computes change actions that resolve single expression violations. Second, the large local search algorithm computes change actions that resolve single expression violations but takes into account all feature values and object instances that are involved in single expression violations. Third, the global search algorithm computes arbitrary change actions in the entire model.

```
1 <xs:simpleType name="nameType">
```

```
  <xs:restriction base="xs:string">
    <xs:maxLength value="32"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="isbnType">
   <xs:restriction base="xs:string">
    <xs:pattern value="[0-9]{3}-[0-9]{2}-[0-9]{4}-[0-9]{3}-[0-9]"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="dimensionType">
   <xs:restriction base="xs:string">
    <xs:pattern value="([0-9]|([1-9][0-9]+))([.][0-9]+)? x ([0-9]|([1-9][0-9]+)
        )([.][0-9]+)? x ([0-9]|([1-9][0-9]+))([.][0-9]+)? (centimeters|cm|in|
        inches)"/>
  </xs:restriction>
</xs:simpleType>

<xs:complexType name="bookType">
  <xs:sequence>
    <xs:element name="name" type="xs:ID"/>
    <xs:element name="title" type="xs:string"/>
    <xs:element name="author" type="nameType"/>
    <xs:element name="pages" type="xs:int"/>
    <xs:element name="dimension" type="dimensionType"/>
  </xs:sequence>
  <xs:attribute name="isbn" type="isbnType" use="required"/>
</xs:complexType>

<xs:complexType name="customerType">
    <xs:sequence>
       <xs:element name="firstName" type="xs:string"/>
       <xs:element name="lastName" type="xs:string"/>
       <xs:element name="borrowedBookId" type="xs:IDREF" minOccurs="0"/>
    </xs:sequence>
</xs:complexType>
```

Listing 5.5: Library language XML Schema definition (excerpt).

Listing 5.5 depicts an excerpt from an XML Schema definition file of the library language[4]. Instances of the XML Schema complex type named *customerType* may specify values for attributes *firstName*, *lastName*, and *borrowedBookId*. Similarly, instances of the XML Schema complex type named *bookType* may define values for attributes *name*, *title*, *author*, *pages*, *dimension*, and *isbn*. Moreover, the XML Schema simple type named *nameType*, *dimensionType*, and *isbnType* defines valid values which instances of the attribute named *author*, *dimension*, and *isbn* (i.e. employed by an XML Schema complex type named *bookType*) may hold respectively. More specifically, *nameType*, *dimensionType*, and *isbnType* define restrictions based on XML Schema type *string* by

[4]The complete version of the library language XML Schema definition can be found in Appendix 7.

restricting valid character sequences by a maximum length in the former and a matching pattern in the latter two.

Valid instances of *dimension* must match the pattern defined by *dimensionType* and, thus, conform to the format "`X x Y x Z U`" (i.e. with `X`, `Y`, `Z`, and `U` indicating the length of each axis in a three-dimensional space and measurement unit respectively). Moreover, valid instances of dimension are single or multi-digit and integer numbers, or dot-separated float numbers. Valid instances of measurement unit are either `centimeters`, `cm`, `in`, or `inches`. Valid instances of *isbn* must match the pattern defined by *isbnType* and, hence, conform to the 13-digit ISBN format "`III-GG-PPPP-TTT-C`" (i.e. with `III`, `GG`, `PPPP`, `TTT`, and `C` indicating the international article number, group, publisher, title, and check digit respectively) as defined by the ISO standard 2108:2017 [107].

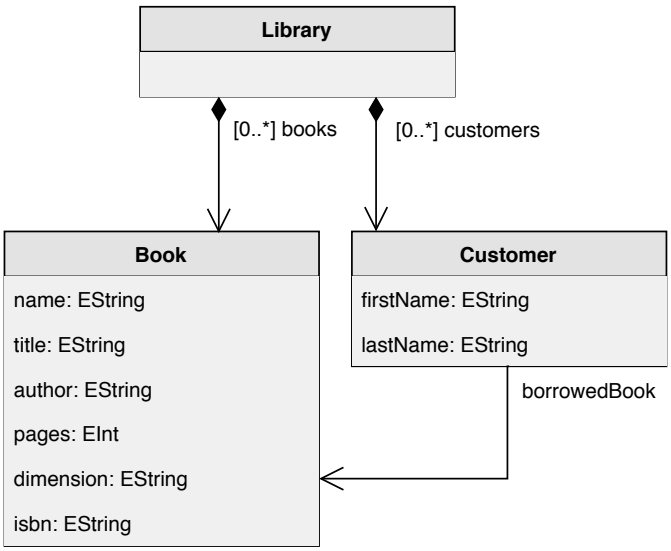

Figure 5.6: Library language metamodel (excerpt).

```
– Book
invariant maxLengthAuthor:
  self.author.size() ≤ 32;
invariant patternDimension:
  self.dimension.matches('
    ([0-9]|([1-9][0-9]+))([.][0-9]+)? x
    ([0-9]|([1-9][0-9]+))([.][0-9]+)? x
    ([0-9]|([1-9][0-9]+))([.][0-9]+)?
    (centimeters|cm|in|inches)');
invariant patternIsbn:
  self.isbn.matches('[0-9]3-[0-9]2-[0-9]4-[0-9]3-[0-9]');
```

Listing 5.6: Selected constraints in the library modeling language.

Figure 5.6 and Listing 5.6 illustrate the metamodel and selected formal constraints of

the library modeling language (i.e. allowing to capture libraries of books and customers) generated by XMLIntellEdit through the supply of the XML Schema definition of the library modeling language. Figures 5.7-5.9 depict screenshots of the library modeling language IDE captured at runtime, and illustrate consistency-recovering model repair solutions that are computed for the value of the attributes named *author*, *dimension*, and *isbn* by ranked recursive constraint evaluation. More specifically, a closed match of a set of expected values with actual values of subexpressions is selected based on the Levenshtein distance metric [143].

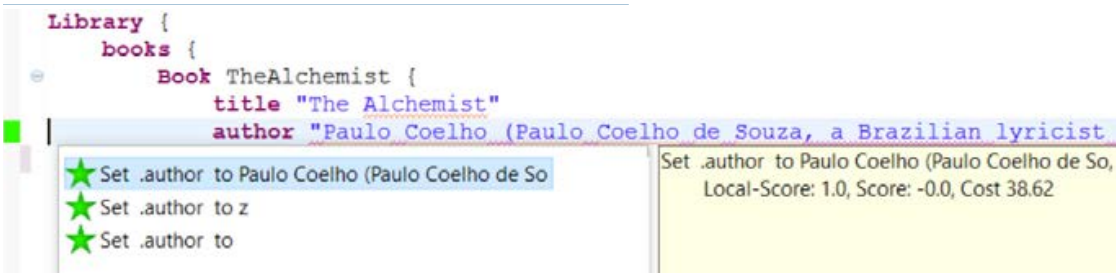

Figure 5.7: Model repair solutions for *author* attribute value in IntellEdit-generated library modeling language IDE.

In the example illustrated in Figure 5.7, the highest ranked solution is represented by a change action that replaces the value of attribute named *author* with the first 32 characters of the value of attribute named *author*.

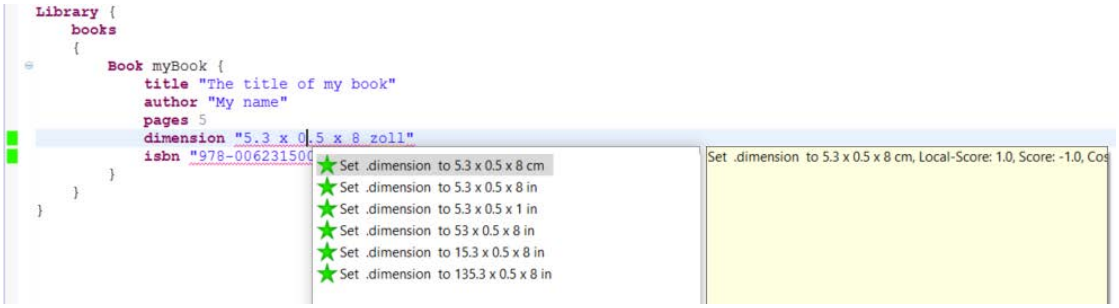

Figure 5.8: Model repair solutions for *dimensions* attribute value in IntellEdit-generated library modeling language IDE.

Figure 5.8 depicts an example in which the highest ranked solution is represented by a change action that replaces the value "`5.3 x 0.5 x 8 zoll`" of attribute named *dimension* with the value "`5.3 x 0.5 x 8 cm`" (i.e. essentially replacing the inconsistent measurement unit "`zoll`" with a consistent measurement unit "`cm`"). Note that in this case the first two solutions (i.e. "`5.3 x 0.5 x 8 cm`" and "`5.3 x 0.5 x 8 in`") are equally ranked; however, due to alphanumeric ordering, the solution with value ending in "`cm`" is displayed beforehand.

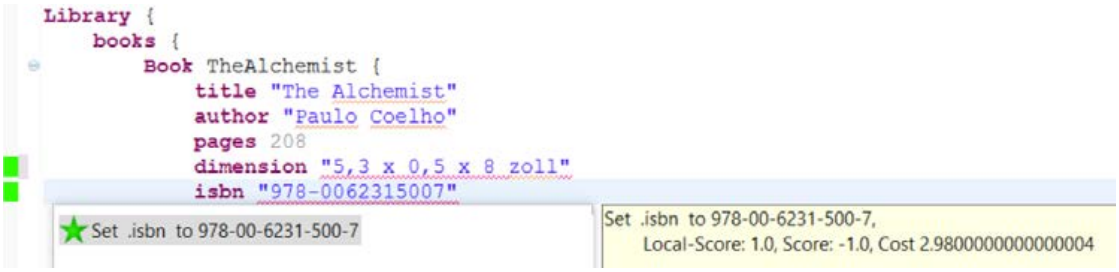

Figure 5.9: Model repair solutions for *isbn* attribute value in IntellEdit-generated library modeling language IDE.

Figure 5.9 depicts the highest ranked solution in a change action that replaces the value "978-0062315007" of attribute named *isbn* with the value "978-00-6231-500-7" (i.e. essentially adding hyphen characters to fit the formatting of ISO standard 2108:2017 [107]).

5.5.7 Prototype implementations

The approach introduced in this chapter has been prototypically realized using the EMF [197] and Xtext [70]. Additional information, such as slides, source code, and examples, of the IntellEdit and XMLIntellEdit prototype are provided at dedicated web pages: `https://intelledit.big.tuwien.ac.at` and `https://xmlintelledit.big.tuwien.ac.at`.

5.5.8 Limitations

First, the propagation for operations with a finite operation table and a certain target value has (only) been implemented for the boolean operations and, or, not, implies, and xor. Second, if the type of a collection source set S is finite, additions that do not yield a matching $f(y)$ are not (yet) avoided in the prototypical implementation. Third, the content-assist algorithm considers feature additions and updates from at most 1,000 domain feature values. Fourth, although the application of IntellEdit-provided model repair solutions, i.e. relying on the definition of cause and effect [221], may resolve errors, IDE users may categorize them as workarounds as opposed to desirable solutions. Fifth, for specific expressions, such as and, the OCL evaluation engine may only produce parts of the result. For example, in case the first subexpression is *false*, the second subexpression will not be evaluated and, thus, will cause the number of displayed validation errors to be reduced. This behavior, however, may be adjusted by altering the implementation of the evaluation engine. Sixth, an arbitrary function call that combines the value of more than one variable is handled by the propagation of a change. Thus, errors in an OCL constraint with a function call as defined in the latter may only be repaired by applying model repair solutions that are found in the second and third layers of the search algorithm.

5.6 Evaluation

The evaluation of the approach implemented by the IntellEdit framework has been conducted separately for validation, content-assistance, and quick fix resolution.

5.6.1 Research questions and evaluation criteria

The validation results provided by IntellEdit are evaluated based on whether they improve the estimation of locations where a model should be changed to resolve errors. The purpose of content-assist suggestions provided by IntellEdit is to support IDE users in the development of consistent models. The purpose of evaluating model repair solutions provided by IntellEdit is to determine whether they lead to a model with fewer violated constraints.

Accordingly, the following research questions are formulated:

RQ1: *Is the validation result produced by* IntellEdit *more precise than that produced by Xtext?*

RQ2: *Is the application of content-assist suggestions produced by* IntellEdit *consistency-preserving?*

RQ3: *Is the application of model repair solutions produced by* IntellEdit *consistency-restoring?*

In regards to *RQ1*, the criteria for precise validation are defined by the accuracy of the error feature location. In other words, a validation result is precise if and only if it indicates the location where the fault has originally been introduced. More specifically, error locations are considered to be correct if the feature is indicated as erroneous for every deleted feature value; the container of the feature is indicated as erroneous for deleted objects; and the whole object is indicated as erroneous for created objects. Furthermore, if a complete object is indicated as erroneous, its features are indicated as erroneous as well. Finally, precision and recall of the approach and of Xtext are determined by comparing the set of erroneously indicated features and objects.

In regards to *RQ2*, the criteria for evaluating content-assist suggestions are based on the number of violated constraints by comparing the application of content-assist suggestions produced by IntellEdit and the Xtext framework.

In regards to *RQ3*, the criteria for evaluating model repair solutions are based on the number of constraint violations that exist in a model before and after the application of a model repair solution.

5.6.2 Procedure

The subject of study is represented by a modeling language for modeling service clusters, and defined by a metamodel and a set of formal constraints, i.e. facilitated by the approach for the manifestation of enhanced IDE support, as described below.

Metamodel. The modeling language metamodel (cf. Figure 5.10) is composed of classes representing services, clusters, and servers. A Service is defined in terms of *designSpeed*, i.e. an integer number representing the speed for which a service is intended to be used, and *type*, which may either be *BESTEFFORT*, *WEAKCONTRACT*, *FAILSAFE*, or *IMPORTANT*, and has to be provided by exactly one cluster. A Cluster has a *designSpeed*, i.e. an integer number representing the speed for which a cluster is intended to be used, as well as one or multiple containing servers. Moreover, a cluster is associated with one or more services, and up to one backup cluster. Finally, each Server provides a *speed*.

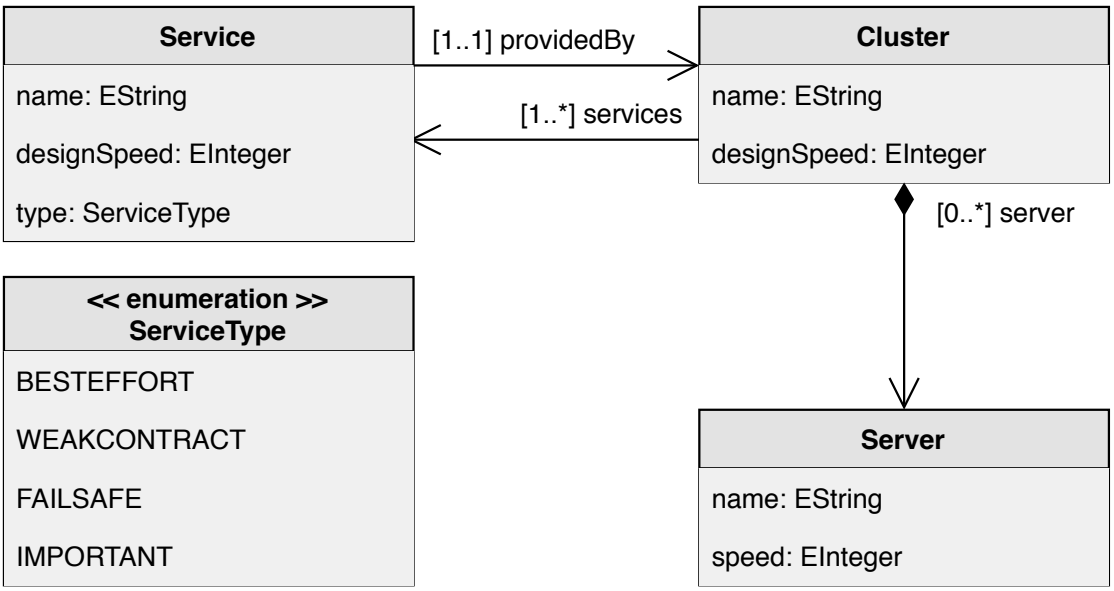

Figure 5.10: Modeling service cluster metamodel.

Formal constraints. Formal constraints are defined for servers, clusters, and services (cf. Listing 5.7). For example, a Service is constrained by the type of service that it provides. More specifically, it either has to have a *BESTEFFORT* service type with no further restrictions or, in the case of a *WEAKCONTRACT* service type, its associated cluster has to be designed for a speed that is equal to or greater than the designed speed of the service itself. A *FAILSAVE* service type means that a backup cluster has to be provided. The *IMPORTANT* service type further extends the restrictions associated with the *FAILSAFE* service type by requiring the designed speed of the associated backup cluster to be equal to or greater than the designed speed of the service itself.

Models. A set of randomized service cluster models are generated so that each model contains twenty objects, and 100 associations and feature values. Moreover, erroneous changes are introduced on valid models at arbitrary locations. Figure 5.11 illustrates an example of a service cluster model. More specifically, it specifies a single *IMPORTANT* service with speed *4* that is provided by Cluster *WebCl*. This cluster has a low-speed

```
– Service
invariant speedFulfilled: type =
  ServiceType::BESTEFFORT or (
    designSpeed ≤ providedBy.designSpeed and
      (if type = ServiceType::IMPORTANT then
        designSpeed ≤ providedBy.backup.designSpeed
      else type = ServiceType::WEAKCONTRACT or
       providedBy.backup ≠ null endif));
```

Listing 5.7: Selected constraints in the service cluster modeling language.

backup, which is not sufficient (cf. speedFulfilled) since its design speed is lower than the speed required by the main service. In the following, the handling of extended validation, content-assistance and quick fix solution providers is illustrated based on this example.

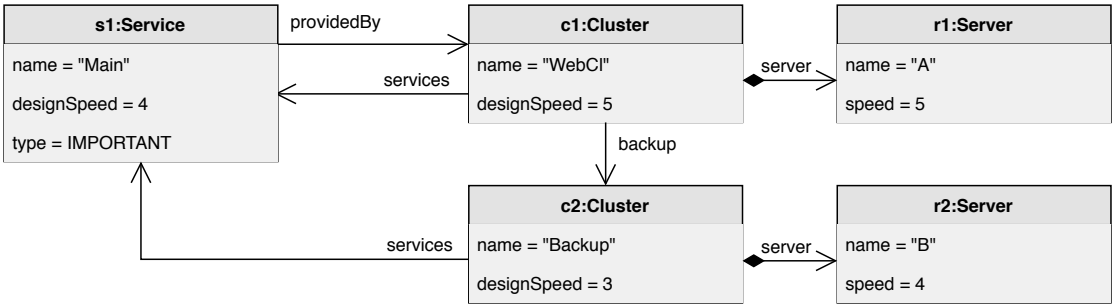

Figure 5.11: Modeling service cluster example model.

Evaluation tree. The evaluation tree of the failed constraint of the service cluster modeling language is shown in Figure 5.12. The syntactic tree is shown in a monospace font. Below each subexpression node, the current result is listed first, followed by the expected result. The numbers at the top right indicate the equality of the actual and expected results by means of a boolean value.

Model validation

The model validation results produced by the cluster service modeling language IDE generated by Xtext and IntellEdit from the example model introduced above, are illustrated in the left-hand and right-hand sides of Figure 5.13 respectively. In particular, as opposed to visualizing all lines of the modeled service as erroneous, thus, indicating that only these may be changed, the IntellEdit-generated IDE indicates the value assignment-part of all lines which impact the consistency of the model. More specifically, the IntellEdit-generated IDE indicates that the value of features *designSpeed*, *type*, and *providedBy* of Service named *Main*, the value of reference *backup* of Cluster named *WebCl*, and the value of feature *designSpeed* of Cluster named *Backup* render the model inconsistent. In other words, it shows that the Service named *Main* with value *4* and *IMPORTANT* of feature *designSpeed* and *type* respectively, must reference a Cluster which

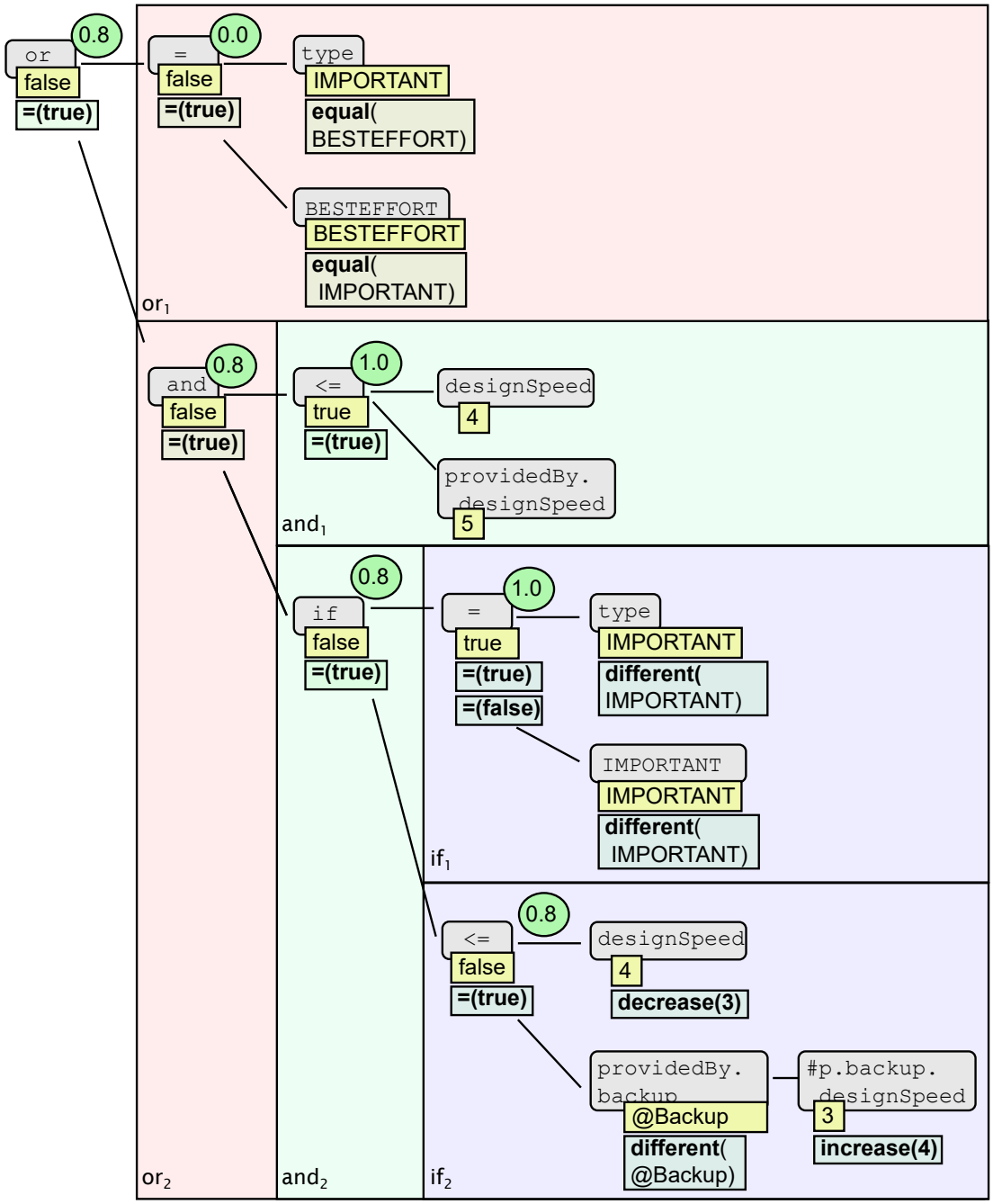

Figure 5.12: Failed evaluation tree for the service cluster expression in Listing 5.7.

references a *backup* cluster with value equal to or greater than *4* of feature *designSpeed* (i.e. ensuring that at least the same speed is available for the modeled service in case Cluster named *WebCl* becomes unavailable and Cluster named *Backup* must take over).

Figure 5.13: Model validation in Xtext-default (left-hand side) and IntellEdit-generated service cluster modeling language IDE (right-hand side) respectively.

Content assistance

The constraint failed because the type was *IMPORTANT*, as opposed to *BESTEFFORT*, and the *designSpeed* provided by the backup service was too low. In other words, the *designSpeed* of a backup service must be equal to or greater than the *designSpeed* of a service of type *IMPORTANT* for which it acts as backup service. Solutions to restore the validity of a constraint are computed by setting the expected value to

true and subsequently propagating the result. In case at least one subexpression of the expression or evaluates to *true*, the expected value *true* is propagated to both subexpressions. The first subexpression is an *equal*, which is *true* if the value of *type* is equal to *BESTEFFORT*, so the expected value of *type* is set to *BESTEFFORT* and the expected value of *BESTEFFORT* to *IMPORTANT*. In order to make the and boolean expression *true*, both subexpressions must return *true*. The first subexpression returns *true* so that no further propagation is performed. Similarly, the feature *designSpeed* does not need further propagation. The second subexpression, however, does not yet return *true*. Hence, if will evaluate to *true* in cases where the condition is fulfilled and the then-subtree is *true*, or the condition is not fulfilled and the else-subtree is *true*; thus, both cases in the conditional equals are considered. This equals expression is currently *true* and rendered to *false* by setting either subexpression to a different value. The then-part is evaluated to *true* if the *designSpeed* of a service is lower than or equal to the *designSpeed* of its associated backup service, thus, suggesting to decrease or increase the *designSpeed* of a service or its associated backup service respectively.

As a result of content-assist suggestions provided by the approach for the purpose of constructing consistent models, the following evaluation procedure is employed. First, model containment hierarchies are randomly generated. Second, the result of the content-assist suggestions are applied for feature assignment. More specifically, the content-assist suggestions employed have the highest score in 0% (i.e. randomly), 25%, 50%, 75%, and 100% of the cases, i.e. simulating the combination of random suggestions with IntellEdit-provided suggestions. Finally, the number of violated constraints is compared.

Model repair

The goal of model repair is to reduce the number of violated constraints in the model by employing quick fix solutions generated by the approach. More specifically, ten IntellEdit quick fix suggestions, i.e. created as a result of the three-stage search process, are applied to each model that has been generated based on random values. Moreover, in order to mitigate the risk of choosing repair solutions which cause the removal of violated parts of the model subsequently yielding empty models, their selection is governed by the number of newly fulfilled violated constraints, as opposed to the number of constraint violations which remain in the model.

Model repair solutions produced by the *local search* algorithm of the IntellEdit-generated cluster service modeling language IDE from the example model introduced above are depicted in Figure 5.14. More specifically, model repair solutions are ranked according to individual cost and displayed in a context menu; solutions associated with the lowest cost are ranked first. Model repair solutions that are produced by the local and global search algorithms are indicated by a green and blue star respectively (cf. Figure 5.15) in the context menu. The model repair solution listed first in the context menu depicted in Figure 5.14 presents a single change action that involves changing the value of feature *designSpeed* of Service named *Main* to *3* from *4*. As a result of applying the latter model repair solution, the value of *designSpeed* of Cluster named *Backup* is

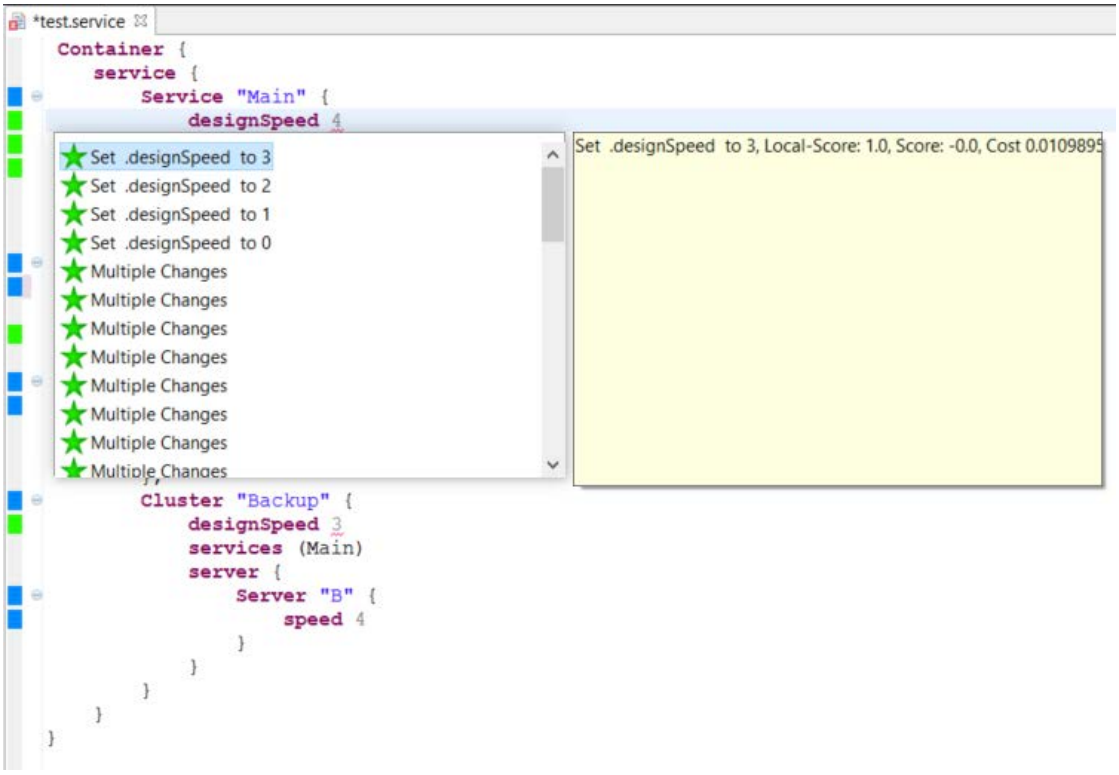

Figure 5.14: Local search model repair solutions in IntellEdit-generated service cluster modeling language IDE.

sufficient and the consistency of the model is restored. In other words, the *designSpeed* of the backup cluster is equal to the *designSpeed* of the modeled service.

Model repair solutions produced by the *global search* algorithm of the IntellEdit-generated cluster service modeling language IDE from the example model introduced above are illustrated in Figure 5.15. The same syntax and semantics apply as for the model repair solutions depicted in Figure 5.14 and described above. Figure 5.15, however, visualizes the selection of a model repair solution produced by the global search algorithm which is associated with a higher value in cost (i.e. $\tilde{2}.02$ as opposed to $\tilde{0}.01$) and involves a series of change actions as opposed to the single change action selected in Figure 5.14. More specifically, the selected solution represents a lower ranked solution consisting of a set of change actions including the change of the value of the feature named *type* in Service named *Main* to *BESTEFFORT* from *IMPORTANT*; the change of the value of the feature named *designSpeed* in Cluster named *WebCl* to *5* from *4*; the change of the value of the feature named *name* in Server named *A* to *AOA* from *B*; and the change of the value of the feature named *designSpeed* in Service named *Main* to *11* from *4*. In other words, the solution illustrates a less prioritized model repair solution that has been produced by global search (i.e. issuing a genetic search algorithm). Note that

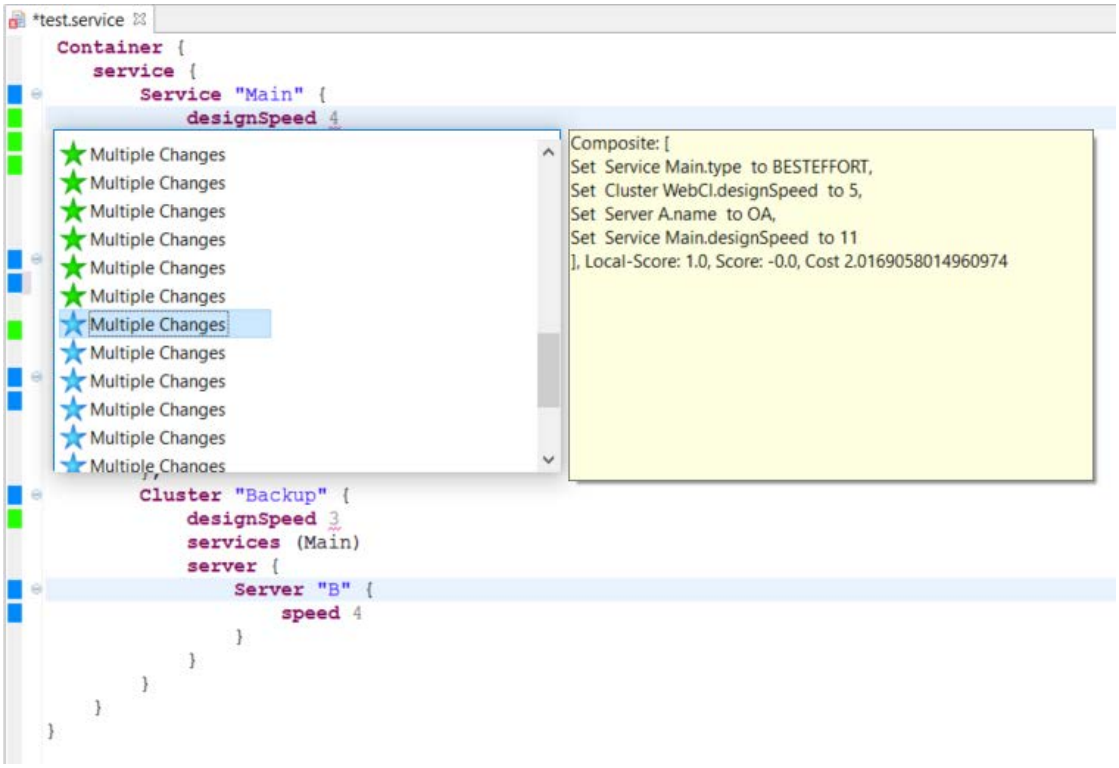

Figure 5.15: Global search model repair solutions in IntellEdit-generated service cluster modeling language IDE.

model repair solutions that are produced as described above may generate favorable solutions in cases that involve a series of complex change actions that may be difficult to be established by end users of modeling language IDEs.

5.6.3 Results

The following presents the results of employing Xtext and the IntellEdit framework in terms of the capabilities of the generated implementations for model validation, content assist, and model repair.

Model validation

Table 5.2 shows the evaluation results of the IntellEdit validation compared with the Xtext validation for indicating erroneous locations in a model. In total, 6,442 evaluations were performed on 50 generated models. The numbers on the left represent the average precision and recall per single changed feature, and those on the right the average precision and recall per single change. Hence, the left-hand numbers focus on large changes and the right-hand numbers on small changes. The results state that the validation precision achieved by IntellEdit is nearly three times as high as that achieved by Xtext. Thus,

the IntellEdit validation displays only erroneous features instead of indicating the entire object as erroneous. Hence, it provides more accurate evidence on which parts of the model should be changed. For example, in case a metamodel containing formally specified consistency constraints, such as restricting the `speed` attribute of a `Server` element to be represented by an integer number that is greater than `0`, is used to derive a modeling language IDE, it does not take into account such restrictions for displaying an appropriate validation result. More specifically, it highlights the entire `Server` element as erroneous and does not provide any quick fix to solve the violated constraint.

Feature	IntellEdit	Xtext
Precision	44.8% / 55.1%	14.0% / 21.4%
Recall	37.5% / 77.1%	41.5% / 65.2%
F-Measure	0.408 / 0.643	0.209 / 0.322

Table 5.2: Validation evaluation results (weighted/unweighted).

In terms of recall, IntellEdit produces a higher recall per expression, i.e. performing better on expressions leading to small model changes, and a lower recall per edited feature. The recall for larger changes in the evaluation results, however, is low overall. Hence, the validation of IntellEdit doubles the F-measure compared with Xtext.

In terms of ***RQ1***, the validation result produced by IntellEdit (i.e. while keeping recall measures of Xtext intact) indicates a three-fold improvement in precision in comparison to the precision produced by Xtext.

Content assistance

The results depicted in Table 5.3 indicate that the IntellEdit content-assist provider is effective in preserving constraints in models. On the one hand, highly scored IntellEdit content-assist suggestions, i.e. suggestions that are listed at the top, do not introduce constraint violations in the model. On the other hand, valid random assignments that do not consider eventual restrictions introduced by formal constraints, i.e. similar to those suggested by the Xtext default content-assist provider, are likely to introduce constraint violations in the model. Thus, in terms of ***RQ2***, the application of content-assist suggestions that are produced by IntellEdit alone are consistency-preserving.

Ratio of IntellEdit suggestions	0%	25%	50%	75%	100%
Constraint violation count average	8.8	6.5	3.8	2.4	0.0

Table 5.3: Constraint violations in the generated models.

Model repair

The results show that applying quick fix suggestions on 97 randomly generated models leads to a reduction in the total number of violated constraints in 89 of the same models. Moreover, the number of violated constraints could not be improved in eight models. More specifically, one model contained the same number of violations and seven models introduced further violations. Therefore, in terms of ***RQ3***, the application of model repair solutions that are provided by IntellEdit are consistency-restoring and, in particular, reduce the number of violated constraints from 7.9 to 2.6 (≈67%).

5.6.4 Threats to validity

The validity of the evaluation is limited by the subject of study, i.e. a modeling language for designing service clusters, as well as randomly generated models, and hence may not be representative for languages of different domains. Moreover, randomly generated faults introduced in models may deviate from faults that are introduced by human users. Similarly, the conflict-resolving change actions of human users may deviate from change actions that are implied by content-assist and model repair solutions produced by the approach. For instance, in the constraint violation in the example, a human user may aim at increasing the speed of existing servers or adding new servers to meet the service requirements, as opposed to weakening the service type from *IMPORTANT* to *BESTEFFORT*. In other words, a human user may prefer the latter change action over a series of change actions as a result of the lower number of change actions (i.e. a representation of cost) required to obtain model consistency. Further, reverting erroneous changes in error feature location is assumed to be correct. Different change actions may exist, however, that lead to desired results. Hence, the evaluation does not clarify whether increased precision leads to more desirable results for IDE users. Likewise, the preference to create consistent, as opposed to inconsistent, models does not imply that the desired result is arrived at with increased effectiveness, i.e. by a lower number of steps and effort on the part of the IDE user. Although content-assist suggestions provided by the approach are consistency-preserving and thus lead to consistency-maintaining models, it cannot be claimed that effectiveness is increased in comparison to the employment of change actions that do not preserve model consistency or introduce new inconsistencies, such as those provided by the Xtext default modeling language IDE.

5.6.5 Summary

*Precision of validation results (**RQ1**)*: The presented evaluation indicates that IntellEdit raises the precision of validation results by a factor of three, while maintaining the measurement of recall.

*Consistency-preservation of content-assist suggestions (**RQ2**)*: The results indicate that the combination of IntellEdit content-assist suggestions and valid random assignments, such as those suggested by Xtext default IDEs, introduces model constraint violations. Choosing IntellEdit content-assist suggestions only, however, eliminates the introduction of constraint violations in the model.

*Consistency-preservation of model repair solutions (**RQ3**)*: The application of model repair solutions that are provided by IntellEdit have been shown to reduce the number of violated constraints by a factor of approximately two over three (≈67%). Thus, IntellEdit indicates locations in the model that are more likely to be relevant for repairing violated constraints, produces consistency-preserving change actions through content assistance, and seeks to restore model consistency by applying model repair solutions which induce the lowest cost measured in change actions.

5.7 Analysis

In general, the generation of language grammar from metamodels through MDE techniques and MDLE frameworks still suffers from several limitations, such as the ability to store values for data type instances, and hence requires extensive manual customization and extension by language engineers [162]. Moreover, to the best of the author's knowledge, no approach has been proposed in the technical space of grammarware that would allow for the derivation of formal constraint-refined metamodels from language grammars. The inherent limitations imposed by Context-Free Grammars (CFGs) render the construction and maintenance of complex structural constraints for languages defined within the technical space of grammarware infeasible (cf. Section 2.3.2). Formal constraints, however, have become key components in MDE for expressing different kinds of (meta)model queries, as well as the specification and manipulation of language requirements. Section 5.7.1 below presents and compares a series of related approaches on model repair with the approach presented in this chapter. Section 5.7.2 presents an analysis of language workbenches in regards to their capability to restrict language definitions with formal constraint specifications, as well as to enable the automated generation or manual specification of model validation, content assist, and model repair mechanisms.

5.7.1 Model repair

Egyed et al. and Reder et al. [66, 180, 181] present an approach to assist IDE users in fixing inconsistencies in UML models by generating a set of concrete changes, and their impact on consistency rules. In [180] and [181], they focus on the cause of inconsistencies by analyzing consistency rules and their behavior during validation. They also present validation and repair in the form of linearly growing trees. The scalability of their approach is evaluated based on the application of UML models and OCL rules; the results indicate that validation and repair trees may be computed within a millisecond-range, which demonstrates the applicability of tree-based data structures for the problem at

hand. Although their approach is similar to the generation of basic repair rules, such as change actions that insert a reference, the approach presented in this chapter differs in the process of establishing model repair solutions that involve series of change actions.

Egyed et al. found that the performance of the brute force technique in generating repair solutions is not adequate and, hence, decided to add manual specifications in the form of value generation functions and (back)pointer specifications [66]. They follow the same motivation as the approach presented in this chapter for automating the generation of repairs for constraints that otherwise require the effort of language engineers to implement model repair mechanisms. The approach presented in this chapter, however, delivers language designers from the burden to handcraft model repair implementations, and instead automates the creation of value generation functions and (back)pointer references by adopting SBSE-techniques [96]. Further, manually implementing value generation functions as well as (back)pointer specifications may be infeasible due to the underlying problem of an infinite amount of possible repair solutions. Although providing handcrafted implementations, which realize the mechanism of model repair, may increase the effectiveness of a modeling language IDE, it may also (unintentionally) limit the delivery of favorable model repair solutions. Finally, the approach presented by Egyed et al. only supports the generation of model repair solutions which introduce new classes or attributes in models where a constraint violation occurs. As a consequence, model repair solutions that involve change actions in multiple locations cannot be generated.

Hegedüs et al. [100] present an approach to automate the generation of quick fix solutions for modeling languages by taking a set of constraints and model manipulation policies as input. Repair solutions are realized as a sequence of operations that are computed by employing state space exploration techniques targeting a decrease in the number of inconsistencies. Compared to the approach presented in this chapter, their approach also looks for local and global model repair solutions. It does not consider, however, the mechanism of content assistance or the validity of a model. Moreover, their approach employs graph patterns to capture inconsistency rules and graph transformation rules for change actions in model repair solutions, instead of textual formal constraints and SBSE-techniques.

Silva et al. [55] describe a method for the generation of model repair solutions (i.e. referred to as *repair plans* in their work) for inconsistent models using the configuration of the search space to antagonize the problem of the infinite amount of possible repair solutions. Compared to the approach presented in this chapter, their approach restricts the exploration of search space by limiting the generation of repair solutions overall, the applicability of models at hand, and the requirement to implement the mechanism of model validation manually (referred to as *inconsistency detection rules* in their work). Therefore, the quality of generated results may suffer, while language engineers are still responsible for handcrafting and maintaining inconsistency detection rules. In comparison, the approach presented in this chapter asynchronously searches for solutions during runtime of modeling language IDE execution and, thus, may deliver model repair solutions of higher quality compared to those generated within a more limited amount of

time.

Ali et al. [7] and Semeráth et al. [193] generate models which fulfill OCL invariants through optimization and reasoning respectively. In comparison to the approach presented in this chapter, however, their approaches only consider single-objective optimization. In other words, the generated test models must fulfill the complete set of OCL invariants and, thus, their approaches only consider the consistency of the entire model and not the set of change actions that may be employed to restore model consistency. Therefore, their work neglects the power of multi-objectiveness [58], which enables establishing model repair solutions for change actions that are based on multiple conflicting optimization objectives, such as validity and length of model repair solutions.

5.7.2 Language workbenches

For the purpose of evaluating the applicability of the approach, i.e. the specification of structural constraints as part of language definitions for the automated generation of enhanced modeling language IDEs, a set of language workbenches, including those listed by the Language Workbench Challenge 2016[5] and Microsoft Visual Studio [51] are used. In a nutshell, this analysis compares the capabilities of existing language workbenches. In particular, it examines their ability to define formal constraints in general and based on OCL, to establish constraint validation results manually and automatically, and to deliver content assistance suggestions as well as model repair solutions manually and automatically. Additionally, the compatibility of these language workbenches with the approach presented in this chapter is analyzed.

More specifically, the capabilities that are offered by language IDEs created by a particular language workbench are analyzed and include the following: first, the ability of a language workbench in regards to the definition of languages which include formal constraints, such as OCL invariants; second, the range of support that is provided by a language workbench to language designers for the specification of model conformance testing mechanisms, such as constraint-based model validation; third, the length of support provided by a language workbench to language designers for the formulation of model completion mechanisms, such as content-assist suggestions; fourth, the extent of a language workbench in regards to the design of model repair mechanisms, such as quick fix solutions; fifth, in case a language workbench enables the specification of formal constraints as part of a language definition, the degree to which language definitions are facilitated for the generation of IDEs that automatically instrument model validation, model completion, and model repair.

The subjects of study of a language workbench are represented by a set of artifacts and may include academic publications, direct communication with developers of a language workbench, and the documentation and implementation of a language workbench. The analysis also accounts for language workbenches that may not provide the support

[5] Language workbenches listed by the Language Workbench Challenge 2016 can be found online at http://2016.splashcon.org/track/lwc2016.

for structural constraints but still offer approaches for the implementation of model validation, content-assistance, and model repair solutions. Note that some workbenches, such as the "Onion" language workbench, have been omitted because of the lack of availability of artifacts such as publication, documentation, and implementation. Thus, what follows presents the differences and similarities of the approach with approaches on the specification of structural constraints in language workbenches as presented in the literature and outlined in Section 3.2. In particular, these include the following language workbenches: MetaEdit+ [195, 206], Intentional Software [194], Rascal [123], EMFText [101, 102], MontiCore [135], Spoofax [118], Xtext [70], Cedalion [146], JetBrains MPS [212], Eco [60], DrRacket [80], Melange [59], Ensō [209], SugarJ [68], Microsoft Visual Studio [51]. Epsilon [186], and Whole Platform [196].

MetaEdit+ The MetaEdit+ Workbench [195, 206] was last released in February 2017 and offers the expression of constraints as data integrated in the metamodel. Constraints, however, cannot be defined by the use of OCL. The website of the MetaEdit+ Workbench vendor metacase claims that neglecting the use of OCL prevents performance issues. Moreover, it is claimed that the need for constraints in modeling languages is small and may, therefore, be better expressed by the parameterization of constraint templates[6]. Although the MetaEdit+ Workbench offers the ability to specify mechanisms for constraint validation, content assist, and model repair, only constraint validation is available automatically. Thus, language designers employing the MetaEdit+ Workbench must manually establish implementations for content assist and model repair.

Intentional Software Intentional Software [194] represents a projectional language workbench approach describing a domain workbench tool that enables viewing and editing combinations of multiple domains. Domain schemas are described so as to define the terms of a domain that are processed by a generator to deliver executable target code. The Intentional Software approach does not provide source code, binaries, release notes, and documentation. Further, the authors did not respond to e-mail enquiries. Thus, it may not be claimed that Intentional Software supports the specification of formal constraints within language definitions.

Rascal Rascal [123] was last released in November 2017[7] and has no built-in mechanism or structure to define constraints. Moreover, Rascal does not offer a built-in constraint solver, and requires language designers to define custom type constraints and manually implement behavior to extract constraint specifications and solve matching constraints. Similarly, model repair mechanisms may be handcrafted by language designers. Listings four, five, and six in [123] illustrate examples of language restrictions, extraction algorithms, and constraint solving algorithms respectively. By disregarding the ability

[6] Further information is available online at the MetaEdit+ frequently asked questions page at https://www.metacase.com/faq/showfaq.asp?cate=MWB#MWBDoes_MetaEdit+_use_O.

[7] The Rascal Eclipse Update Site is located at https://update.rascal-mpl.org/stable/.

to provide means to restrict language definitions through formal constraint specifications, Rascal is unable to facilitate language definitions for the automated generation of implementations for the mechanisms of model validation, content assist, and model repair.

EMFText EMFText [101, 102] was last released in May 2013[8] and offers a feature metamodel that allows to specify constraints in terms of its language and expression. Although the application of modeling language definitions based on metamodels and OCL invariants is explored for generic static code analysis in languages of arbitrary domains, the facilitation of such language definition for the automated generation of model validation, content assist, and model repair functionality is neglected.

MontiCore MontiCore [135] represents a framework for the compositional development of textual DSLs and associated tool support which was last updated in March 2019[9]. Although languages in MontiCore are defined by means of EBNF-like MontiCore grammars, concepts, such as associations and inheritance, that are known from metamodeling and facilitated by model-first approaches (cf. Section **??**) are employed for the construction of arbitrary graph structures. Butting et al. [39] present an approach for the translation of MontiCore grammars to restricted Ecore metamodels. Krahn et al. [135] argue that metamodels generated from grammars, such as Ecore metamodels that are derived from Xtext grammars by facilitating the grammar-first approach of modeling language construction (cf. Section **??**), support a larger set of (valid) instances than those supported by (corresponding) grammars. In other words, metamodels that are derived from MontiCore grammars are less restrictive and, therefore, may indicate that the expressiveness of MontiCore grammars is greater than that of Ecore metamodels restricted by OCL invariants. Consequently, additional Java solvers are required to be handcrafted by language engineers in order to capture the intricate and implicit cardinalities of MontiCore grammars. Moreover, MontiCore grammars may define non-terminals by embedding languages that are defined by means of externally available MontiCore grammars. Similarly, handcrafted implementations must be provided to ensure that the requirements of embedded languages are fulfilled. In comparison to the approach presented in Chapters 4 and-5, metamodels with OCL invariants are generated from XML Schema restrictions, as well as being employed for the generation of consistency-achieving IDEs (as opposed to the generation of constrained metamodels from MontiCore grammars). Although the authors reference the contribution described in Chapter 4 that has resulted in the XMLText framework [162], they do not mention the contributions that extend and merge the latter framework, manifested in IntellEdit [164] and XMLIntellEdit [165] respectively. In conclusion, MontiCore does not offer native means to define languages with formal constraints. Although Butting et al. [39] present an

[8] The EMFText Eclipse Update Site is located at `http://update.emftext.org/release/`.

[9] The MontiCore distribution is available as archive file at `http://www.monticore.de/gettingstarted/monticore-cli-5.0.1.zip`.

approach which enables the generation of Ecore metamodels and OCL invariants from MontiCore grammars, constraints are evaluated based on handcrafted Java solvers.

Spoofax Spoofax [118] is a platform for the design of textual DSLs in an IDE that provides dedicated and declarative metalanguages for the definition of syntax, name binding, type analysis, program transformation, and code generation. It was last released in June 2019[10]. Although Spoofax does not support OCL, constraint generation rules that are directed to the abstract syntax of a language implementation may be specified. In particular, the program transformation language Stratego is employed to define term rewrite rules in the form "r : t1 –> t2 where s". Thus, both manual and automated constraint evaluation is available in Spoofax. Further, language engineers may provide content assist by means of handcrafted implementations.

Xtext Xtext [70] was last released in September 2019[11]. Xtext may be employed in combination with Xpand, a Java-like statically typed template language that is published alongside similar tools in the Eclipse Model to Text project[12]. Xpand provides a language for checking constraints that is referred to as Check Language[13] and represents a Java dialect to restrict Xtext-based languages. The Check Language is a declarative constraint language that is similar to OCL, and employs the expression syntax of Xpand. Moreover, a check component may define guard conditions to restrict the application of a check constraint to model elements that satisfy those guard conditions. The Xtext framework enables language designers to provide implementations for model validation, content assistance, and model repair. Although automated model validation in Xtext provides coarse results, as discussed in this chapter and in particular Section 5.3, it is fully automated. Content assistance and model repair, however, are only partially automated. The content-assist implementation that is provided by default employs the *SimpleNameProvider* that is based on looking up the name of an *EAttribute* and concatenating the qualified name of its parent exported *EObject* [22]. The default behavior of content assistance for cross-references is similar. Therefore, model repair is only partially automated and, more specifically, it only provides repair actions for refactoring reference names. It relies on language engineers to provide implementations for more complex model repair resolution.

[10] The Spoofax Language Workbench release notes are available online at `http://www.metaborg.org/en/latest/source/release/stable.html`.

[11] The Xtext Language Workbench release notes are available online at `https://www.eclipse.org/Xtext/releasenotes.html`.

[12] Xpand is part of the Eclipse Model to Text project and available online at `https://www.eclipse.org/modeling/m2t/?project=xpand`.

[13] Further details on the Check language are depicted in the Xpand documentation available online at `https://help.eclipse.org/photon/index.jsp?topic=/org.eclipse.xpand.doc/help/Check_language.html`.

Cedalion Cedalion [146] was last released in June 2012[14] and is an approach for the definition and application of internal DSLs, i.e. DSLs that are implemented as libraries for a given host language, based on projectional editing and static validation. Therefore, manual implementation for both types and checkers in dedicated Cedalion code is necessary in order to evaluate whether an instance conforms to a specified type.

JetBrains MPS JetBrains Meta Programming System (MPS) [212] is a projectional language workbench that was last released in March 2020[15] and enables the creation of languages and IDEs that can be used to edit hierarchical or marked up text. In contrast, text-based IDEs enable the editing of any document in the form of plain text files. JetBrains MPS employs its own dedicated component (referred to as Constraints aspect[16]) for the specification of structural constraints and the expression of constraints that are not represented by the MPS Structural Language. More specifically, it enables the restriction of relationships between nodes and allowed property values. Additionally, the JetBrains MPS workbench offers an action language for the definition of change actions (i.e. referred to as transformation actions) that are employed to realize the mechanism of content assist. Although MPS offers automated validation and content assistance for constraints specified in a dedicated constraint language, engineers must handcraft the mechanism of model repair by resorting to a GPL. Similar to the Spoofax language workbench, the specification of structural constraints in MPS is limited by the expressiveness of its dedicated constraint language and, thus, deviates from the expressiveness offered by mature formal constraint specification languages such as OCL. For example, the specification of iterators in structural constraints, i.e. available in OCL, is not provided. Hence, MPS and Spoofax may only be partially able to benefit from the advantages that are provided by the approach presented in this chapter.

Eco Eco [60] is a language composition editor that enables the use of multiple programming languages in a single file. It was last released in September 2018[17]. The core of this approach is represented by the extension of an incremental parser that allows the nesting of languages by employing the notion of *language boxes*, in that each box has a dedicated incremental parser that maintains its own parse tree. Eco implements a subset of the Name Binding Language approach [129], which defines a declarative language for specifying scoping rules and to avoid variables from different methods to "bleed" into each other. The result is that only names visible to a given scope are shown as content-assist suggestions. In conclusion, Eco does not support OCL or formal constraints, but instead

[14] The Cedalion project files are available online at https://sourceforge.net/projects/cedalion/files/.

[15] A link to the JetBrains MPS release notes is available at https://www.jetbrains.com/mps/download/.

[16] Constraints in JetBrains MPS User Guide are available online at https://www.jetbrains.com/help/mps/constraints.html.

[17] The Eco language workbench is available online for download at https://soft-dev.org/src/eco/.

employs simple scoping mechanisms for coping with individual contexts that may be represented by multi-lingual instances.

DrRacket The DrRacket language workbench [80] offers the Racket language and was last released in November 2019[18]. Racket is a programming language for the creation of new programming languages. A Racket programmer may annotate an existing module with explicit types and expected type soundness. Language invariants are not automatically protected by Racket. Language engineers of Racket-based languages build on Miller's proxy mechanism and, thus, may create customized proxies that monitor values to guarantee basic invariants. More specifically, function access, immutable values, and mutable structures of objects may be monitored. In conclusion, Racket does not support OCL, and language designers may create customized types with basic invariants, as well as handcraft proxy monitor implementations to enforce the consistency of invariants.

Melange The Melange language workbench [59] was last released in July 2017[19]. Melange allows to supplement static semantic rules expressed as OCL constraints in Ecore metamodels. Thus, Melange relies on the EMF compiler, which is based on a generator model that enables the generation of Java code from an associated Ecore metamodel, and the Xtend compiler, which generates Java code from a Melange aspects file. Moreover, Melange is operable within the EMF ecosystem and can be employed with other EMF ecosystem tools, such as Xtext or Sirius. The Melange approach focuses on the building of DSLs by safely assembling and customizing legacy DSL artifacts. Although Melange supports the process of DSL creation by employing OCL constraints in Ecore metamodels, it does not offer automated generation of IDE features, such as model validation, content assist, and model repair.

Ensō The Ensō language workbench [209] was last updated in July 2019[20]. Ensō is an external language workbench with graphical and textual editing capabilities. Ensō-based languages are defined by a schema or the model of its internal representation that may be rendered textually or graphically. The mapping between text and object graphs may be further controlled using predicates, i.e. constraint expressions on fields of objects in the object graph. During parsing, the values of these fields are updated to ensure that these constraints are fulfilled. In conclusion, Ensō does not support OCL. It employs predicates, however, in order to control the mapping between text and object graphs.

SugarJ SugarJ [68] was last updated in April 2016[21]. SugarJ offers encoding constraint specifications as part of rules (referred to as desugaring rules) that define and transform

[18] The Racket download page is located at `https://download.racket-lang.org`.

[19] The Melange language workbench Eclipse Update Site is located at `http://melange.inria.fr/updatesite/releases/`.

[20] The Enso language workbench Github project is located at `https://github.com/enso-lang/enso`.

[21] The SugarJ Github project is located at `https://github.com/sugar-lang/main`.

extended syntax into host language syntax. Restrictions in SugarJ may be defined using regular expressions in desugaring rules. By neglecting the support of model repair mechanisms, however, and due to the limited expressiveness of regular expressions based on a dedicated language, the approach presented in this chapter may not be applicable in SugarJ. The applicability of a constraint system that separates the generation from the verification of constraints may allow their interleaved execution with desugaring, as opposed to interfering with the application of desugaring, and may form the focus of future work. In conclusion, SugarJ does not support OCL. Although a dedicated regular expression language is provided to enable constraint specification within desugaring rules, no automated validation, content assist, and model repair mechansism are available.

Visual Studio Microsoft Visual Studio [51] offers no support for formal constraints. Validators and rules that change model elements depending on other model elements may be implemented manually to define model validation, content-assistance, and limited model repair. The approach presented in this chapter may be applicable in Visual Studio if the requirement of support for EMF/OCL is fulfilled.

Epsilon The Epsilon platform [186] was last released in September 2018[22]. The Epsilon language family includes the Epsilon Validation Language (EVL), an OCL-like validation language that supports dependencies between constraints displayed to the user and the specification of fixes in the Epsilon Object Language (EOL), which may be invoked to repair inconsistencies. Epsilon is integrated with the EMF validation framework and GMF. EVL supports OCL-like first-order logic operations, such as select, reject, and collect. Work on EVL has identified some shortcomings of OCL and, in particular, poor support for user feedback, no support for warnings, no support for dependent constraints, limited flexibility in context definition, and no support for the repair of constraint violations [126]. Language designers may employ EVL for the definition of semi-automatic model repair mechanisms alongside invariants. Hence, compared to the approach presented in this chapter, Epsilon does not automatically generate model repair solutions for violated constraints. In conclusion, Epsilon offers EVL to enable the manual specification of model repair functionality. Model repair solutions, however, are not automatically generated. Moreover, as a result of EVL constraint specifications not being defined as part of metamodels, the approach presented in this chapter may not be readily applicable in Epsilon. The transformation of OCL constraints in metamodels to EVL specifications, however, as well as the implementation of model repair solutions based on EVL may ultimately enable the applicability of the approach presented in this chapter within the Epsilon platform.

[22]The Epsilon stable Update Site is located at `http://download.eclipse.org/epsilon/updates/`.

Whole Platform The Whole Platform [196] was last released in 2014 and updated in November 2019[23]. The Whole Platform is an Eclipse-based language workbench that employs generative model driven technology for the design and implementation of new languages and tools. In summary, the Whole Platform does not support OCL or, according to its technical report, textual formal constraints. Thus, the support for language constraints, model validation, content assist, and model repair must be handcrafted by language engineers.

Language workbench	Formal constraint definition	OCL	Manual constraint validation	Automated constraint validation	Manual content assist	Automated content assist	Manual model repair	Automated model repair
MetaEdit+	●	○	●	●	●	○	●	○
Rascal	○	○	●	○	N/A[24]	○	●	○
MontiCore	○	○	●	○	○	○	○	○
Spoofax	◐	○	●	●	●	○	○	○
Xtext	◐	◐	○	●	●	◐	●	◐
Jetbrains MPS	◐	○	●	●	●	●	●	○
Eco	○	○	●	○	●	○	○	○
DrRacket	●	○	●	●	○	○	●	○
Melange	◐	◐	●	○	○	○	○	○
Ensō	○	○	◐	◐	○	○	○	○
SugarJ	◐	○	●	◐	●	◐	○	○
VisualStudio	○	○	●	○	●	○	○	○
Epsilon	●	●	●	○	○	○	●	○
Whole Platform	○	○	○	○	○	○	○	○

Table 5.4: Structural constraints in language workbenches (●= full support; ◐= partial support; ○= no support).

In summary, the results of this analysis are depicted in Table 5.4 and indicate that none of the analyzed workbenches offer both the capability to extend language definitions with formal constraint specifications, such as OCL invariants, or the application of formal constraints for the automated generation of language IDEs with built-in validation, content assistance, and model repair mechanisms. Moreover, the results of this analysis also indicate that the restriction of languages with structural constraints is manifested in the analyzed workbenches either by the application of workbench-specific languages, GPL code, or a combination thereof. Note that, although the results of this analysis include direct communication with tool developers, a comparison that involves the employment of individual language workbenches for the development of comprehensive language implementations may produce deviating results. Further, note that language

[23]The Whole Platform Github release page is located at `https://github.com/wholeplatform/whole/releases`.

[24]Not enough information for an evaluation of manual scoping in Rascal could be retrieved.

workbenches that have not been updated for three years prior to November 2019, or for which crucial information, such as documentation and implementation, has not been made available, do not appear in Table 5.4. In particular, these include Intentional Software [194], EMFText [101, 102], and Cedalion [146].

Full support for structural constraints is provided in Epsilon [186], DrRacket [80], and MetaEdit+ [195, 206]. Epsilon, however, does not represent an actual language workbench, which may be used to generate a modeling language, but a family of languages, including the Epsilon Validation Language (EVL), which is based on similar concepts as OCL. Hence, EVL may be used to formulate constraints that may be used in conjunction with other EMF tools, such as Xtext, for the creation of modeling languages. More specifically, EVL may be used to augment and evaluate structural constraints in Ecore models. Therefore, the approach presented in this chapter may be extended to support the application of EVL, and benefit from its advantages, such as support for dependent constraints and enhanced flexibility in context definition [126]. DrRacket offers specification of structural restrictions and automated validation of contracts. Further, DrRacket allows language engineers to handcraft functions for the visualization of error locations and the recovery of contract validity. MetaEdit+ employs a metamodeling language for the specification of graphs, objects, properties, ports, roles, and relationships, and a scripting language for the specification and automated validation of structural constraints. Moreover, MetaEdit+ allows language engineers to implement validation, content assistance, and model repair manually. MetaEdit+, however, represents a non-generative approach and, thus, facilitates both language definition and application within the same tool instance. As such, MetaEdit+ may only allow the non-generative part of the approach presented in this chapter.

Xtext [70], JetBrains MPS [212], Melange [59], Spoofax [118], and SugarJ [68] provide partial support for structural constraints. Both Xtext and Melange build on EMF by means of language specifications in the form of Ecore metamodels that may be augmented with OCL invariants and, thus, reflect the same means as the approach presented in this chapter. Although Melange does not engage in the generation of modeling language implementations, it may do so in conjunction with EMF-based tools, such as Xtext or Sirius [211]. Hence, the capabilities of modeling languages that are specified by employing Melange are limited by the EMF-based tool that is applied to generate language implementations. Additionally, Melange provides a dedicated metalanguage for the specification of model types that are based on the definition of groups of related types, i.e. a set of constraints over admissible model graphs. Both Xtext and Melange employ the default OCL interpreter that is provided by EMF, offering limited automated validation, such as the display of error messages as illustrated in Figure 5.13. Moreover, Xtext provides facilities for the manual specification of content-assist and model repair behavior and offers automated generation of limited model validation and basic content assistance for modeling languages.

5.8 Summary

This chapter presented an approach for leveraging formal language definitions with constraint specifications for the automated generation of modeling language implementations with enhanced IDEs. The requirements imposed for the generation of solutions for content assistance and model repair include the preservation of model consistency for the former and the improvement or recovery of model consistency for the latter. In addition, rather than highlighting entire sub-structures of models as erroneous and neglecting structures which impact constraint violations, IntellEdit computes and visualizes consistency violations in a fine-grained manner to IDE users, thus enabling precise localization and eventual repair. Further, generated solutions are sorted according to quality and cost, ranked, and visualized to IDE users. As a result, a custom search approach based on SBSE-techniques has been developed, which proposes consistency-maintaining and consistency-restoring change actions that are manifested as content-assist suggestions and model repair solutions respectively. Therefore, challenges imposed by the requirements of the presented approach and, in particular, the delivery of change actions that do not introduce constraint violations, such as limitations of SBSE implementations, have been addressed. More specifically, challenges include the procurement of neighborhood populations, the escape of local optima solutions, and the computation of optimal solutions that are based on multi-objective search, thus enabling proposals that consist of simultaneous change actions on multiple model features.

In conclusion, the application of the approach presented in this chapter enables language designers to do away with the handcrafting of implementations of modeling language IDEs by facilitating the automated generation of modeling language IDEs from language definitions composed of metamodels and formal constraints. More specifically, the presented approach enables the automated generation of modeling language IDEs with precise validation, consistency-preserving content assistance, and consistency-restoring model repair, all of which are capabilities far beyond those offered by state-of-the-art language workbenches. Overall, the evaluation of the approach indicates a two-fold improvement in validation precision over existing solutions, content assistance that retains model validity, and model repair solutions that reduce the number of violated constraints by approximately 67%.

CHAPTER 6

Reusable notation-template language and design framework

The previous chapter introduced an approach and tool for automating the generation of enhanced modeling language implementations with IDEs that feature precise validation, consistency-preserving content assist, and consistency-recovering model repair. This chapter presents an approach composed of a language and framework for the design and application of reusable modeling language notation templates that are metamodel-agnostic (i.e. structure-independent), metamodel-dependent, or a combination thereof.

Domain-specific languages enable concise and precise formalization of domain concepts and promote direct employment by domain experts. Therefore, syntactic constructs are introduced to empower users to associate concepts and relationships with visual textual symbols. Model-based language engineering facilitates the description of concepts and relationships in an abstract manner. Concrete representations, however, are commonly attached to abstract domain representations, such as annotations in metamodels, or directly encoded into language grammar. Thus, they introduce redundancy between metamodel elements and grammar elements. Against this background, an approach is proposed that enables autonomous development and maintenance of domain concepts and textual language notations in a distinctive and metamodel-agnostic manner by employing notation-specification models containing grammar rule templates and injection-based property selection. The implementation of the proposed notation-specification language is showcased in a comparison with state-of-the-art practices during the creation of notations for modeling language implementations that are based on the Eclipse Modeling Framework and Xtext. Figure 6.1 recaptures the contributions of this thesis and highlights the contribution presented in this chapter.

The remainder of this chapter is organized as follows. Section 6.1 introduces the approach and its motivations. Section 6.2 provides a brief overview of the methodologies

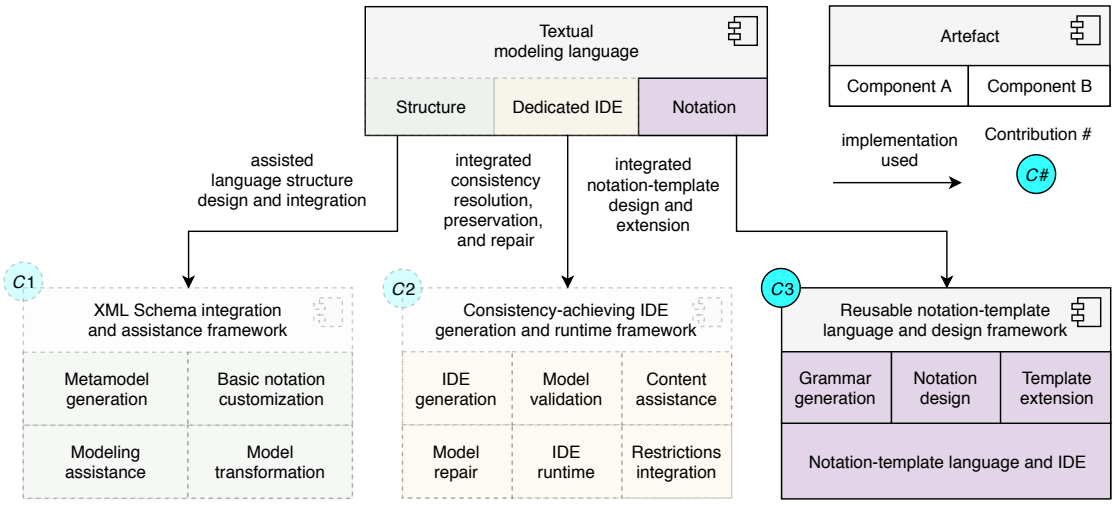

Figure 6.1: Contribution presented in this chapter.

and techniques upon which the approach is built. Section 6.3 and Section 6.4 present the challenges and requirements of the approach presented in this chapter respectively. Section 6.5 conceptually aligns the approach alongside the model-first and grammar-first approaches, and presents the language and framework for the design of reusable and extensible notation templates [163]. Section 6.6 presents the evaluation of the approach on a case study based on usability indicators of Domain-Specific Languages (DSLs) [105, 6]. More specifically, a set of popular open-source modeling languages are retrieved from Github and employed to evaluate the integratability, expressiveness, and conciseness of the approach and, in particular, the notation-template language in comparison with handcrafted grammar and grammar produced by a state-of-the-art metamodel-to-grammar generator. Section 6.7 analyzes and compares the proposed approach with related work. Finally, Section 6.8 concludes the chapter by summarizing the presented work.

6.1 Introduction

The engineering of a modeling language (cf. Section 2.4.2) is typically initiated by the construction of an artifact that captures concepts and relationships inherent to the domain being represented. Typical artifact types include variations of metamodels and grammars—each, of inherently different nature [122]. Metamodels are commonly used to capture concepts and relationships of a domain in the form of structural features such as classes, attributes, and relationships. Grammars are employed to capture domain concepts and relationships, as well as their visual representation by utilizing production rules and terminal rules.

The design of a notation of a language is in fact the design of the user interface of the given language. Therefore, the syntax of a language plays an important role

in user acceptance and is, therefore, regarded as a crucial factor for the success of a language [198]. For example, multiple different notations facilitate the integration of various types of stakeholders and support numerous use cases by targeting particular aspects of a domain [69]. The design of effective user interfaces, however, is an iterative process by nature, as it requires the evaluation of individual alternatives involving the testing of alternative notations in real-world conditions [166].

The classical approach to language development presumes that the syntax of a language is constructed upfront and results in the complete specification of language grammar that is augmented with semantic actions (i.e. functions that receive a list of values from subordinate nodes and produce a value) and employed to generate a parser for the given language [3]. Thus, modifying the syntax of a language typically requires adaptation of semantic rules, which makes the reshaping of the notation of a language a strenuous process. For example, modifying the syntax of a language to support the combination of flexible declaration sequences and whitespace semantics requires to state every possibly occurring sequence explicitly and causes the size of production rules to multiply by the number of structural features occurring in the containing classes.

The separation of syntax and the structural features of a domain arises from the motivation to increase notational flexibility by enabling the accommodation of various notations, such as those observed in programming languages as well as other domains. As a result, acknowledging the individual preferences of domain experts (shaped by their respective backgrounds), leads to the establishment of notations that closely resemble representations familiar to stakeholders, thus, driving greater user satisfaction.

Although state-of-the-art language workbenches, such as Xtext [70], provide means to generate grammars from metamodels and vice-versa, they provide a single (default) notation, i.e. either graphical, textual, or a combination thereof, that has to fit the needs of every type of stakeholder, such as domain experts, or that requires dedicated language engineering skills for adaptation and extension. The construction of a bridge between metamodel and grammar, and in particular from metamodel to grammar, is commonly approached by the introduction of annotations in metamodels or the construction of metamodel-to-grammar transformations [5]. The construction and maintenance of such bridges, however, is inherently complex and error-prone due to fundamental differences between metamodels and grammars. Moreover, such bridges are often metamodel-dependent (i.e. depend on a particular domain-specific metamodel) and are, thus, not universally applicable to arbitrary domains.

In this chapter, the Ecore Concrete Syntax Specification (ECSS) framework—a novel textual notation description language and toolkit that enables the definition of both metamodel-dependent and metamodel-agnostic representations for EMF-based modeling languages—is presented and employed for the adoption of diverse language variants by automating the generation of textual modeling language implementations with supporting editors and tools from Ecore metamodels. In summary, ECSS facilitates the creation, extension, and reuse of textual notations (henceforth referred to as “style models”, “ECSS models”, “notation-specifications”, and “notation-templates”), as well

as the generation of grammar and executable implementation of modeling languages from pairs that consist of domain metamodels and style models.

6.2 Background

Model-Driven Engineering and Domain-Specific Languages. The work presented in this chapter specifically focuses on the construction and maintenance of textual modeling languages or textual Domain-Specific Modeling Languages (DSMLs), i.e. the employment of Model-Driven Engineering (MDE) in the context of DSLs [85], by constructing an approach on top of the Xtext language workbench that is built on the EMF [70, 197]. More specifically, EMF is the quasi-reference implementation of the Essential Meta Object Facility (EMOF) standard [168] and provides a closed and strict metamodeling architecture, which defines the model on the uppermost layer to conform to itself, as well as the correspondence of every model element with a model element of the layer above respectively. EMOF, as well as the Extended Backus–Naur Form (EBNF) [219], represent DSLs to define languages in the form of metamodels and Context-Free Grammars (CFGs), i.e. also referred to as "text-based concrete syntaxes" and "notations" respectively. In the EMF, an Ecore model—also referred to as EMF-based "metamodel" or "abstract syntax"—corresponds to the M2-layer in EMOF (cf. Figure 2.1) and acts as an abstract representation of the concepts, properties, and relationships that are embodied in a real-world system. Further, the M1-layer in EMOF represents instances that specify actual values for concepts, properties, and relationships as defined in their corresponding M2-layer Ecore model.

Language Engineering and Workbenches. Language workbenches [84], such as Xtext, are tools that provide a range of features, such as dedicated editors, model transformations and validations, for modeling language specifications. In general, Xtext employs the ANTLR parser generator [174] for the production of implementation artifacts, such as lexers and parsers, and offers two different language-construction mechanisms, i.e. typically selected as a result of an engineer's familiarity with the technical spaces[1] of grammarware and modelware [218]. On the one hand, grammarware engineers (i.e. experienced with traditional CFGs) may construct CFGs and employ the Xtext mechanism for deriving EMF-based metamodels. On the other hand, modelware engineers (i.e. skilled with the application of MDE-based technologies), may develop EMF-based metamodels and derive CFGs by employing metamodel-to-grammar transformations. Although Xtext supports both, the main focus is to facilitate tool interoperability by providing grammars at the front-end and metamodels at the back-end [171, 223].

[1] *Technical space* refers to a working context with a set of associated concepts, body of knowledge, tools, required skills, and possibilities often associated with a given user community with shared know-how, educational support, common literature, and scientific venues [139].

6.3 Challenges

Despite the fact that state-of-the-art modeling language workbench frameworks facilitate language development, for example by automating the generation of complex parser implementations, they still restrict the development and maintenance of production-ready modeling language implementations to engineers that are trained in the construction of unambiguous EBNF-based grammars. For example, the Xtext framework proposes a model-first approach that builds on a single notation that must either fit the needs of different types of stakeholders or that requires modelware engineers to perform dedicated language engineering tasks, such as customization and extension of model transformations, grammar generators, or grammars themselves. More specifically, model transformations or grammar generators, such as the state-of-the-art grammar generator of the Xtext framework [70], produce grammars from domain-specific structural specifications such as Ecore metamodels. Therefore, the development of modeling language implementations, and more specifically the customization of the visual textual language notations, is challenged by the language engineering skills of individual modelware engineers and, in particular, their ability to ingrain notational information into language grammars effectively.

Moreover, the implementation of a modeling language workbench framework steers and constraints the method of creating and maintaining language implementations and, in particular, language grammars. Thus, as opposed to focusing on language design, such as the capturing of domain-specific concepts, language developers must handle the peculiarities of maintaining customized, unambiguous grammars. For example, performing changes in the structure of a domain, such as the addition, removal, or editing of structural features that are represented by metamodel classes and attributes, must be followed by the re-generation of grammar and, thus, the re-execution of any manual steps that led to the final state of the previous version of the grammar.

Further, use cases which employ hand-crafted bridges between metamodels and grammars challenge the maintenance of modeling language implementations as a result of the inherently fundamental differences between metamodels and grammars and, in particular, the complexity and structural dependence of mappings between metamodels and grammars. Although tools, such as model transformations or grammar generator implementations, are available for the construction of such mappings, they rarely manifest as metamodel-agnostic bridges and are, thus, not universally applicable [5]. In other words, these bridges are typically bound to either a specific metamodel (i.e. designed to be applied to a specific set of domain-specific concepts), or a specific notation (i.e. designed to yield the same representation for all domain-specific concepts), or both (i.e. designed for a set of domain-specific concepts and the production of a dedicated representation thereof).

6.4 Requirements

The requirements to establish an approach enabling reusable notation templates for the design of textual modeling languages include the following.

The approach must be applicable within the context of model-first language development and, thus, support the manifestation and maintenance of modeling language implementations that are initiated by language developers and, in particular, modelware engineers, through the materialization of structural features in domain-specific metamodels (i.e. acting as representations of domain-specific concepts and relationships).

Moreover, as opposed to combining the specification of concepts and notations in a singular artifact, such as through the annotation of components in domain-specific metamodels, the approach must separate language structure from language representation to outline the fundamental premise of notation-reusability. Additionally, notation-templates must be extendable and, therefore, enable the inheritance of specifications of representation that are defined by (pre)existing notation-templates.

Further, the notation-template language must enable the design of textual representations that are metamodel-dependent, metamodel-agnostic, or a combination thereof. Therefore, the notation-template language must facilitate notation-templates for a set of domain-specific concepts and relationships (i.e. represented by a set of structural features in an Ecore-based domain-specific metamodel), a domain-agnostic set of concepts and relationships (i.e. represented by a set of structural features in individual domain-specific metamodels[2]), and a set that combines domain-specific and domain-agnostic concepts and relationships (i.e. represented by a set of structural features introduced by individual domain-specific metamodels and the metamodel of the Ecore language itself).

Next, the approach must enable the facilitation of pairs that consist of metamodels and notation-templates for the automated generation of grammars that act as artifacts with overlapping structural and representational specifications as defined by metamodels and notation-templates respectively.

Finally, the approach must provide a facility for the creation and maintenance of notation-templates through the contribution of a dedicated IDE for the creation and editing of notation-templates.

6.5 Approach

Within this section, the approach is outlined alongside its application to a domain-specific model created as a case study. First, a typical language engineering use case is introduced by illustrating the application of the state-of-the-art model-first and grammar-first approach on the structure, model, and notational requirements of the domain-specific

[2]Structural features that are depicted in the Ecore language metamodel, such as *EClass*, *EAttribute*, and *EReference*, are instantiated by Ecore-based domain-specific metamodels.

language for modeling space transportation services (i.e. introduced throughout Section 2). Second, the design principles, structural components, and selection modes of the notation-template language are introduced. Third, the notation-template language is employed to design a notation-template that fulfills the notational requirements of the space transportation service language, as outlined in Section 6.4. Finally, the mechanism that generates executable modeling language implementations from tuples that consist of notation-template models and domain-specific metamodels is presented, and the limitations of the contributed implementation are outlined.

6.5.1 Overview

Within this section, a typical language engineering use case [136, 121] is presented that involves the construction of a modeling language by employing Xtext and the EMF and, in particular, a metamodel for capturing the concepts and relationships of space transportation services language. Moreover, this metamodel formulates the foundation upon which model-first state-of-the-art practices (i.e. employing model-to-text transformations and manual grammar adaptation), as well as the approach presented in this chapter construct and modify grammars (i.e. embodying structural and representational information).

Language Structure. The metamodel of the exemplary language (cf. Figure 2.3) instantiates the core components of the Ecore language, such as (abstract) classes, attributes, (containment) references, and enumerations. More specifically, the following concepts and relationships are defined: a *SpaceTransportationService* may own launch sites, launch schedules, spacecrafts, and engine types; a *Spacecraft* is defined by name, relaunch-cost, stages, manufacturer, country of origin, physical properties, functions, such as being an orbital launcher or intercontinental transport vehicle, and launch sites for which it is certified for launch; a *Stage* is defined by name, engine type, and physical properties; a *PhysicalProperty* is defined by type, such as length, volume or mass, unit, and value; a *LaunchSite* is defined by name, location, operator, number of launchpads, operational status, and physical properties; a *LaunchSchedule* is defined by name and launch events; and a *LaunchEvent* is defined by mission title, start date and time, launch site, and spacecraft.

Example Model. Figure 6.2 illustrates a handcrafted graphical model of an example space transportation service that is composed of a spacecraft, a launch schedule, a launch site, and two types of engines. The depicted spacecraft named `Falcon Heavy` functions as `ORBITAL_LAUNCHER`, is manufactured by `SpaceY`, originates from the country `USA`, has a relaunch cost of `90` million USD, and has a set of physical properties describing its `LENGTH`, `WIDTH`, `DIAMETER`, and `MASS`. Further, the modeled spacecraft is composed of two stages and, in particular, a first stage with nine engines of type `Merlin 1D` and a secondary stage with a single engine of type `Merlin 1D Vacuum`. In addition, an operational launch site named `Kennedy Space Center` is defined with operator `NASA`,

three launchpads, and a location based on latitude and longitude. Finally, a launch schedule with two launch events is specified for the spacecraft and launch site described above.

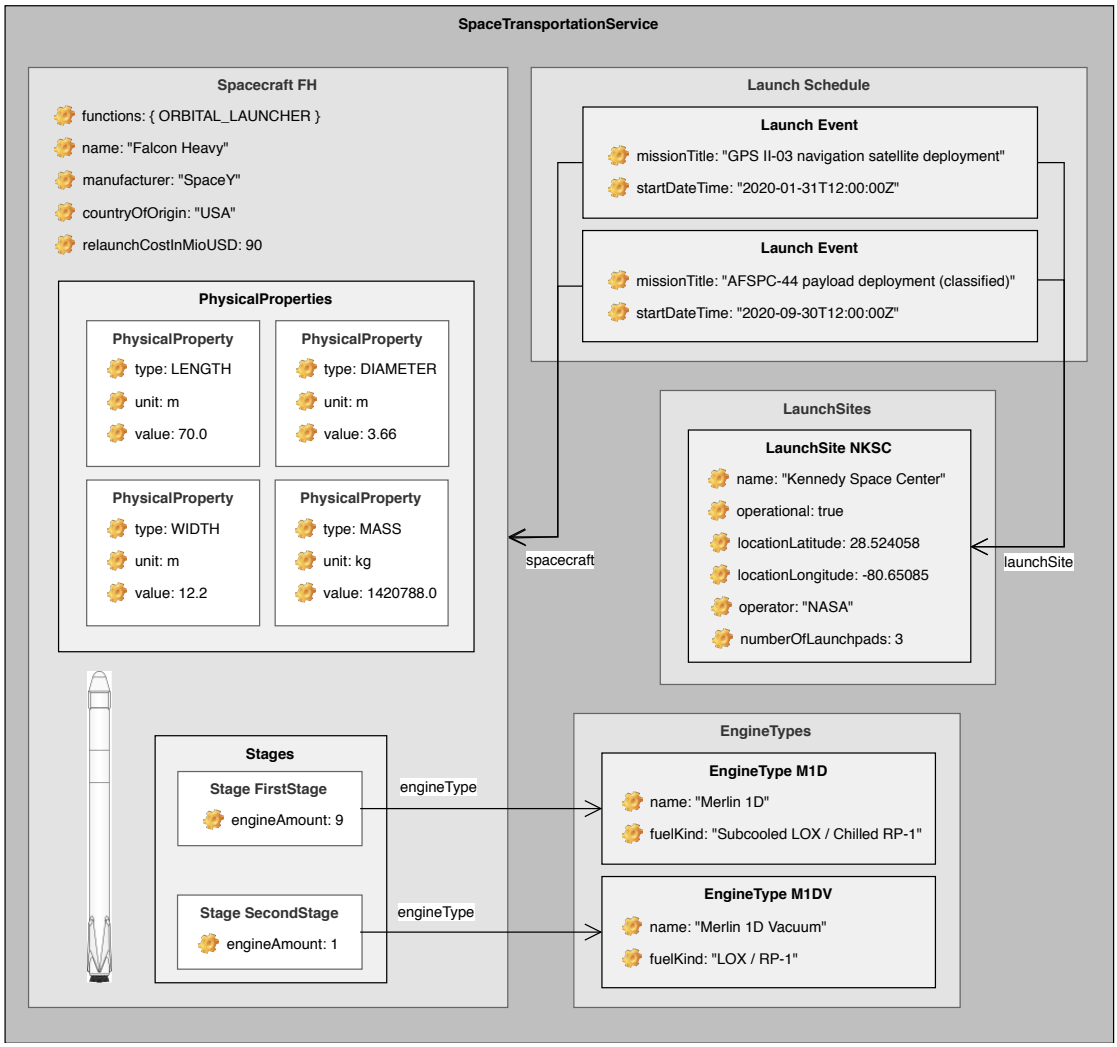

Figure 6.2: Visual graphical model of an example space transportation service.

Notational Requirements. The requirements of the textual notation of the language to be constructed include indentation-based layout (also referred to as offside-rules [142, 1], representing the determination of code block-structure by means of indentation and layout based on the concept of layout-sensitive languages [142, 1], such as Python, Haskell, CoffeeScript, and YAML Ain't Markup Language (YAML) [17]), and arbitrary order of declaration (i.e. flexible or unordered sequence of instantiation). Listing 6.1 presents an excerpt of the space transportation service model of Figure 6.2 (cf. complete model depicted in Appendix 6) visualized by means of a textual notation that fulfills both of the

requirements of indentation-based layout and arbitrary order of declaration. The former allows the use of hidden tokens, such as whitespace, as separators instead of visible tokens, such as curly brackets. The latter enables the use of alternative sequences (i.e. sequences that are differentiated from those defined in a domain-specific metamodel). For example, as opposed to restricting the occurrence of location coordinates to the beginning of a launch site definition, they may appear at the end.

```
SpaceTransportationService:
  launchSites:
    name: KennedySpaceCenter
    operator: NASA
    operational: true
    numberOfLaunchpads: 3
    locationLatitude: 28.524058
    locationLongitude: 80.65085
```

Listing 6.1: Instance of a space transportation service (excerpt).

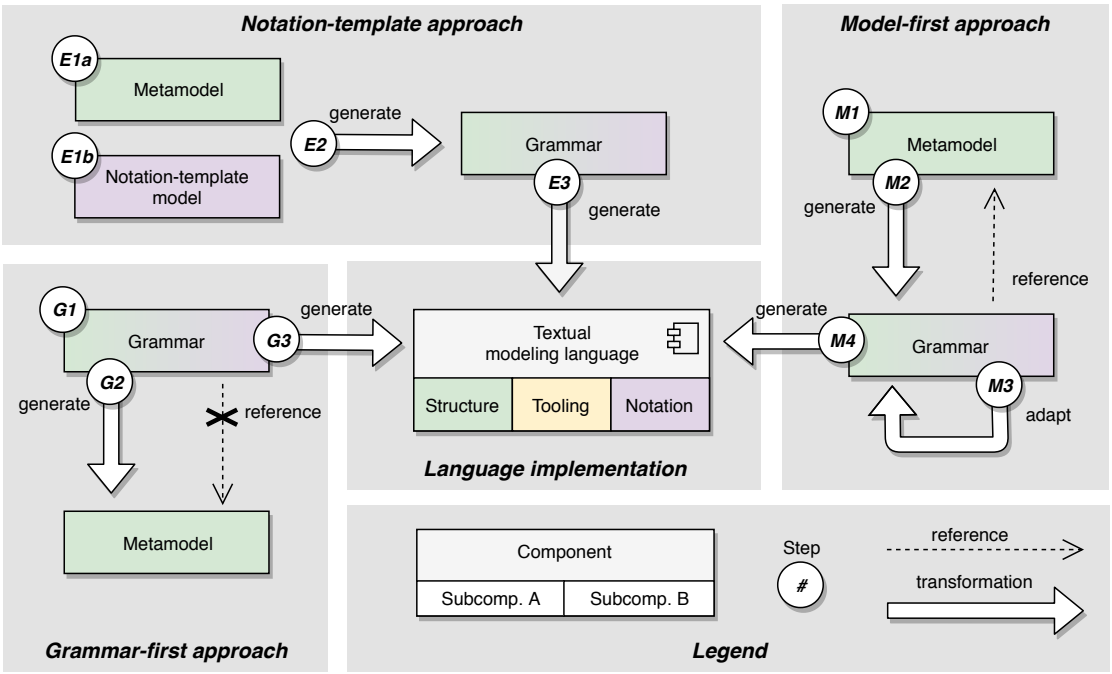

Figure 6.3: Overview of steps involved in the manifestation of language implementations as occurring in the approaches of model-first and grammar-first, and the approach presented in this chapter.

The *model-first* approach [170] of constructing a textual modeling language (cf. top-right gray rectangle in Figure 6.3) is typically applied by developers who are most familiar with MDE, and is composed of the following steps: $M1$, i.e. construction of a domain-specific metamodel (cf. result in Figure 2.3); $M2$, i.e. application of a generic metamodel-to-grammar transformation (cf. excerpt and complete version of result in

Listing 6.2 and Appendix 3 respectively); $M3$, i.e. manual adaptation of generated grammar to fulfill the notational requirements stated above (cf. excerpt and complete version of result in Listing 6.3 and Appendix 4 respectively); and $M4$, i.e. generation of language implementation (cf. center gray rectangle in Figure 6.3). Note that in the case steps $M2$ and $M3$ are replaced by the creation and application of a dedicated metamodel-to-grammar transformation (i.e. as opposed to the Xtext metamodel-to-grammar transformation [203]), which is typically highly coupled to a specific metamodel, requires the maintenance of both dedicated metamodel and metamodel-to-grammar transformation.

```
1 Stage returns Stage:
2  'Stage' name=ID '{'
3    'engineAmount' engineAmount=EInt
4    'engineType' engineType=[EngineType]
5    ('physicalProperties' '{'
6      physicalProperties+=PhysicalProperty
7      ( "," physicalProperties+=PhysicalProperty)*
8    '}' )?
9  '}';
```

Listing 6.2: Result of step $M2$—generated domain-specific grammar (excerpt).

In order to yield a textual modeling language that supports indentation-based layout (i.e. prescribing that every non-whitespace token of a structure must be further to the right than the token that initiated the structure), as well as a flexible order of specification, the following adaptations on the generated grammar are performed. First, synthetic tokens (i.e. offering the specification of whitespace-semantics employing synthetic terminal rules) are introduced for the beginning and end of a line (cf. lines 1–2 in Listing 6.3). Next, production rules are adapted to employ the specified synthetic tokens (cf. lines 6–11). Second, in order to support arbitrary order of declaration alongside whitespace-semantics, every possibly occurring sequence must be depicted by the grammar, which is accomplished by intermediating rule assignments with a vertical line (i.e. indicating a logical *or*) and enclosing them with brackets ending with a star-character (i.e. indicating zero or multiple occurrences). The combination of flexible sequences and whitespace-semantics, however, requires to state every possibly occurring sequence explicitly and, thus, causes the size of production rules to multiply by the number of structural features occurring in the containing classes. In Xtext, the ampersand character "`&`" may used to define an unordered group of two or more elements (i.e. indicating arbitrary order of occurrence). The result of extending Listing 6.3) with support for arbitrary order of declaration is depicted in Appendix 4.

```
1 terminal BEGIN: 'synthetic:BEGIN';
2 terminal END: 'synthetic:END';
3
4 Stage returns Stage:
5    name=ID ':'
6    BEGIN
7       (
```

```
8           ( 'engineAmount' ':' engineAmount=EInt )
9           ( 'engineType' ':' engineType=[EngineType|EString] )
10          ( 'physicalProperties' ':' BEGIN physicalProperties+=PhysicalProperty (
                physicalProperties+=PhysicalProperty)* END )?
11      )
12  END
13 ;
```

Listing 6.3: Result of step $M3$—manually adapted domain-specific grammar (excerpt).

The *grammar-first* approach of constructing a textual modeling language [170] (cf. bottom-left gray rectangle in Figure 6.3) is typically applied by developers most acquainted with grammar-based language engineering. It is composed of the construction of a domain-specific grammar (cf. step $G1$), the application of a generic grammar-to-metamodel transformation (cf. step $G2$), also referred to as metamodel-derivation, and the generation of the language implementation (cf. step $G3$). Although step $G2$ may be performed as a background process in Xtext, i.e. alongside step $G3$, possibly without the language developers' awareness, it represents an obligatory process in yielding the executable implementation of a textual modeling language (cf. center gray rectangle in Figure 6.3).

Model-first language engineering creates metamodels for capturing domain-specific structural semantics and constraints and, thus, represents the target application methodology of the approach presented in this chapter. The approach, however, is also applicable for use cases in which domain-specific metamodels are derived from grammars and employed alongside notation-template models to generate executable implementations of textual modeling languages which are based on grammars that follow the notation defined in respective notation-template models.

6.5.2 Notation-template language

This section presents the design principles, as well as the structural and syntactic components of the notation-template framework and language, and in particular its implementation as represented by the ECSS framework and ECSS language respectively.

The ECSS notation-template language is designed to capture common styles of textual syntaxes, such as YAML and JSON, and to support the automated generation of Xtext grammars that follow the rules of such syntaxes. An initial catalogue of reusable ECSS styles is available online at `http://bit.ly/ecss-styles`. ECSS is inspired by CSS, which allows to style HTML code and offers straightforward composition, thus aiming for similar composability and parameterizability. For example, developers must be able to reuse, extend, and adapt notation templates by composition of notation-template models.

The core structural component of the ECSS language (cf. Figure 6.4) is a *Model* that may extend existing instances of notation-template models through imports, and contains

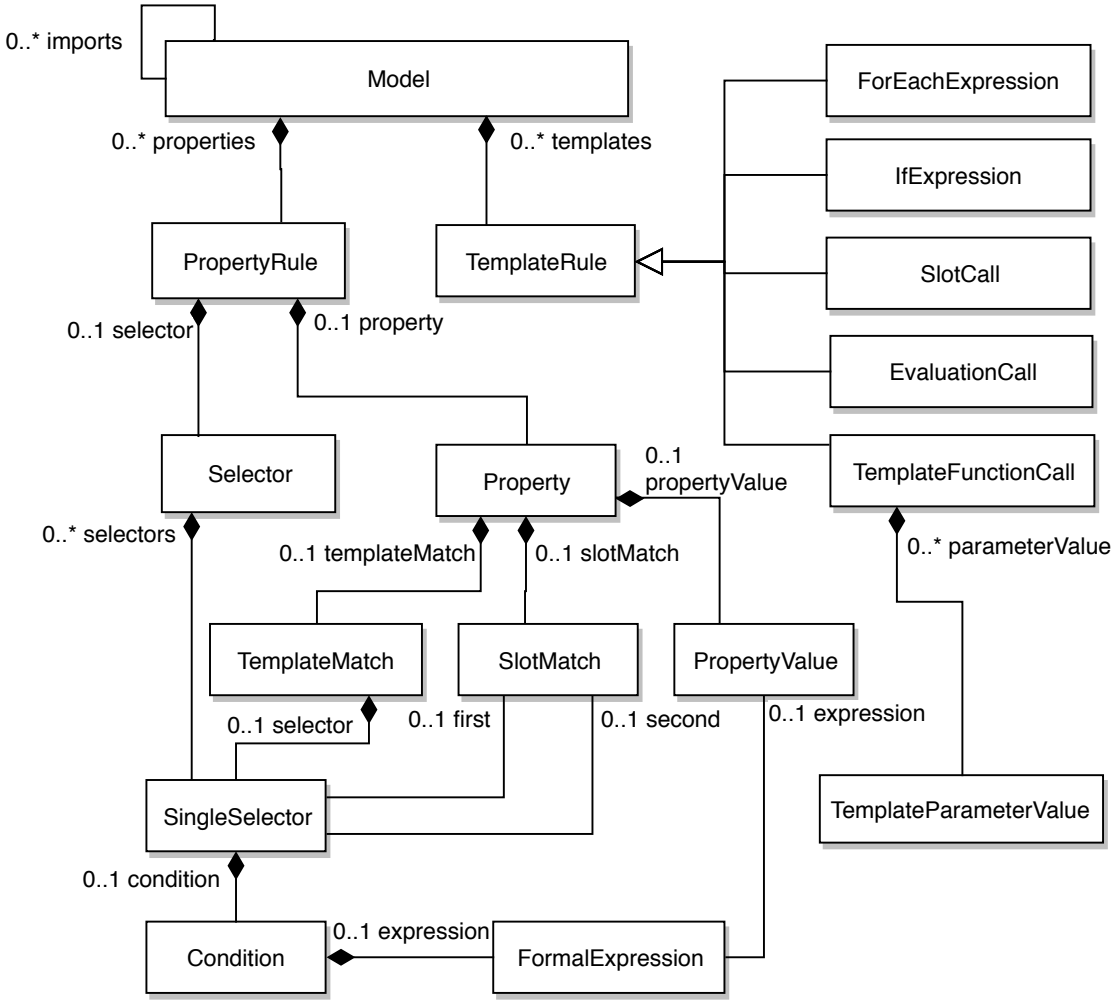

Figure 6.4: Structural components of the notation-template language (excerpt).

a set of rules that define properties and templates. *PropertyRules* may be composed of a selector and a property. *Selectors* may be composed of *SingleSelector* instances that may contain a *Condition* that is defined by a *FormalExpression*, such as an OCL expression. Each single selector selects a particular rule application instance. A *Selector* defines a *name* and, thereby, selects rule applications with matching names. More specifically, the name for a rule that has a single feature as a parameter refers to the name of the feature, i.e. the name of the class in the case of a single class. Additionally, a *Selector* may select subclasses by specifying the name of the rule that is named by its superclasses. Similarly to HTML, a sequence of single selectors chooses rule applications based on their hierarchy. More specifically, a rule application is chosen for a sequence of selectors if it matches the last single selector, and also if a direct or indirect parent matches the remaining selectors. In addition to basic names, selectors may also specify the admissibility of rule

applications by employing OCL expressions.

Instances of *Property* may be composed of a *TemplateMatch*, a *SlotMatch*, and a *PropertyValue*. A template match contains a *name* and associates a priority for using the template *name* for the rule application under consideration. A *SlotMatch* contains a pair composed of *attributename* and *slotname* and, thereby, defines the priority of an attribute being matched to a slot. A property value associates a property of the rule application to a certain value, which may be a constant or an OCL expression evaluation result. The application of properties may be further limited by restrictive OCL expressions.

TemplateRules manifest as expressions, such as *ForEachExpression* and *IfExpression*, or calls, such as *EvaluationCall* and *TemplateFunctionCall*, and may contain static executed code parts and static outputs. A *ForEachExpression* triggers the generation of code for each object in a parameter of the rule application. A *TemplateFunctionCall* provides an opportunity to create a subsequent rule (subrule) application and may own instances of *TemplateParameterValue* that originate from rule application parameters, values calculated from property rules, and values of a slot (i.e. defined by a multiplicity determining the minimum and maximum amounts of subsequent features that are distributed in the slot). An *EvaluationCall* produces output based on a property value. A *TemplateFunctionCall* produces output either as part of the parent rule (i.e. acting as the container) or as a reference identifier (i.e. triggering content generation outside the parent rule).

The primary syntactic components that may be employed by an ECSS model include pronounceable keywords, such as *import*, *template*, *rule*, *for*, and *if*. Further, *[%=...%]* defines (local) evaluation calls or value calls; *[%...%]* defines Java calls; and *::ruleName()* and *ruleName()* define template-activating character sequences for value insertion through variable access and template function calls respectively.

6.5.3 Notation design, template extension, and IDE

The requirements imposed on the space transportation service language are fulfilled by supplying the ECSS grammar creator with the Ecore metamodel and ECSS model created in steps *E*1*a* and *E*1*b* of Figure 6.3 respectively. In particular, the ECSS model *ws-aware.ecss* and *arbitrary-order.ecss* are created to fulfill the first and second requirement respectively. Figure 6.5 depicts a screenshot of the notation-template language IDE illustrating the design of the notation-template model file named “ws-aware.ecss”.

The rule *whitespaceClassRule* in Listing 6.4 defines that the features of a class are indented (i.e. with respect to the class itself). More specifically, line 6 produces the grammar rule header. Next, in case the class being processed has no associated attribute, the initial content of a class is represented by *classname* followed by an empty space and colon (cf. line 8). Alternatively, in case a class is composed of a set of attributes, a rule call to the rule group *nameDistRules* is performed and a subrule selected (cf. line 10). Next, additional indentation is created for feature definitions that are returned by the rule call *attributeDistRules* (cf. lines 12–14). The property rule *NamedElement+*

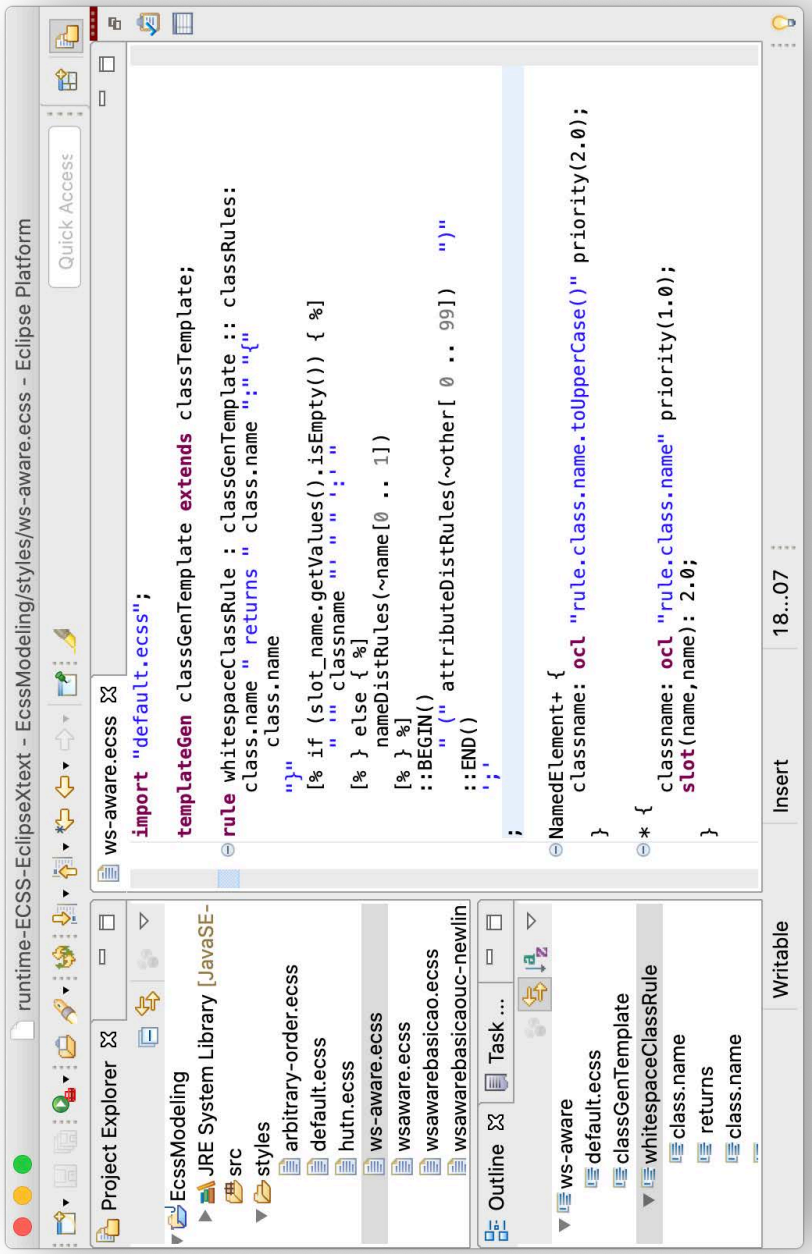

Figure 6.5: Screenshot of notation-template language IDE displaying ECSS model file "ws-aware.ecss".

affects domain-specific metamodel classes that extend a class named *NamedElement* by modeling its *classname* to be represented in uppercase characters (cf. line 19). The global property rule, indicated by a star-selector, is metamodel-agnostic and specifies that the *classname* of (any matching) class is defined by its name or, in other words, the name of a class in its metamodel (cf. lines 23–26). Note that due to the (higher) priority of *2.0*, defined by property rule *NamedElement+*, the global property rule with the (lower) priority of *1.0* is only matched by classes that do *not* extend the class *NamedElement*.

```
1  import "default.ecss";
2
3  templateGen classGenTemplate extends classTemplate;
4
5  rule whitespaceClassRule: classGenTemplate :: classRules:
6     class.name " returns " class.name ":" "{" class.name "}"
7     [% if (slot_name.getValues().isEmpty()) {%]
8        " '" classname "' " " ':' "
9     [% } else {%]
10       nameDistRules(~name[0 .. 1])
11    [% }%]
12    ::BEGIN()
13       " (" attributeDistRules(~other[0 .. 99]) ")"
14    ::END()
15    ';'
16 ;
17
18 NamedElement+ {
19    classname: ocl "rule.class.name.toUpperCase()"
20    priority(2.0);
21 }
22
23 * {
24    classname: ocl "rule.class.name" priority(1.0);
25    slot(name,name): 2.0;
26 }
```

Listing 6.4: Notation-template model for indentation-based layout (excerpt of ECSS model file "ws-aware.ecss").

The rule *arbitraryAttributeDistr* in Listing 6.5 defines a notation-template that fulfills both the first and second notational requirement (i.e. indentation-based arbitrary order of declaration) that is not entailed by the use of (simple) unordered groups (i.e. by use of "&" only). More specifically, line 1 defines the import of the notation-template model with file name "ws-aware.ecss" and, thus, instantiates an inheritance-based template extension mechanism on the notation-template defined above. Next, line 8 defines the initially occurring feature as arbitrary (i.e. any feature from the set of features of a class may occur first). Following that, lines 10–11 encapsulate Java code that computes a list of remaining class features that is subsequently iterated for the production of individual feature occurrences that are indented (cf. lines 13–15).

```
1  import "wsaware.ecss";
```

```
2
3  template attributeTemplate: uk.ac.york.cs.ecss.newproc.AttributeXtendRule;
4  templateGen attributeGenTemplate extends attributeTemplate
5
6  rule arbitraryAttributeDistr: attributeGenTemplate :: attributeDistRules:
7     "("
8     for esf: features join ") | (" {
9        attributeRule(esf)
10       [% List<EStructuralFeature> subFeat = new ArrayList(features); %]
11       [% subFeat.remove(esf); boolean first = true; %]
12       "(("
13       for EStructuralFeature sub: subFeat join ") & (" {
14          attributeRule(sub)
15       } "))"
16    } ")";
```

Listing 6.5: Notation-template model for indentation-based arbitrary order of declaration (excerpt of ECSS model file "arbitrary-order.ecss").

6.5.4 Generation of grammar and language implementation

In this section, the process of generating grammars (cf. step $E2$ in Figure 6.3) that is followed by the final step (cf. $E3$), i.e. the generation of executable textual modeling language implementations as illustrated in Figure 6.6, is described in detail. The grammar generator component of the ECSS framework produces textual modeling language grammars based on ECSS models by substituting template parameters in notation-template models with actual values and suitable subsequent values.

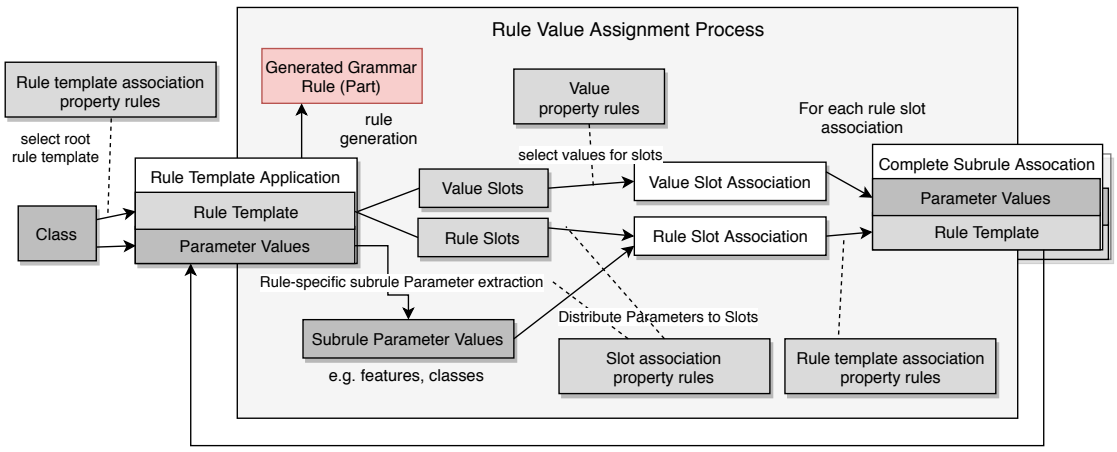

Figure 6.6: Overview of the grammar generation workflow in the ECSS framework.

ECSS is built on template rules that act as code generating classes, producing code based on values that are assigned to class fields. Class fields may manifest as (i) directly assigned fields (i.e. field values determined by constructor parameters), (ii) styled fields

(i.e. field values determined by property selectors), and (*iii*) slot fields (i.e. field values derived from directly assigned fields that are distributed in slots based on priorities computed from the set of available associations between slot and value). Moreover, directly assigned fields may refer to model elements in the input metamodel; the value of styled fields is the result of determining a rule template from the set of available priorities in associations between slot and rule template. Initially, the root rule application is selected using template property rules with their root class as a single parameter. In case no root class is specified, this is automatically determined by the selection of the class, which contains the highest number of (other) classes. Finally, variable associations of the rule application are established depending on the rule application class and any subsequent parameters that are derived from parameters of the rule. Then, property rules assign property values to rule properties. Next, the priority of slot parameters is calculated by applying slot value property rules and specific subsequent parameters that are distributed among rule slot priorities and slot multiplicities. Next, the output-generating function of the rule application (i.e. defined by values computed in the previous step) is executed and dynamically calls subrules by selecting and similarly processing the most suitable subrule templates (i.e. based on property rules and parameter types).

Priority Computation. Assuming that the class *SpaceTransportationService* represents the root class in the space transportation service language metamodel, it is converted by employing the rule *whitespaceClassRule* (cf. Listing 6.4). As a result of the definition of *classGenTemplate*, the variable *class* is set to the class parameters and every class feature is distributed to slots as follows. First, values for non-slot fields (i.e. *classname* in this case) are computed. Although *SpaceTransportationService* is a subclass of *NamedElement* and, therefore, both associations for *classname* are possible, the association defined by the *NamedElement* property rule is chosen due to its higher priority value. Second, features are assigned to slots based on defined priority values or `1.0`, if undefined. In the example, the rule *slot(name,name)* defines a priority of `2.0` and, thus, causes the attribute *name* and remaining attributes to be associated with the slot *name* and the slot *other* respectively. The generation process in the example continues with a call to the rule group *nameDistRules* (i.e. imported from the notation-template model in file named *default.ecss* shown in Appendix 5) with feature *name* as a parameter for the non-empty slot *name*. Finally, rule *arbitraryDistRule* is prioritized over rule *defaultAttributeDistr* (i.e. similarly imported from *default.ecss*) due to its priority value being higher by `0.5`. Thus, *attributeDistRules([engineAmount, engineType, physicalProperties])* is established as a new rule call that is similarly executed.

6.5.5 Prototype implementation

The approach introduced in this chapter has been prototypically realized using the EMF [197] and Xtext [70]. Additional information, such as slides, source code, and examples, of the ECSS prototype is provided at a dedicated web page: `https://www-users.cs.york.ac.uk/neubauer/ecss/`.

6.5.6 Limitations

Although the approach presented in this chapter is technology agnostic, the implementation of the ECSS framework is technologically based on the EMF and Xtext framework, as well as the Java programming language. Therefore, executing the ECSS grammar generator and notation-specification IDE requires running a Java virtual machine and Eclipse with installed Xtext plugins. As a consequence, the support for different language workbench frameworks and, in particular, those that build on technologies that are fundamentally different from the EMF and the Java programming language will require the (re)construction of behavior by following the principles of the targeted technology.

6.6 Evaluation

This section introduces the evaluation design based on the structure for conducting and reporting case study research in software engineering by Runeson and Höst [187]: objective, case, theory, research questions, method, and selection strategy. Both the results and the threads of validity are presented. Ultimately, the obtained results are discussed in the context of DSL engineering.

The objective of the study is to demonstrate the applicability of the proposed approach and, in particular, notation-template language and its capability to capture textual notations for DSLs in a more concise, expressive, and integratable manner, such as the applicability of notation-template models to arbitrary domains (i.e. represented by concepts and relationships in domain-specific metamodels) and the conciseness of notation-templates in comparison to grammar specifications created by hand or generated by the Xtext metamodel-to-grammar generator.

6.6.1 Research questions and evaluation criteria

The case study is represented by the proposed textual notation-specification language and composed of a set of analysis units, which have been reported as paramount usability indicators of DSLs [105, 6]. These include integratability, expressiveness, and conciseness.

The following declares a set of research questions that are designed to support the evaluation objective, which is to evaluate the usability of the proposed notation-specification language for modeling languages.

RQ1: *Are* ECSS *models integratable with languages of arbitrary domains?*

RQ2: *Is the* ECSS *language capable of expressing the description of notations of textual modeling languages?*

RQ3: *Are* ECSS *models more concise than notation-descriptions in grammar specifications?*

Integratability. The integratability of a DSL is represented by its ability to integrate with other applications, which may include other DSLs. It, thus, also represents factors

that decrease the cost of development and maintenance. Therefore, in the case of the approach presented in this chapter, integratability refers to the ability to integrate a textual notation-specification (represented by an ECSS model) with other domains and, more specifically, arbitrary concepts that are defined as members of a domain (i.e. captured by the structural components in the metamodel of a DSL).

Expressiveness. The expressiveness of a DSL refers to the degree that a language represents the concepts of a domain directly. In other words, it measures the ability to express the logic that is necessary for the respective application domain. Thus, in the case of the proposed approach, expressiveness refers to the degree to which the intended notation of a textual modeling language can describe the means of ECSS models.

Conciseness. The conciseness of a DSL refers to the economy of terms without harming the artifact's comprehension (i.e. short yet comprehensible statements). More specifically, decreasing the occurrence of boilerplate code and inadequate representations enables faster error detection and increases the ability to involve domain experts, both of which lower the cost of development and maintenance. Consequently, in the case of the proposed notation-specification language, conciseness refers to the ability to express intended textual modeling language notations through brief and comprehensible ECSS models.

6.6.2 Procedure

This section presents the methodological steps to evaluate the approach presented in this chapter and, in particular, its implementation by the ECSS framework. More specifically, the expressiveness, conciseness, and integratability (i.e. the DSL usability factors outlined in the literature [105]) of the ECSS language are measured based on the following types of artifacts. First, a pre-existing handcrafted textual modeling language and its grammar are referred to as *source language* and *source grammar* respectively. Second, a textual modeling language and its grammar, which is generated by the Xtext state-of-the-art metamodel-to-grammar transformation, are referred to as *default language* and *default grammar* respectively. Third, a textual modeling language generated by the ECSS framework (i.e. from combinations of domain-specific metamodels and ECSS *model*) and its grammar are referred to as *target language* and *target grammar* respectively. Fourth, a set of grammar metrics are established from components of the Xtext grammar language metamodel and include parser rules, terminal rules, type reference rules, keywords, alternatives, groups, assignments, and concrete syntax validation diagnostics (as reported by the interface *IConcreteSyntaxDiagnosticProvider*[3]). Fifth, a set of ECSS models are employed as notation specifications for the generation of variations of target languages. These include "basicao.ecss", "basicaouc.ecss", "basicsq.ecss", "basicnoao.ecss", and "basicnows.ecss"[4].

[3]The implementation of org.eclipse.xtext.validation.IConcreteSyntaxDiagnosticProvider is available online at `http://bit.ly/2uuekKg`.

[4]This set of ECSS models is available online at `https://github.com/patrickneubauer/ECSS/tree/master/styles`.

Integratability. The integratability of the proposed language is evaluated according to the following steps. First, the ECSS framework is applied to generate a set of target grammars for each combination of ECSS models and domain-specific metamodels (i.e. capturing innate domain concepts). More specifically, the former is represented by "basicao.ecss" (i.e. a notation specification that produces grammars that allow the creation of models following an arbitrary order of declaration), "basicaouc.ecss" (i.e. a notation specification that produces grammars with uppercase keywords for classes that extend the class named "NamedElement"), "basicsq.ecss", "basicnoao.ecss" (i.e. a notation specification that produces grammars that allow the creation of models following a sequential order of declaration), and "basicnows.ecss" (i.e. a notation specification that produces grammars that do not assign semantics to whitespace tokens occurring in models). The latter is represented by the Ecore metamodel of a domain-specific language in the set of subject languages. Second, for each generated target grammar, the count of grammar rules (i.e. instances of terminal rules, enumeration rules, and parser rules) is computed. Finally, in order to evaluate the integratability of the presented approach, the mean and median of the count of grammar rules for each set of target languages that has been generated from the same domain-specific metamodel are compared based on equality.

Expressiveness. The expressiveness of the ECSS language is evaluated according to the following steps. First, the Xtext metamodel-to-grammar transformation is employed to generate a *default grammar* for each domain-specific metamodel in the set of subject languages. Second, the ECSS framework is applied for the automated generation of a *target grammar* for each pair that consists of a metamodel-agnostic ECSS model and a domain-specific metamodel in the set of subject languages. More specifically, in order to mitigate the risk of generating target languages that closely resemble the notation that is depicted by a respective source grammar, a common metamodel-agnostic ECSS model named "basicnoao.ecss" is employed for each language in the set of subject languages. Third, for each language (i.e. *source language*, *default language*, and *target language*), the respective grammar (i.e. *source grammar*, *default grammar*, and *target grammar*) is employed to compute a set of grammar metrics that are composed of the number of parser rules, terminal rules, type reference rules, keywords, alternatives, groups, assignments, and concrete syntax validation diagnostics. Fourth, the percentile closeness between instances of *default language* and *source language*, as well as *target language* and *source language*, in each of the aforementioned grammar metric types is computed based on the following respective formulas:

$$closeness_{DefGra,SrcGra} = 1 - \left\| \frac{MetricType_{DefGra} - MetricType_{SrcGra}}{MetricType_{SrcGra}} \right\|$$

$$closeness_{TrgGra,SrcGra} = 1 - \left\| \frac{MetricType_{TrgGra} - MetricType_{SrcGra}}{MetricType_{SrcGra}} \right\|$$

Fifth, the difference between the percentile closeness in individual metric types (i.e. parser rules, terminal rules, type reference rules, keywords, alternatives, groups, assignments, and concrete syntax validation diagnostics) of instances of *default language*

and *source language*, as well as *target language* and *source language*, is computed. Finally, the mean and median of grammar metric type percentile closeness is computed for *default language* and *source language*, as well as *target language* and *source language*, to illustrate the expressiveness of the presented approach in comparison to the Xtext metamodel-to-grammar transformation.

Conciseness. The method for gathering the artifacts required to determine the conciseness of the ECSS language is initiated with a similar procedure as that described for expressiveness. More specifically, the ECSS *model* named "basicnoao.ecss" that has been applied by the framework alongside the respective domain-specific metamodel to produce the *target grammar* automatically, is numerically quantified based on the following set of metrics. First, the factor between LOC_{SrcGra} (i.e. the total Lines of Codes (LOCs) of the *source grammar*) and LOC_{ECSS} (i.e. the total LOCs of the ECSS *model*) is computed. Second, the state-of-the-art metamodel-to-grammar transformation provided by the Xtext framework is employed to obtain the *default grammar* that is subsequently adapted to yield a parser capable of interpreting *source models*. Third, the increase in conciseness of the ECSS *model* to the respective source grammar, as well as the default grammar, is computed as follows:

$$conciseness_{Ecss,SrcGra} = 1 - \frac{LOC_{Ecss}}{LOC_{SrcGra}}$$

$$conciseness_{Ecss,DefGra} = 1 - \frac{LOC_{Ecss}}{LOC_{DefGra}}$$

Fourth, in order to address the risk of eventually existing additional semantics represented by single- or multi-line comments and empty lines in respective grammars, the increase in conciseness of the ECSS *model* is also computed based on the amount of grammar rules (i.e. parser rules, enumeration rules, and terminal rules) as follows:

$$conciseness_{TrgGra,SrcGra} = 1 - \frac{GrammarRuleCount_{TrgGra}}{GrammarRuleCount_{SrcGra}}$$

$$conciseness_{TrgGra,DefGra} = 1 - \frac{GrammarRuleCount_{TrgGra}}{GrammarRuleCount_{DefGra}}$$

6.6.3 Selection strategy

The evaluation of the subject selection strategy is composed of the following steps: (*i*) retrieval of publicly available Xtext projects from Github; (*ii*) sorting of retrieved projects according to normalized popularity as indicated by the sum of the normalized number of assignees, subscribers, and stargazers of a given project, where each normalized number is computed by the subtraction of the average and division by standard division over the entire set of values followed by the weight-distribution of $assignees_{\text{weight}} = 0.1$, $subscribers_{\text{weight}} = 0.2$, and $stargazers_{\text{weight}} = 0.7$. Ultimately, a set of languages with a normalized weighted popularity of ≥ 0.5 is obtained; (*iii*) filtering of projects that can be compiled and provision of five or more conforming models; and (*iv*) selection of the first ten projects (i.e. projects accepted in the previous step) as final subjects of the study.

6.6.4 Results

This section presents the results of the evaluation of the contributed framework and notation-template language for textual modeling languages. The results of employing the above described selection strategy are shown in Table 6.1. In total, 4,910 publicly available Xtext projects have been retrieved. Further, the absolute, normalized, and normalized and weighted popularity of the final selection of projects ranges from 79 to 2,910, 5.81 and 87.80, and 0.87 and 48.17 respectively.

Github Xtext project owner / repository	Unified grammar name	Popularity	Popularity (normalized)	Popularity (normalized, weighted)
antlr4ide / antlr4ide[5]	Antlr4	226	8.34	3.79
reTHINK-project / dev-service-framework[6]	ClassDiagram	79	7.80	0.96
puppetlabs / geppetto[7]	Module_2	570	59	8.93
cloudsmith / geppetto[8]	PP_1	219	7.57	3.53
puppetlabs / geppetto[9]	PP_2	570	59	8.93
temenostech / IRIS[10]	RIMDsl_2	121	11.17	1.93
sculptor / sculptor[11]	Sculptordsl	238	9.66	4
wesnoth / wesnoth[12]	WML_6	2910	87.80	48.17
uqbar-project / wollok[13]	WollokDsl	107	7.39	1.50
AKSW / Xturtle[14]	Xturtle	81	5.81	0.87

Table 6.1: Languages obtained by the execution of the selection strategy.

Integratability

Table 6.2 illustrates the grammar rule counts produced by the ECSS framework for the sets of selected languages and notation-specifications. More specifically, the former and latter are represented by domain-specific metamodels in the form of Ecore files and ECSS models respectively. The number of grammar rules produced by the grammar generator from a domain-specific metamodel and varying ECSS models is consistent.

[5]The Antlr4id Github project is located at https://github.com/antlr4ide/antlr4ide.
[6]The dev-service-framework Github project is located at https://github.com/reTHINK-project/dev-service-framework.
[7]The Modules grammar of the Puppetlabs Geppetto Github project is located at https://bit.ly/2KQraHH.
[8]The Cloudsmith Geppetto Github project is located at https://github.com/cloudsmith/geppetto.
[9]The Puppetlabs Geppetto Github project is located at https://github.com/puppetlabs/geppetto.
[10]The IRIS Github project is located at https://github.com/temenostech/IRIS.
[11]The Sculptor Github project is located at https://github.com/sculptor/sculptor.
[12]The Wesnoth Github project is located at https://github.com/wesnoth/wesnoth.
[13]The Wollok Github project is located at https://github.com/uqbar-project/wollok.
[14]The Xturtle Github project is located at https://github.com/AKSW/Xturtle.

Thus, in terms of ***RQ1***, the produced results indicate that the ECSS framework is able to generate target grammars successfully for every language in the set of selected languages while maintaining grammar rule count consistency. This indicates that the ECSS framework is integratable with the set of selected languages and the set of varying ECSS models.

Unified grammar name	basicao.ecss	basicaouc.ecss	basicsq.ecss	basicnoao.ecss	basicnows.ecss
Antlr4	60	60	60	60	60
ClassDiagram	22	22	22	22	22
Module_2	22	22	22	22	22
PP_1	84	84	84	84	84
PP_2	84	84	84	84	84
RIMDsl_2	48	48	48	48	48
Sculptordsl	52	52	52	52	52
WML_6	22	22	22	22	22
WollokDsl	57	57	57	57	57
Xturtle	23	23	23	23	23
Mean	50	50	50	50	50
Median	47.4	47.4	47.4	47.4	47.4

Table 6.2: Grammar rule count of different variants of the *target language*.

Expressiveness

Tables 6.3, 6.4, and 6.5 present the absolute numbers of the grammar metrics that have been computed for the set of selected *source languages*, *default languages*, and *target languages* respectively, and from their respective grammars. The generation of target grammars entailed the application of the ECSS model named "basicao.ecss". Table 6.6 depicts the percentile closeness between the grammar metrics of pairs of *default language* and *source language* (cf. Table 6.4 and Table 6.3 respectively). Table 6.7 depicts the percentile closeness between the grammar metrics of pairs of *target language* and *source language* (cf. Table 6.5 and Table 6.3 respectively). Note that the value "N/A" indicates that a result could not be produced as a consequence of division by zero. For example, the target language named "Module_2" has a count of zero enumeration rules.

Table 6.8 depicts the difference of the percentile closeness between the grammar metrics of pairs of *default language* and *source language* (cf. Table 6.6) and the percentile closeness between the grammar metrics of pairs of *target language* and *source language* (cf. Table 6.7). In other words, Table 6.8 illustrates the percentile gain or loss in grammar metric closeness between the Xtext metamodel-to-grammar transformation (i.e. producing instances of *default language*) and the ECSS grammar generator (i.e. producing instances of *target language*) in respect to instances of *source language*.

In terms of ***RQ2***, the produced results indicate that the ECSS framework is capable of generating languages (from pairs that consist of domain-specific metamodels and ECSS models) that are closer to handcrafted languages than languages produced by the Xtext metamodel-to-grammar generator (from domain-specific metamodels) when comparing instances of Xtext grammar components.

More specifically, the ECSS framework outperforms the Xtext metamodel-to-grammar generator in percentile metric closeness in seven and five out of eight types of metrics measured by mean and median respectively. In other words, supplying the ECSS grammar generator with pairs consisting of a common and metamodel-agnostic ECSS model and a domain-specific metamodel results in the production of languages that are closer to handcrafted languages and, therefore, more expressive than those produced by the Xtext metamodel-to-grammar generator.

Unified grammar name	Parser rule count	Enum rule count	Terminal rule count	TypeRef rule count	Key-word count	Alter-natives count	Group count	Assign-ment count	Action count	Valid-ation count
Antlr4	70	1	26	109	172	44	87	161	6	13
ClassDiagram	18	4	8	37	106	27	36	41	1	8
Module_2	58	0	6	72	146	23	56	74	7	9
PP_1	94	0	15	155	259	50	139	123	46	47
PP_2	94	0	15	155	259	50	139	123	46	47
RIMDsl_2	42	0	2	67	151	26	69	108	5	24
Sculptordsl	49	5	6	92	572	83	284	421	0	33
WML_6	18	0	18	36	114	28	36	26	0	1
WollokDsl	84	0	9	141	254	64	174	125	35	49
Xturtle	27	0	15	53	122	26	42	29	7	12
Mean	55.40	1	12	91.70	215.50	42.10	106.20	123.10	15.30	24.30
Median	53.5	0	12	82	161.5	36	78	115.5	6.5	18.5

Table 6.3: Grammar metrics of *source languages*.

Conciseness

Table 6.9 presents the total amount of LOC of the metamodel-agnostic ECSS model, the *source grammar*, the *default grammar*, and the *target grammar* for each language in the sets of selected languages. More specifically, LOC_{Ecss} is represented by the LOC of the ECSS model named "basicao.ecss" and takes into account the LOC of imported models (i.e. "noao.ecss" with 65 LOC and "nows.ecss" with 94 LOC). Further, LOC_{SrcGra}, LOC_{DefGra}, and LOC_{TrgGra} are represented by the LOC of the grammar of the *source language*, *default language*, and *target language* respectively. Table 6.10 compares the conciseness between ECSS model and *source grammar*, ECSS model and *default grammar*, *target grammar* and *source grammar*, and target grammar and default grammar.

Unified grammar name	Parser rule count	Enum rule count	Terminal rule count	TypeRef rule count	Key-word count	Alter-natives count	Group count	Assign-ment count	Action count	Valid-ation count
Antlr4	59	1	0	113	283	9	152	112	50	53
ClassDiagram	19	4	0	43	99	9	41	43	14	21
Module_2	20	0	0	37	81	3	38	25	16	18
PP_1	74	0	0	139	390	8	209	148	65	66
PP_2	74	0	0	139	391	8	210	149	65	66
RIMDsl_2	48	0	0	110	242	6	138	124	41	63
Sculptordsl	49	5	0	133	739	12	414	518	41	80
WML_6	23	0	0	40	86	6	46	37	17	18
WollokDsl	59	0	0	124	310	8	158	126	51	65
Xturtle	31	0	0	60	105	7	57	33	24	30
Mean	45.60	1	0	93.80	272.60	7.60	146.30	131.50	38.40	48
Median	48.5	0	0	111.5	262.5	8	145	118	41	58

Table 6.4: Grammar metrics of *default languages*.

Unified grammar name	Parser rule count	Enum rule count	Terminal rule count	TypeRef rule count	Key-word count	Alter-natives count	Group count	Assign-ment count	Action count	Valid-ation count
Antlr4	50	1	4	101	177	15	82	106	43	46
ClassDiagram	15	4	3	39	65	16	26	40	11	18
Module_2	17	0	2	32	55	7	23	20	12	14
PP_1	78	0	3	143	235	22	115	136	62	63
PP_2	78	0	3	143	235	22	115	137	62	63
RIMDsl_2	42	0	4	99	152	12	80	118	35	54
Sculptordsl	42	5	4	119	188	19	131	449	36	69
WML_6	18	0	2	31	65	14	34	34	11	12
WollokDsl	51	0	3	112	187	15	109	122	44	58
Xturtle	19	0	2	37	62	14	23	20	12	17
Mean	41	1	3	85.60	142.10	15.60	73.80	118.20	32.80	41.40
Median	42	0	3	100	164.5	15	81	112	35.5	50

Table 6.5: Grammar metrics of *target languages*.

In terms of ***RQ3***, the produced results indicate that the employed ECSS model is approximately 20 and 51 percent more concise by mean and median respectively, than *source grammar*. Further, the employed ECSS model is approximately 36 and 57 percent more concise by mean and median respectively, when compared with *default grammar*. Moreover, *target grammar* (i.e. grammar generated by the ECSS framework) is approximately 31 and 32 percent more concise by mean and median respectively, than

Unified grammar name	Parser rule count	Enum rule count	Terminal rule count	TypeRef rule count	Key-word count	Alter-natives count	Group count	Assign-ment count	Action count	Valid-ation count
Antlr4	84.29%	100%	0%	96.33%	35.47%	20.45%	25.29%	69.57%	-633.33%	-207.69%
ClassDiagram	94.44%	100%	0%	83.78%	93.40%	33.33%	86.11%	95.12%	-1200%	-62.50%
Module_2	34.48%	N/A	0%	51.39%	55.48%	13.04%	67.86%	33.78%	-28.57%	0%
PP_1	78.72%	N/A	0%	89.68%	49.42%	16%	49.64%	79.67%	58.70%	59.57%
PP_2	78.72%	N/A	0%	89.68%	49.03%	16%	48.92%	78.86%	58.70%	59.57%
RIMDsl_2	85.71%	N/A	0%	35.82%	39.74%	23.08%	0%	85.19%	-620%	-62.50%
Sculptordsl	100%	100%	0%	55.43%	70.80%	14.46%	54.23%	76.96%	N/A	-42.42%
WML_6	72.22%	N/A	0%	88.89%	75.44%	21.43%	72.22%	57.69%	N/A	-1600%
WollokDsl	70.24%	N/A	0%	87.94%	77.95%	12.50%	90.80%	99.20%	54.29%	67.35%
Xturtle	85.19%	N/A	0%	86.79%	86.07%	26.92%	64.29%	86.21%	-142.86%	-50%
Mean	78.40%	N/A	0%	76.57%	63.28%	19.72%	55.94%	76.23%	N/A	-183.86%
Median	81.50%	N/A	0%	87.37%	63.14%	18.23%	59.26%	79.27%	N/A	-46.21%

Table 6.6: Percentile closeness between grammar metrics of pairs consisting of *default language* and *source language*.

Unified grammar name	Parser rule count	Enum rule count	Terminal rule count	TypeRef rule count	Key-word count	Alter-natives count	Group count	Assign-ment count	Action count	Valid-ation count
Antlr4	71.43%	100%	15.38%	92.66%	97.09%	34.09%	94.25%	65.84%	-516.67%	-153.85%
ClassDiagram	83.33%	100%	37.50%	94.59%	61.32%	59.26%	72.22%	97.56%	-900%	-25%
Module_2	29.31%	N/A	33.33%	44.44%	37.67%	30.43%	41.07%	27.03%	28.57%	44.44%
PP_1	82.98%	N/A	20%	92.26%	90.73%	44%	82.73%	89.43%	65.22%	65.96%
PP_2	82.98%	N/A	20%	92.26%	90.73%	44%	82.73%	88.62%	65.22%	65.96%
RIMDsl_2	100%	N/A	0%	52.24%	99.34%	46.15%	84.06%	90.74%	-500%	-25%
Sculptordsl	85.71%	100%	66.67%	70.65%	32.87%	22.89%	46.13%	93.35%	N/A	-9.09%
WML_6	100%	N/A	11.11%	86.11%	57.02%	50%	94.44%	69.23%	N/A	-1000%
WollokDsl	60.71%	N/A	33.33%	79.43%	73.62%	23.44%	62.64%	97.60%	74.29%	81.63%
Xturtle	70.37%	N/A	13.33%	69.81%	50.82%	53.85%	54.76%	68.97%	28.57%	58.33%
Mean	76.68%	N/A	25.07%	77.45%	69.12%	40.81%	71.50%	78.84%	N/A	-89.66%
Median	82.98%	N/A	20%	82.77%	67.47%	44%	77.48%	89.02%	N/A	17.68%

Table 6.7: Percentile closeness between grammar metrics of pairs consisting of *target language* and *source language*.

source grammar.

Unified grammar name	Parser rule count	Enum rule count	Terminal rule count	TypeRef rule count	Key-word count	Alter-natives count	Group count	Assign-ment count	Action count	Valid-ation count
Antlr4	-12.86%	0%	15.38%	-3.67%	61.63%	13.64%	68.97%	-3.73%	116.67%	53.85%
ClassDiagram	-11.11%	0%	37.50%	10.81%	-32.08%	25.93%	-13.89%	2.44%	300%	37.50%
Module_2	-5.17%	N/A	33.33%	-6.94%	-17.81%	17.39%	-26.79%	-6.76%	57.14%	44.44%
PP_1	4.26%	N/A	20%	2.58%	41.31%	28%	33.09%	9.76%	6.52%	6.38%
PP_2	4.26%	N/A	20%	2.58%	41.70%	28%	33.81%	9.76%	6.52%	6.38%
RIMDsl_2	14.29%	N/A	0%	16.42%	59.60%	23.08%	84.06%	5.56%	120%	37.50%
Sculptordsl	-14.29%	0%	66.67%	15.22%	-37.94%	8.43%	-8.10%	16.39%	N/A	33.33%
WML_6	27.78%	N/A	11.11%	-2.78%	-18.42%	28.57%	22.22%	11.54%	N/A	600%
WollokDsl	-9.52%	N/A	33.33%	-8.51%	-4.33%	10.94%	-28.16%	-1.60%	20%	14.29%
Xturtle	-14.81%	N/A	13.33%	-16.98%	-35.25%	26.92%	-9.52%	-17.24%	171.43%	108.33%
Mean	-1.72%	N/A	25.07%	0.87%	5.84%	21.09%	15.57%	2.61%	N/A	94.20%
Median	-7.35%	N/A	20%	-0.10%	-11.07%	24.50%	7.06%	4%	N/A	37.50%

Table 6.8: Differences between percentile metric closeness of pairs consisting of *default grammar* and *source grammar* (cf. Table 6.6), and *target grammar* and *source grammar* (cf. Table 6.7).

Unified grammar name	LOC_{Ecss}	LOC_{SrcGra}	LOC_{DefGra}	LOC_{TrgGra}
Antlr4	175	739	569	127
ClassDiagram	175	115	148	79
Module_2	175	255	138	52
PP_1	175	748	567	177
PP_2	175	768	568	177
RIMDsl_2	175	289	364	183
Sculptordsl	175	535	835	318
WML_6	175	98	156	79
WollokDsl	175	462	454	123
Xturtle	175	86	198	77
Mean	175	409.5	401.1	139.2
Median	175	375.5	411	125

Table 6.9: LOC of ECSS *model*, *source grammar*, *default grammar*, and *target grammar*.

6.6.5 Threats to validity

This section describes threats to validity as revealed by the presented evaluation.

Unified grammar name	*conciseness Ecss,SrcGra*	*conciseness Ecss,DefGra*	*conciseness TrgGra,SrcGra*	*conciseness TrgGra,DefGra*
Antlr4	76.32%	69.24%	38.14%	25%
ClassDiagram	-52.17%	-18.24%	26.67%	4.35%
Module_2	31.37%	-18.24%	65.63%	0%
PP_1	76.60%	69.14%	22.94%	-1.20%
PP_2	77.21%	69.19%	25%	-1.20%
RIMDsl_2	39.45%	52.45%	-9.09%	4%
Sculptordsl	67.29%	79.04%	13.33%	16.13%
WML_6	-78.57%	-12.18%	38.89%	15.38%
WollokDsl	62.12%	61.45%	38.71%	16.18%
Xturtle	-103.49%	11.62%	45.24%	-9.52%
Mean	19.61%	36.35%	30.55%	6.91%
Median	50.78%	56.95%	32.41%	4.17%

Table 6.10: Comparison of ECSS *model* and *source language*, *default language*, and *target language* in LOC of grammar and parser rule count.

Construct Validity The parameter values that have been chosen within the selection strategy and, in particular, weight-distribution, weighted popularity, and number of available source grammars, represent guiding preferences in sorting retrieved textual domain-specific modeling languages. They, thus, determine the final selection of study subjects. Choosing different parameter values for weight-distribution, weighted popularity, and number of available source grammars may produce a different set of study subjects, which subsequently may lead to different evaluation results. Although the said parameter values have been chosen from previous experience with mining open-source software artifacts [128, 188], the risk of diverging evaluation results caused by different parameter values may be mitigated by re-executing the presented evaluation through the use of different sets of parameter values. Moreover, compared to grammars that are produced by the Xtext metamodel-to-grammar transformation or the ECSS framework (i.e. through combinations of domain-specific metamodels and ECSS models), grammars that are produced by hand may contain additional semantics, such as indentations and comments, that may affect the conciseness based on LOC. In order to mitigate this risk, the results of conciseness have been validated based on the number of grammar rules (i.e. parser rules, enumeration rules, and terminal rules).

Internal Validity The direct quantification of *performance* (described in existing literature [105] as a usability quantifier for DSLs), for example through execution time, has not been considered. Grammar specifications that are similar, however, may lead to the production of parsers with similar execution times, and adaptations of implementation classes, such as model validators (i.e. neglecting the parsing of domain-specific models),

and have, therefore, been neglected. In order to mitigate this risk, individual grammar specifications (i.e. instances of *source grammar*, *default grammar*, and *target grammar*) have been employed to generate executable parsers and implementation classes.

External Validity External validity is concerned with the extent to which it is possible to generalize findings and whether these findings are of interest to third-parties outside the investigated case. On the one hand, the presented findings illustrate the potential of the approach to be integrated with the EMF and Xtext framework. On the other hand, the presented findings are limited to the said set of modeling languages and challenge the potential of the approach to be integrated with third-party tools, such as language workbench frameworks that are different to the EMF and Xtext framework. More specifically, the ability to re-use the EMF and Java-based implementation of the ECSS framework, and the effort required to implement similar behavior based on the codebase of existing third-party tools requires further investigation for tools that deviate from ANTLR-based grammars. Although a set of notation-specifications has been employed to generate a set of target grammars from each domain-specific metamodel, the employed set of modeling languages (captured by the abstract concepts depicted in respective domain-specific metamodels) is not complete. In order to mitigate this risk, however, the final set of modeling languages that has been employed for the evaluation of the approach is composed of the most popular and, thus, relevant projects.

Reliability The usability quantifier for the conciseness of DSLs also includes comprehensibility. The interpretation of language notations that are depicted by grammars, models, and (formal) standard specifications may depend on the experience and knowledge of the participant at hand and, in particular, in the fields of language engineering and MDE. Thus, participants with diverging experiences in those fields may produce deviating results for conciseness. In order to determine the potential impact of the latter, further studies must be conducted and, if required, measures to mitigate that risk may need to be developed.

Conclusion validity Conclusion validity is the degree to which conclusions we reach about relationships in our data are reasonable. The evaluation of the approach presented in this chapter and, in particular, the set of metrics chosen for expressiveness represent an instance in this category of threats. More specifically, the chosen set of metrics includes a subset of classes that appear in the metamodel of the Xtext grammar language. In order to mitigate this threat, a set of metrics has been chosen that is based on fundamental grammar components, such as parser rules, enumeration rules, and terminal rules, as well as a set of periphery grammar components, such as type reference rules, alternatives, groups, assignments, and actions.

6.6.6 Summary

*Integration with arbitrary domains (**RQ1**)*: The obtained results indicate that the implementation of the approach is integratable with domain-specific languages that are defined by means of Ecore metamodels.

*Notation-specification language expressiveness (**RQ2**)*: The application of the approach, and in particular the metamodel-agnostic notation-specifications, to Ecore metamodels of a set of popular Xtext-based languages produces language grammars that express the components in the grammar of handcrafted languages more closely when compared to respective language grammars that are produced by the Xtext metamodel-to-grammar generator.

*Conciseness of notation-specifications (**RQ3**)*: The conciseness of notation-specifications that are modeled as ECSS models are approximately 20 and 51 percent more concise by mean and median respectively, than their handcrafted counterparts.

6.7 Analysis

This section presents existing literature on the specification of visual textual representations and differentiates them from the approach presented in this chapter.

A classification of concrete textual syntax mapping approaches presented in [89] distinguishes among approaches that are based on (*i*) manual development or auto-generation of domain-specific metamodels from grammar specifications; (*ii*) manual development of grammar specifications based on domain-specific metamodels; and (*iii*) manual development of mappings between domain-specific metamodels and grammar specifications. The approach presented in this chapter automates the generation of grammar specification from pairs consisting of domain-specific metamodels and ECSS models (i.e. acting as notation-specification) by instantiating a transformation that is capable of both metamodel-agnostic and metamodel-dependent notation-specifications. Therefore, according to the classification in [89], the approach presented in this chapter represents a novel technique and may be categorized as automated development of grammar specifications based on domain-specific metamodels and notation-specifications that are structure-agnostic, structure-dependent, or a combination thereof.

In [115], concrete textual representations are defined using model annotations specified in terms of a dedicated DSL. This approach represents an effort towards reducing redundancy between the specification of metamodels and grammars (e.g. as introduced by the duplicated definition of element-multiplicity) by employing a sequence of transformations on Textual Concrete Syntax (TCS) models and metamodels. Compared to the approach presented in this chapter, the definition of a DSL is achieved utilizing TCS models that are similarly employed alongside metamodels for the generation of textual grammar. As a

result of TCS models specifying associations to individual metamodel elements, however, this approach does not enable the definition of domain metamodel-agnostic styles and, thus, limits the application of styles to particular metamodels.

TCSSL [158] represents an approach to establish bidirectional mappings between abstract syntax trees and concrete syntax trees by defining EBNF-like rules, which differ from EBNF rules by having sub-rules that are triggered based on the inheritance hierarchy depicted in abstract syntax trees. Compared to the approach presented in this chapter, TCSSL also allows to define multiple different mappings based on the same metamodel in order to provide different concrete representations of the same abstract concepts (i.e. catering for the needs of different stakeholders). At the same time, it does not allow to define concrete representations that are applicable to different metamodels. Moreover, TCSSL requires language engineers to specify mapping rules manually for each metamodel element as well as each concrete representation thereof, instead of employing structure-agnostic notation-template models to arbitrary domain-specific metamodels. Further, multiple pass-analysis checks need to be manually implemented in order to address challenges, such as type checking and reference resolution mechanisms, that are raised during the compiler construction process.

The Textual Editing Framework (TEF) [190] represents an approach to embed generated EMF-based textual model editors into graphical editors created with GMF and tree-based editors that are generated by the EMF. Compared to the approach presented in this chapter, TEF similarly offers the capability to create style specifications for domain metamodels. TEF, however, requires language engineers to implement manually complete concrete syntax specifications (i.e. also referred to as "textual syntax language models" in [190]) for elements in a domain-specific metamodel that are intended to be instantiated within the context of the embedded textual editor. Although TEF presents a layout manager to adapt the textual concrete syntax of a language, such as by customizing whitespace, it is based on the metamodel of the Ecore language (i.e. it is structurally defined by the components of the Ecore language metamodel) and, thus, ignores the ability to define notation-specifications for individual domain-specific metamodels.

An approach that is built on the Spoofax language workbench and offers library-based syntactic extensibility for the host languages Java, Haskell, and Prolog, based on the Spoofax language workbench, is presented in [118]. More specifically, this approach employs transformations to weave and unweave notational information (referred to as "syntactic sugar") in and out of host language notations respectively. In comparison to the approach presented in this chapter, notation-specifications are represented as extensions of host language grammar and involve the construction of transformations that depend on the grammar of a particular host language, as opposed to decoupling notational information from domain-specific concepts (i.e. captured by instances of a host language in [118]). Consequently, this approach requires the construction and maintenance of bi-directional transformations to enable grammar backward-compatibility.

SugarJ [68] presents an approach for the development of language extension modules to embed DSLs into the Java host language by means of object language grammar

extensions (referred to as "syntactic sugar" by Erdweg et al.). In comparison to the approach presented in this chapter, this approach extends host language grammar with notation-specifications by executing specifically constructed grammar-dependent transformations (i.e. adding and removing style information from a particular grammar), as opposed to the decoupling of notational information from domain-specific concepts that are introduced by a domain-specific metamodel. Hence, by treating metamodels and notation-specification models as first-class citizens, and the generation of target grammars as a bridging-component to the generation of target language components, such as dedicated IDEs, the approach presented in this chapter satisfies the need to retain the backward-compatibility of language grammar and, subsequently, liberates language engineers from creating and maintaining host language grammar transformations. Furthermore, both host language grammar syntax sugar extensions (i.e. acting as notation-specifications) and desugaring transformations (i.e. obligatory artifacts to obtain source grammar specifications) depend on a particular grammar and are, therefore, not agnostic to the application of individual domain-specific metamodels. As a consequence, this approach imposes challenges on target languages due to dependencies that are introduced between sugar libraries and modeling language components. Although SugarJ and the approach presented in this chapter offer implementations that manifest as Java source code and Eclipse-based IDEs, distinctive language workbench frameworks (i.e. Spoofax and Xtext respectively) are employed. As a consequence, the creation and customization of target language components, such as domain-specific IDEs, are driven by the capabilities and limitations of the underlying language workbench framework. In terms of commonalities, the implementation of both SugarJ and the approach presented in this chapter manifest as Java programming language source code and offer Eclipse-based IDEs.

EMFText [101, 102] presents an approach for the definition of textual representations and the generation of editors from Ecore-based metamodels. Similarly to the approach presented in this chapter, ANTLR-based parser generation is employed and concrete syntax rules[15] may be defined based on domain-specific metamodel classes and attributes. The definition and application of metamodel-agnostic notations, however, are not considered. More specifically, if the condition on the right-hand side of a concrete syntax rule is fulfilled, an instance of a domain-specific metamodel class or attribute (i.e. specified by the name of the right-hand side of the concrete syntax rule) must be created. In other words, notation-specifications in EMFText are tightly coupled to classes and attributes captured by a domain-specific metamodel. Further, similar to Xtext, this approach automates the generation of IDE features, such as syntax highlighting, basic code completion, and reference resolution. Although EMFText features the capability of customizing the appearance of textual tokens, such as the definition of text color, it does so by relying on its own generator implementation to pick up such specifications during editor generation. Therefore, the migration of non-native ANTLR specifications

[15] Further insights are provided in section 3 of the EMFText User Guide located online at https://github.com/DevBoost/EMFText/blob/master/Core/Doc/org.emftext.doc/pdf/EMFTextGuide.pdf.

(i.e. containing additional information, such as the color of textual tokens) to other ANTLR-based language workbench frameworks is rendered strenuous by the requirement to provide the implementation which interprets additional (non-native ANTLR) information based on the API of the target language workbench framework.

MontiCore [39, 135] represents an approach that is based on the generation of (restricted) Ecore metamodels, Java classes, and parsers from EBNF-like grammars (referred to as MontiCore grammars in [135]). In general, this approach and the approach presented in this chapter target language engineers that are most familiar with grammarware and modelware, and initiate the language development process by means of domain-specific grammars and metamodels respectively. The process of language development and, in particular, the construction of concrete and abstract syntax involves significant redundancy. The approach presented in this chapter follows a similar motivation in regards to the independent development of language components and, in particular, to the reduction of redundancy and the improvement of maintenance. Finally, the implementation of MontiCore does not consider the independent modeling of abstract and concrete syntax specifications, such as those represented by domain-specific metamodels and notation-specifications.

A dedicated DSL for the construction of (complex) bridges between grammars and metamodels is presented in [103, 36]. In general, their work differs to the approach presented in this chapter by promoting the construction and maintenance of bridge-specifications between domain-specific metamodels and grammars, as opposed to metamodel-agnostic notations. Thus, effective construction of valid bridges between domain-specific metamodels and grammars requires language engineers to be fully aware of the actions and tool specifications that are available on both sides of individual bridges.

6.8 Summary

This chapter presented an approach for the definition and maintenance of textual notations applicable to domain-specific metamodels and may involve domain-agnostic and domain-specific components. The approach is implemented based on EMF and Xtext, and manifests as a framework, notation-template language, and IDE for textual style definition and editing. Moreover, the approach implementation has been showcased based on the creation of domain metamodel-agnostic notational definitions and, in particular, by comparison with the model-first approach. The approach was evaluated based on a set of popular Xtext-based languages (i.e. obtained from the open-source repository hosting service Github) and indicates that the ECSS framework is integratable to domain-specific languages that are defined by means of Ecore metamodels. Further, ECSS models are approximately 20 and 51 percent more concise by mean and median respectively, than their handcrafted counterparts. Finally, domain-agnostic ECSS models produce language grammars that express the components captured by handcrafted grammars more closely compared to language grammars that are produced by the Xtext metamodel-to-grammar generator.

CHAPTER 7

Conclusion and future work

The previous chapter introduced an approach composed of a language and a framework for the design and application of reusable and extensible notation-template specifications that are structure-agnostic (i.e. agnostic to individual instances of domain-specific metamodels), structure-dependent, or a combination thereof. The following summarizes the contributions of this thesis and presents directions for future work.

7.1 Summary

This thesis presented efforts towards automating the modernization of domain-specific languages through the contribution of a set of approaches and framework implementations (cf. Figure 7.1) that enable the assisted integration of modeling languages from XML Schema definitions, the generation of consistency-achieving IDEs, and the design and application of reusable and extensible notation-template specifications. In what follows, the contributions elaborated in the course of this thesis, and the conclusions drawn from their evaluation are summarized.

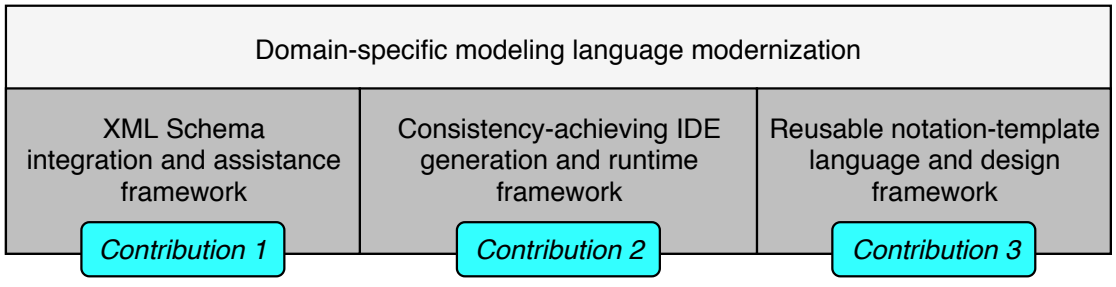

Figure 7.1: Summary of the contributions of this thesis.

Contribution 1: XML Schema integration and assistance. The first part of this contribution is published in [162] and entails the automated migration of textual modeling language implementations from XML Schema definitions, based on the language exploitation design pattern, and a reproduction study in the domain of cloud application topology modeling. This work highlights the limitations in bridging XML-based languages with textual modeling languages, and overcomes them by explicitly representing XML Schema concepts, such as mixed content, wildcards, identifiers and identifier references, and datatypes, in the technical space of modelware. As a result, the representation of important characteristics of XML, such as semi-structured data, is enabled for textual modeling languages, and the respective generation from XML Schema definitions is automated. The approach is bundled into the XMLText framework and evaluated based on completeness, which is measured according to the availability of domain-specific concepts and relationships in the generated language. More specifically, an evaluation that is based on an industrial language and, in particular, the OASIS TOSCA standard [173] has been carried out. The results of this study indicate that the XMLText framework significantly improves existing solutions and generates a respective textual modeling language that is more complete in comparison to the handcrafted Cloudify DSL. The second part of this contribution is published in [192] and entails the integration of modeling assistance for XML Schema descriptions, and the evaluation of XML Schema-based language exploitation involving the eCl@ss standard (i.e. an XML Schema-based industrial specification for classification and product descriptions) [63]. More specifically, the implementation of the approach has been integrated into the Extremo modeling and metamodeling assistant to enable developers to import and query domain-specific knowledge embodied by XMLware artifacts, as well as facilitate them for the construction of novel modeling languages. This integration has been evaluated based on the development of an eCl@ss standard-conforming modeling language implementation and its subsequent application for the modeling of a conveyor-belt production system. The results of this study indicate that: the integration of XMLText requires a limited amount of effort (i.e. five LOC in the case of the Extremo assistant); the integration of XMLText in Extremo is useful for solving practical problems in language engineering; the common data scheme employed by the assistant is able to capture heterogeneous information and, in particular, the import of the SigPML metamodel [46], as well as the XML-based eCl@ss standard specification. Finally, the presented approach for automating the migration of textual modeling languages from XML Schemas has been qualitatively compared with the latest developments in the bridging of the technological spaces of modelware and grammarware, XMLware and modelware, and XMLware and grammarware.

Contribution 2: Consistency-achieving IDE generation and runtime. The first part of this contribution is published in [164] and entails the automated generation of consistency-achieving language implementations from formally constrained language structures alongside a custom approach for the timely mitigation of the state-space explosion problem. More specifically, an approach for leveraging language analysis and

SBSE-techniques for the automated generation and runtime support of advanced IDEs has been presented. As a result, language designers are enabled to employ language definitions that are refined with formal constraints and, in particular, Ecore metamodels and OCL invariants for the automated generation of modeling language IDEs that offer enhanced validation, content-assist, and quick fix solutions. In other words: consistency violations are visualized in a fine-grained manner (i.e. as opposed to visualizing complete sub-structures of a model as erroneous), thus enabling precise localization and eventual repair; content assistance is provided that preserves model consistency; model repair solutions are proposed that increase or fully recover model consistency. Further, generated model repair solutions are sorted according to quality and cost to solicit the visualization of ranked quick fixes to IDE users. The developed custom search approach procures neighborhood populations, escapes local optima solutions, and computes optimal solutions by employing a multi-objective search algorithm that constructs model repair solutions that consist of simultaneous change actions on multiple model features. The second part of this contribution is published in [165] and entails the automated generation of formal constraint specifications from XML Schema data type restrictions, as well as the synthesis of XMLText and IntellEdit into XMLIntellEdit. The implementation of this contribution has been evaluated in a quantitative comparison of IDE tooling generated by the Xtext language workbench, and IDE tooling generated by the implementation of this contribution, relative to the effectiveness of model validation, content-assist, and model repair. The results of this evaluation exhibit: a twofold improvement in validation over existing solutions; content-assist suggestions that do not introduce new constraint violations; model repair solutions that reduce the number of violated constraints by approximately 67%.

Contribution 3: Reusable notation-template language and design. This contribution is published in [163] and entails the development and application of notation-template models that are structure-agnostic, structure-dependent, or a combination thereof. This contribution is represented by an approach for the definition and maintenance of reusable and extensible textual notation-specifications, and implemented by means of the ECSS framework and notation-template language based on the Eclipse Modeling Framework and Xtext framework. This approach and, in particular, ECSS models and their generated grammars have been quantitatively evaluated based on integratability (i.e. applicability of notation-template models to arbitrary domains represented by concepts and relationships in domain-specific metamodels), expressiveness, and conciseness in comparison to both respective grammar specifications created by hand and those generated by the Xtext metamodel-to-grammar generator. The study subjects are represented by a set of popular open-source DSLs that have been obtained from Github. The results of this study indicate that: the framework and, in particular, ECSS models integrate with a set of popular open-source DSLs; structure-agnostic notation-specifications expressed by means of ECSS models are closer to handcrafted languages in comparison to grammars produced by the Xtext metamodel-to-grammar generator; ECSS models are approximately 20% and 51% more concise by mean and

median respectively, than their handcrafted counterparts. Finally, the state of the art of textual notation design has been analyzed in a qualitative comparison with the proposed contribution for the construction and maintenance of textual notations for modeling languages.

7.2 Future Work

This section briefly outlines directions for future work, building on the research conducted in the course of this thesis. These future research directions originate in experiences gained during the implementation of prototypes, as well as from the evaluation results obtained by applying these prototypes in case studies. On the one hand, they concern the identified limitations of the solutions proposed in this thesis and, on the other, research topics furthering efforts towards automating the modernization of domain-specific languages.

Dissemination of ECSS framework evaluation. A line of future work is to disseminate the ECSS framework by extending existing research [163] via the quantitative evaluation presented in Section 6.6. This may prompt research on domain-specific language workbenches that emphasizes the development and application of modular languages through the independent development of abstract and concrete representations of domain-specific concepts and relationships.

Extension of XML Schema data type support. With respect to future work, one direction is to employ the approach implementation to a set of XML Schema definitions that cover each type of data that is defined within the W3C Recommendation on XML Schema data types [28] at least once, eventually addressing any potential gaps in the transformation chain of the presented XML Schema-based language exploitation approach, and extending the validity of existing evaluation results.

Synthesis of XMLIntellEdit and ECSS. Concerning the synthesis of XMLText and IntellEdit into the XMLIntellEdit framework, one line of future work is to merge the capabilities of XMLIntellEdit and ECSS into a common framework. The current implementation of XMLIntellEdit relies on the grammar generation mechanism of Xtext. Therefore, XML documents that are viewed and edited in an XMLIntellEdit-generated IDE must conform to the customized metamodel for XML artifacts that is employed by XMLIntellEdit. The current implementation of ECSS, however, is limited by the support of structural language specifications that are defined using domain-specific Ecore metamodels. Therefore, merging the ability of XMLIntellEdit and ECSS to create customized domain-specific Ecore metamodels from XML Schema definitions, and to generate textual modeling languages from pairs that consist of an XML Schema definition and an ECSS model will extend the set of use cases, and provide a more seamless modernization of XML Schema-based languages with modeling languages and advanced IDEs (i.e. offering enhanced validation, consistency-achieving content-assist,

and model repair). For example, this integration will allow the visualization of arbitrary XMLware artifacts ruled by an ECSS notation-template specification.

An empirical study on the design and engineering of domain-specific languages. An additional direction of future research is to evaluate the impact of the presented approach in regards to both assisted and automated modeling language construction, in comparison to handcrafted language engineering in a set of domains, as well as the involvement of both domain experts and language engineers. This evaluation may also include extending the qualitative analysis of language workbench framework implementations by quantitatively measuring the capability and performance of individual modeling languages created by respective language workbenches, and the contributed framework implementations, particularly in regards to the support for XMLware integration, model validation, content-assistance, model repair, and the design of textual notations. More specifically, the comprehensibility and scalability of ECSS notation-template models may be quantified in comparison to grammar specifications that are fully or partially handcrafted (i.e. the generated grammars are subject to manual modifications) to support language workbench IDEs, such as the Xtext grammar IDE. The results produced by such an evaluation will provide further quantitative and qualitative insights on the performance, scalability, and usability of domain-specific language modernization and the effectiveness of the contributed framework implementations.

UI improvements and quantitative performance analysis. A further line of work includes UI improvements and quantitative performance analysis. The former may include support for syntax-highlighting of Java code in ECSS models, while the latter may include measuring the effect of applying different primitive type conversions (e.g. multiplication, as opposed to min for the and operator). Such UI improvements and analysis will quantify the impact on the performance of generating content-assist and model repair solutions at runtime, and may improve usability.

List of Figures

List of Tables

Acronyms

ANTLR Another Tool For Language Recognition. 19, 20, 34, 35, 46, 47, 53, 58, 117, 125, 168

AST Abstract Syntax Tree. 20, 27, 28, 48, 49, 59, 125

ATL ATLAS Transformation Language. 41, 53

BNF Backus–Naur form. 21, 51

CAML Cloud Application Modeling Language. 97

CFG Context-Free Grammar. 19–21, 23, 34, 40–42, 45–47, 54, 56, 152, 168

CSS Cascading Style Sheets. 6, 175

CST Concrete Syntax Tree. 59

DSL Domain-Specific Language. 1–5, 9, 10, 12, 19, 21, 34, 35, 52, 53, 56, 57, 61, 156–159, 166, 168, 182, 183, 192–194, 197, 200, 201

DSML Domain-Specific Modeling Language. 122, 168

DTD XML Document Type Definition. 13, 14, 38, 43, 117

EBNF Extended Backus–Naur Form. 19–21, 34, 35, 40–42, 45–47, 58, 118, 156, 168, 169, 195, 197

ECSS Ecore Concrete Syntax Specification Language. 82

EMF Eclipse Modeling Framework. 25, 27, 28, 35, 42, 45, 47, 50, 54, 55, 59, 64, 88, 96, 117, 124, 127, 128, 141, 160, 162, 167, 168, 171, 181, 182, 193, 195, 197

EMOF Essential Meta Object Facility. 28, 34, 35, 45, 168

EPML eCl@ss Process Modeling Language. 107

ER Entity Relationship. 43, 57

ETL Epsilon Transformation Language. 41

EVL Epsilon Validation Language. 162

GMF Graphical Modeling Framework. 25, 27, 38, 58

GOPPRR Graph-Object-Property-Port-Role-Relationship. 44

GPL General-Purpose Language. 1, 2, 4, 6, 19, 35, 45, 47, 68, 158, 161

GPML General-Purpose Modeling Language. 27, 83, 122

HOT Higher-Order Transformation. 57

HTML HyperText Markup Language. 175

HUTN Human-Usable Textual Notation. 35, 45, 47, 51, 52, 57, 64, 81

IDE Integrated Development Environment. 2–6, 8, 10, 19, 28, 35, 37, 48, 51, 59, 61, 81, 83, 157, 196, 197, 199, 201–203

ISO International Organization for Standardization. 139, 141

JAXB Java Architecture for XML Binding. 39

JSON JavaScript Object Notation. 56, 175

KM3 Kernel Meta Meta Model. 53

LOC Lines of Code. 185

LOP Language Oriented Programming. 48

MDD Model-Driven Development. 25

MDE Model-Driven Engineering. 2, 3, 9, 24, 25, 35, 47, 52, 54, 58, 83, 84, 152, 168, 173, 193

MDLE Model-Driven Language Engineering. 2, 25, 27, 58, 63, 123, 152

MOF Meta Object Facility. 24, 25, 28, 38, 40, 51, 52, 54, 88, 117, 118

MPS Meta Programming System. 48, 49, 51

OASIS Organization for the Advancement of Structured Information Standards. 96

OCL Object Constraint Language. 6, 31, 34, 35, 59, 65, 89, 90, 110, 122–124, 127–129, 135, 141, 152, 154, 158, 161, 162, 176, 177

OMG Object Management Group. 25, 32

OOP Object-Oriented Programming. 28, 46, 51

ORM Object-role modeling. 42

RAG Reference Attribute Grammar. 45

SBSE Search-based Software Engineering. 37, 124, 153, 163, 201

SDF Syntax Definition Formalism. 45, 47, 55

SQL Structured English Query Language. 19

SVG Scalable Vector Graphics. 56

TCS Textual Concrete Syntax. 46, 53

TOSCA Topology and Orchestration Specification for Cloud Applications. 12, 61, 62, 96, 98, 104, 105, 200, 207

UML Unified Modeling Language. 3, 27, 38, 43, 57, 58, 83, 152

URI Uniform Resource Identifier. 29

W3C World Wide Web Consortium. 2, 13, 14, 64, 88

WYSIWYG What You See Is What You Get. 44

XMI XML Metadata Interchange. 28, 33, 112

XML Extensible Markup Language. 1–3, 5, 6, 8, 13, 35, 38, 42, 61–64, 73, 76, 78, 84–86, 88, 89, 105, 108, 112, 114, 200

XSD XML Schema Definition. 1–6, 8, 10, 12–14, 34, 38, 42, 64, 67, 69, 71, 74, 78, 98

XSD Extensible Stylesheet Language Transformation. 38

YAML YAML Ain't Markup Language. 64, 81, 172, 175

Bibliography

[1] Michael D. Adams. Principled Parsing for Indentation-sensitive Languages: Revisiting Landin's Offside Rule. In *The 40th Annual ACM Sigplan-sigact Symposium on Principles of Programming Languages (POPL), Rome, Italy*, pages 511–522, 2013.

[2] David Aguilera, Cristina Gómez, and Antoni Olivé. Enforcement of Conceptual Schema Quality Issues in Current Integrated Development Environments. In *Proceedings of Advanced Information Systems Engineering - 25th International Conference (CAiSE), Valencia, Spain*, pages 626–640, 2013.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986.

[4] Sethi Ravi Aho A., Lam M. and Ullman J. *Compilers: Principles, Techniques and Tools, 2nd Editio*. Pearson Higher Education, 2007.

[5] Marcus Alanen and Ivan Porres. A Relation Between Context-Free Grammars and Meta Object Facility Metamodels. Technical report, Turku Centre for Computer Science, 2003.

[6] Diego Albuquerque, Bruno Barbieri Pontes Cafeo, Alessandro F. Garcia, Simone Diniz Junqueira Barbosa, Silvia Abrahão, and António Ribeiro. Quantifying USAbility of Domain-specific Languages: An Empirical Study on Software Maintenance. *Journal of Systems and Software*, 101:245–259, 2015.

[7] Shaukat Ali, Muhammad Zohaib Z. Iqbal, Andrea Arcuri, and Lionel C. Briand. A Search-based OCL Constraint Solver for Model-based Test Data Generation. In *Proceedings of the 11th International Conference on Quality Software (QSIC), Madrid, Spain*, pages 41–50, 2011.

[8] Sabharwal Ashish and Selman Bart. S. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, Third Edition. *Artificial Intelligence*, 175(5-6):935–937, 2011.

[9] Colin Atkinson, Bastian Kennel, and Björn Goß. The Level-agnostic Modeling Language. In *Revised Selected Papers of Third International Conference on Software Language Engineering (SLE), Eindhoven, the Netherlands*, pages 266–275, 2010.

[10] Colin Atkinson and Thomas Kühne. Model-Driven Development: A Metamodeling Foundation. *IEEE Software*, 20(5):36–41, 2003.

[11] Colin Atkinson and Thomas Kühne. Reducing Accidental Complexity in Domain Models. *Software and System Modeling*, 7(3):345–359, 2008.

[12] Paolo Atzeni, Paolo Cappellari, and Philip A. Bernstein. Modelgen: Model Independent Schema Translation. In *Proceedings of the 21st International Conference on Data Engineering (ICDE), Tokyo, Japan*, pages 1111–1112, 2005.

[13] Greg J. Badros. JavaML: a markup language for Java source code. *Computer Networks*, 33(1-6):159–177, 2000.

[14] Alessandra Bagnato, Konstantinos Barmpis, Nik Bessis, Luis Adrián Cabrera-Diego, Juri Di Rocco, Davide Di Ruscio, Tamás Gergely, Scott Hansen, Dimitris S. Kolovos, Philippe Krief, Ioannis Korkontzelos, Stéphane Laurière, Jose Manrique Lopez de la Fuente, Pedro Maló, Richard F. Paige, Diomidis Spinellis, Cedric Thomas, and Jurgen J. Vinju. Developer-centric Knowledge Mining from Large Open-source Software Repositories (CROSSMINER). In *STAF Workshops*, volume 10748 of *Lecture Notes in Computer Science*, pages 375–384. Springer, 2017.

[15] Francesco Basciani, Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. Automated Clustering of Metamodel Repositories. In *Proceedings of 28th International Conference on Advanced Information Systems Engineering (CAiSE), Ljubljana, Slovenia*, pages 342–358, 2016.

[16] Bas Basten, Jeroen van den Bos, Mark Hills, Paul Klint, Arnold Lankamp, Bert Lisser, Atze van der Ploeg, Tijs van der Storm, and Jurgen J. Vinju. Modular Language Implementation in Rascal - Experience Report. *Science of Computer Programming*, 114:7–19, 2015.

[17] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. YAML Ain't Markup Language (YAML™), Version 1.1, 2009.

[18] Amine Benelallam, Abel Gómez, Gerson Sunyé, Massimo Tisi, and David Launay. Neo4emf, A Scalable Persistence Layer for EMF Models. In *Proceedings of Modeling Foundations and Applications - 10th European Conference (ECMFA), Held As Part of STAF, York, UK*, pages 230–241, 2014.

[19] Alexander Bergmayr, Javier Troya, Patrick Neubauer, Manuel Wimmer, and Gerti Kappel. Uml-based Cloud Application Modeling with Libraries, Profiles, and Templates. In *Proceedings of the 2nd International Workshop on Model-Driven Engineering on and for the Cloud (CloudMDE) at MoDELS, Valencia, Spain*, pages 56–65, 2014.

[20] Philip A. Bernstein. Applying Model Management to Classical Meta Data Problems. In *Online Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR), Asilomar, Califonia, USA*, 2003.

[21] Jürgen Bettels and F. Avery Bishop. Unicode: A Universal Character Code. *Digital Technical Journal*, 5(3):21–31, 1993.

[22] Lorenzo Bettini. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[23] Jean Bézivin, Frédéric Jouault, and David Touzet. An Introduction to the Atlas Model Management Architecture. *Rapport de recherche*, 5:10–49, 2005.

[24] Jean Bézivin, Ivan Kurtev, et al. Model-based Technology Integration with the Technical Space Concept. In *Metainformatics Symposium*, volume 20, pages 44–49, 2005.

[25] Tobias Binz, Uwe Breitenbücher, Florian Haupt, Oliver Kopp, Frank Leymann, Alexander Nowak, and Sebastian Wagner. Opentosca - A Runtime for Tosca-based Cloud Applications. In *Proceedings of the 11th International Conference on Service-oriented Computing (ICSOC), Berlin, Germany*, pages 692–695, 2013.

[26] Tobias Binz, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. TOSCA: Portable Automated Deployment and Management of Cloud Applications. In *Advanced Web Services*, pages 527–549. 2014.

[27] Linda Bird, Andrew Goodchild, and Terry A. Halpin. Object Role Modeling and XML-schema. In *Proceedings of Conceptual Modeling - ER, 19th International Conference on Conceptual Modeling, Salt Lake City, Utah, USA*, pages 309–322, 2000.

[28] Paul V. Biron, Ashok Malhotra, and World Wide Web Consortium. XML Schema Part 2: Datatypes Second Edition, 2004.

[29] Martin Bjorklund. YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF), 2010.

[30] Alan F. Blackwell, Carol Britton, Anna Louise Cox, Thomas R. G. Green, Corin A. Gurr, Gada F. Kadoda, Maria Kutar, Martin Loomes, Chrystopher L. Nehaniv, Marian Petre, Chris Roast, Chris Roe, Allan Wong, and R. Michael Young. Cognitive Dimensions of Notations: Design Tools for Cognitive Technology. In *Proceedings of the 4th International Conference on Cognitive Technology: Instruments of Mind (CT), Warwick, UK*, pages 325–341, 2001.

[31] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-driven Software Engineering in Practice, Second Edition*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2017.

[32] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/xt 0.17. A Language and Toolset for Program Transformation. *Science of Computer Programming*, 72(1-2):52–70, 2008.

[33] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML). *World Wide Web Journal*, 2(4):27–66, 1997.

[34] Tim Bray, Jean Paoli, C. Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible Markup Language (XML), 1998.

[35] Uwe Breitenbücher, Tobias Binz, Oliver Kopp, Frank Leymann, and David Schumm. Vino4tosca: A Visual Notation for Application Topologies Based on TOSCA. In *Proceedings of On the Move to Meaningful Internet Systems (OTM) at the Confederated International Conferences: CoopIS, DOA-SVI, and ODBASE, Rome, Italy*, pages 416–424, 2012.

[36] Achim D. Brucker, Jordi Cabot, Gwendal Daniel, Martin Gogolla, Adolfo Sánchez-Barbudo Herrera, Frank Hilken, Frédéric Tuong, Edward D. Willink, and Burkhart Wolff. Recent Developments in OCL and Textual Modeling. In *Proceedings of the 16th International Workshop on OCL and Textual Modeling at MoDELS, Saint-malo, France*, pages 157–165, 2016.

[37] Christoff Bürger, Sven Karol, and Christian Wende. Applying Attribute Grammars for Metamodel Semantics. In *Proceedings of the International Workshop on Formalization of Modeling Languages*, page 1, 2010.

[38] Lampson Butler. Personal distributed computing: The alto and ethernet software. Addison-Wesley, 1988.

[39] Arvid Butting, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. Translating Grammars to Accurate Metamodels. In *Proceedings of the 11th ACM Sigplan International Conference on Software Language Engineering (SLE), Boston, Massachusetts, USA*, pages 174–186, 2018.

[40] Jordi Cabot and Martin Gogolla. Object Constraint Language (OCL): A Definitive Guide. In *Advanced Lectures of 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM), Bertinoro, Italy*, pages 58–90, 2012.

[41] Donald D. Chamberlin and Raymond F. Boyce. SEQUEL: A Structured English Query Language. In *Proceedings of 1974 ACM-SIGMOD Workshop on Data Description, Access and Control, Ann Arbor, Michigan, USA*, pages 249–264, 1974.

[42] Philippe Charles, Robert M. Fuhrer, and Stanley M. Sutton Jr. IMP: a Meta-tooling Platform for Creating Language-specific Ides in Eclipse. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE), Atlanta, Georgia, USA*, pages 485–488, 2007.

[43] Sergej Chodarev. Development of Human-friendly Notation for XML-based Languages. In *Proceedings of the Federated Conference on Computer Science and Information Systems (FedCSIS), Gdańsk, Poland*, pages 1565–1571, 2016.

[44] Sergej Chodarev and Jaroslav Porubän. Development of Custom Notation for XML-based Language: A Model-driven Approach. *Computer Science and Information Systems*, 14(3):939–958, 2017.

[45] William D. Clinger and Jonathan Rees. Macros That Work. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida, USA*, pages 155–162, 1991.

[46] Benoît Combemale, Julien DeAntoni, Benoit Baudry, Robert B. France, Jean-Marc Jézéquel, and Jeff Gray. Globalizing modeling languages. *IEEE Computer*, 47(6):68–71, 2014.

[47] Rainer Conrad, Dieter Scheffner, and Johann Christoph Freytag. XML Conceptual Modeling Using UML. In *Proceedings of Conceptual Modeling - ER, 19th International Conference on Conceptual Modeling, Salt Lake City, Utah, USA*, pages 558–571, 2000.

[48] Charles Consel, Fabien Latry, Laurent Réveillère, and Pierre Cointe. A Generative Programming Approach to Developing DSL Compilers. In *Proceedings of the 4th International Conference on Generative Programming and Component Engineering (GPCE), Tallinn, Estonia*, pages 29–46, 2005.

[49] Steve Cook, Gareth Jones, Stuart Kent, and Alan Wills. *Domain-specific Development with Visual Studio Dsl Tools*. Addison-Wesley Professional, first edition, 2007.

[50] James R. Cordy. The TXL Source Transformation Language. *Science of Computer Programming*, 61(3):190–210, 2006.

[51] Microsoft Corporation. Modeling SDK for Visual Studio. Project website: `https://msdn.microsoft.com/en-us/library/bb126413.aspx`.

[52] Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. A Component Model for Model Transformations. *IEEE Transactions on Software Engineering*, 40(11):1042–1060, 2014.

[53] Krzysztof Czarnecki and Michal Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *Proceedings of Generative Programming and Component Engineering, 4th International Conference (GPCE), Tallinn, Estonia*, pages 422–437, 2005.

[54] Alberto Rodrigues Da Silva. Model-driven Engineering: A Survey Supported by the Unified Conceptual Model. *Computer Languages, Systems & Structures*, 43:139–155, 2015.

[55] Marcos Aurélio Almeida Da Silva, Alix Mougenot, Xavier Blanc, and Reda Bendraou. Towards Automated Inconsistency Handling in Design Models. In *Proceedings of Advanced Information Systems Engineering, 22nd International Conference (CAiSE), Hammamet, Tunisia*, pages 348–362, 2010.

[56] Juan De Lara, Esther Guerra, Ruth Cobos, and Jaime Moreno-Llorena. Extending Deep Meta-modeling for Practical Model-driven Engineering. *The Computer Journal*, 57(1):36–58, 2014.

[57] Luís Eduardo de Souza Amorim, Michael J. Steindorfer, Sebastian Erdweg, and Eelco Visser. Declarative Specification of Indentation Rules: A Tooling Perspective on Parsing and Pretty-printing Layout-sensitive Languages. In *Proceedings of the 11th ACM Sigplan International Conference on Software Language Engineering (SLE), Boston, Massachusetts, USA*, pages 3–15, 2018.

[58] Kalyanmoy Deb. Multi-objective Optimization. In *Search Methodologies*, pages 403–449. Springer, 2014.

[59] Thomas Degueule, Benoît Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. Melange: A Meta-language for Modular and Reusable Development of Dsls. In *Proceedings of the ACM Sigplan International Conference on Software Language Engineering (SLE), Pittsburgh, Pennsylvania, USA*, pages 25–36, 2015.

[60] Lukas Diekmann and Laurence Tratt. Eco: A Language Composition Editor. In *Proceedings of the 7th International Conference on Software Language Engineering (SLE), Västerås, Sweden*, pages 82–101, 2014.

[61] Zinovy Diskin, Sahar Kokaly, and Tom Maibaum. Mapping-aware Megamodeling: Design Patterns and Laws. In *Proceedings of 6th International Conference on Software Language Engineering (SLE), Indianapolis, Indiana, USA*, pages 322–343, 2013.

[62] Martin Dougiamas and Peter Taylor. Moodle: Using Learning Communities to Create an Open Source Course Management System. In *Proceedings of the World Conference on Educational Multimedia, Hypermedia and Telecommunications (ED-MEDIA)*, pages 171–178, 2003.

[63] eCl@ss e.V. eCl@ss Standard, Version 9.0, 2014.

[64] Visser Eelco. Stratego: A Language for Program Transformation Based on Rewriting Strategies. In *Proceedings of the 12th International Conference on Rewriting Techniques and Applications RTA, Utrecht, The Netherlands*, pages 357–362, 2001.

[65] Sven Efftinge and Markus Völter. oAW Xtext: A Framework for Textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*, 2006.

[66] Alexander Egyed, Emmanuel Letier, and Anthony Finkelstein. Generating and Evaluating Choices for Fixing Inconsistencies in UML Design Models. In *23rd International Conference on Automated Software Engineering (ASE), L'aquila, Italy*, pages 99–108, 2008.

[67] Sebastian Erdweg, Paolo G. Giarrusso, and Tillmann Rendel. Language Composition Untangled. In *International Workshop on Language Descriptions, Tools, and Applications (LDTA), Tallinn, Estonia*, page 7, 2012.

[68] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. Sugarj: Library-based Syntactic Language Extensibility. In *Proceedings of the 26th Annual ACM Sigplan Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA), Part of SPLASH, Portland, Oregon, USA*, pages 391–406, 2011.

[69] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. The State of the Art in Language Workbenches - Conclusions from the Language Workbench Challenge. In *Proceedings of the International Conference on Software Language Engineering (SLE*, pages 197–217, 2013.

[70] Moritz Eysholdt and Heiko Behrens. Xtext: Implement Your Language Faster Than the Quick and Dirty Way. In *Companion to the 25th Annual ACM Sigplan Conference on Object-oriented Programming, Systems, Languages, and Applications (SPLASH/OOPSLA), Reno/tahoe, Nevada, USA*, pages 307–309, 2010.

[71] Moritz Eysholdt and Johannes Rupprecht. Migrating a Large Modeling Environment from XML/UML to Xtext/gmf. In *Companion to the 25th Annual ACM Sigplan Conference on Object-oriented Programming, Systems, Languages, and Applications (SPLASH/OOPSLA), Reno/tahoe, Nevada, USA*, pages 97–104, 2010.

[72] Jean-Marie Favre. Foundations of Model (driven) (reverse) Engineering : Models - Episode I: Stories of the Fidus Papyrus and of the Solarus. In *Language Engineering for Model-driven Software Development*, 2004.

[73] Jean-Marie Favre. Towards a basic theory to model model driven engineering. In *3rd Uml Workshop in Software Model Engineering*, pages 262–271. Citeseer, 2004.

[74] Jean-Marie Favre. Languages Evolve Too! Changing the Software Time Scale. In *8th International Workshop on Principles of Software Evolution (IWPSE), Lisbon, Portugal*, pages 33–44, 2005.

[75] Jean-Marie Favre, Dragan Gasevic, Ralf Lämmel, and Andreas Winter. Guest Editors' Introduction to the Special Section on Software Language Engineering. *IEEE Transactions on Software Engineering*, 35(6):737–741, 2009.

[76] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

[77] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay A. McCarthy, and Sam Tobin-Hochstadt. The Racket Manifesto. In *1st Summit on Advances in Programming Languages (SNAPL), Asilomar, California, USA*, pages 113–128, 2015.

[78] Daniel Feltey, Spencer P. Florence, Tim Knutson, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. Languages the racket way. *Language Workbench Challenge at the International Conference on Software Language Engineering (SLE)*, 65, 2016.

[79] Joseph Fialli and Sekhar Vajjhala. The java architecture for xml binding (jaxb). *JSR Specification*, 2003.

[80] Matthew Flatt. Creating Languages in Racket. *Communications of the ACM*, 55(1):48–56, 2012.

[81] Martin Fleck, Javier Troya, Marouane Kessentini, Manuel Wimmer, and Bader Alkhazi. Model Transformation Modularization as a Many-Objective Optimization Problem. *IEEE Transactions on Software Engineering*, 43(11):1009–1032, 2017.

[82] Frédéric Fondement, Rémi Schnekenburger, Sébastien Gérard, and Pierre-Alain Muller. Metamodel-aware textual concrete syntax specification. Technical report, 2006.

[83] Eclipse Foundation. Eclipse Simultaneous Releases. Project website: `https://projects.eclipse.org/releases`.

[84] Martin Fowler. Language Workbenches: The Killer-App for Domain Specific Languages? Blog article: `http://www.martinfowler.com/articles/languageWorkbench.html`.

[85] Martin Fowler. *Domain-specific Languages*. Pearson Education, 2010.

[86] Martin Fowler. *Domain-specific Languages*. The Addison-Wesley signature series. Addison-Wesley, 2011.

[87] Ulrich Frank. Multi-perspective Enterprise Modeling: Foundational Concepts, Prospects and Future Research Challenges. *Software and System Modeling*, 13(3):941–962, 2014.

[88] Miguel Garcia and Paul Sentosa. Generation of Eclipse-based IDEs for Custom DSLs. Technical report, Technical report, Software Systems Institute (STS), TU Hamburg-Harburg, Germany, 2007.

[89] Thomas Goldschmidt, Steffen Becker, and Axel Uhl. Classification of Concrete Textual Syntax Mapping Approaches. In *Proceedings of the 4th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA), Berlin, Germany*, pages 169–184, 2008.

[90] Jeff Gray, Sandeep Neema, Juha-Pekka Tolvanen, Aniruddha S. Gokhale, Steven Kelly, and Jonathan Sprinkle. Domain-specific Modeling. In *Handbook of Dynamic System Modeling.* 2007.

[91] Thomas R. G. Green and Marian Petre. USAbility Analysis of Visual Programming Environments: A 'cognitive dimensions' Framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996.

[92] Richard C. Gronback. *Eclipse Modeling Project: A DSL Toolkit.* Pearson Education, 2009.

[93] Object Management Group. Human-USAble Textual Notation Specification, Version 1.0, 2004.

[94] Object Management Group. Unified Modeling Language, Version 2.5, 2015.

[95] Pierre Hansen and Nenad Mladenovic. Variable Neighborhood Search. In *Handbook of Heuristics.*, pages 759–787. 2018.

[96] Mark Harman. The Current State and Future of Search Based Software Engineering. In *Workshop on the Future of Software Engineering (FOSE) at International Conference on Software Engineering (ICSE), Minneapolis, Minnesota, USA*, pages 342–357, 2007.

[97] Elliotte Rusty Harold and W. Scott Means. *XML in a Nutshell.* O'Reilly, 2001.

[98] Görel Hedin. Reference Attributed Grammars. *Informatica (Slovenia)*, 24(3), 2000.

[99] Jan Heering, P. R. H. Hendriks, Paul Klint, and J. Rekers. The Syntax Definition Formalism SDF - Reference Manual. *Sigplan Notices*, 24(11):43–75, 1989.

[100] Ábel Hegedüs, Ákos Horváth, István Ráth, Moisés Castelo Branco, and Dániel Varró. Quick Fix Generation for Modeling Languages. In *IEEE Symposium on Visual Languages and Human-centric Computing (VL/HCC), Pittsburgh, Pennsylvania, USA*, pages 17–24, 2011.

[101] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Derivation and Refinement of Textual Syntax for Models. In *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA), Enschede, the Netherlands*, pages 114–129, 2009.

[102] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Model-based Language Engineering with Emftext. In *Revised Selected Papers of International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE), Braga, Portugal*, pages 322–345, 2011.

[103] Adolfo Sánchez-Barbudo Herrera, Edward D. Willink, and Richard F. Paige. A Domain Specific Transformation Language to Bridge Concrete and Abstract Syntax. In *Proceedings of 9th International Conference on Theory and Practice of Model Transformations (ICMT), Held As Part of STAF, Vienna, Austria*, pages 3–18, 2016.

[104] Alan R. Hevner. Design Science Research. In *Computing Handbook, Third Edition: Information Systems and Information Technology*, pages 22: 1–23. 2014.

[105] Bernhard G. Humm and Ralf S. Engelschall. Language-oriented Programming Via DSL Stacking. In *Proceedings of the Fifth International Conference on Software and Data Technologies (ICSOFT), Volume 2, Athens, Greece*, pages 279–287, 2010.

[106] John Edward Hutchinson, Jon Whittle, and Mark Rouncefield. Model-driven Engineering Practices in Industry: Social, Organizational and Managerial Factors That Lead to Success or Failure. *Science of Computer Programming*, 89:144–161, 2014.

[107] Information and documentation — International Standard Book Number (ISBN). Standard, International Organization for Standardization, London, UK, 2017.

[108] Codes for the representation of names of countries and their subdivisions – Part 1: Country codes. Standard, International Organization for Standardization, Geneva, CH, November 2013.

[109] Information technology – Universal Coded Character Set (UCS). Standard, Japanese Industrial Standards Committee, Tokyo, Japan, 2017.

[110] Information technology – Syntactic metalanguage – Extended BNF. Standard, American National Standards Institute, Washington, United States, 1996.

[111] Javier Luis Cánovas Izquierdo and Jesús García Molina. Extracting Models from Source Code in Software Modernization. *Software and System Modeling*, 13(2):713–734, 2014.

[112] JetBrains. Meta Programming System. Project website: `http://www.jetbrains.com/mps`.

[113] Jean-Marc Jézéquel, Olivier Barais, and Franck Fleurey. Model Driven Language Engineering with Kermeta. In *Revisted Selected Papers of International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE), Braga, Portugal*, pages 201–221, 2009.

[114] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A Model Transformation Tool. *Science of Computer Programming*, 72(1-2):31–39, 2008.

[115] Frédéric Jouault, Jean Bézivin, and Ivan Kurtev. TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE), Portland, Oregon, USA*, pages 249–254, 2006.

[116] Martin Karlsch. A Model-driven Framework For Domain Specific Languages. Master's thesis, Hasso-Plattner-Institute of Software Systems Engineering, 2007.

[117] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design guidelines for domain specific languages. *Computing Research Repository (CoRR)*, abs/1409.2378, 2014.

[118] Lennart C. L. Kats and Eelco Visser. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and Ides. In *Proceedings of the 25th Annual ACM Sigplan Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA), Reno/tahoe, Nevada, USA*, pages 444–463, 2010.

[119] Steven Kelly and Juha-Pekka Tolvanen. *Domain-specific Modeling - Enabling Full Code Generation*. Wiley, 2008.

[120] Amir A. Khwaja and Joseph E. Urban. Syntax-directed Editing Environments: Issues and Features. In *Proceedings of the ACM/SIGAPP Symposium on Applied Computing: States of the Art and Practice SAC), Indianapolis, Indiana, USA*, pages 230–237, 1993.

[121] Anneke Kleppe. *Software Language Engineering: Creating Domain-specific Languages Using Metamodels*. Pearson Education, 2008.

[122] Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an Engineering Discipline for Grammarware. *ACM Transactions on Software Engineering and Methodology*, 14(3):331–380, 2005.

[123] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), Edmonton, Alberta, Canada*, pages 168–177, 2009.

[124] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. The Epsilon Object Language (EOL). In *Proceedings of the Second European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA), Bilbao, Spain*, pages 128–142, 2006.

[125] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. The Epsilon Transformation Language. In *Proceedings of Theory and Practice of Model Transformations, First International Conference (ICMT), Zürich, Switzerland*, pages 46–60, 2008.

[126] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. On the Evolution of OCL for Capturing Structural Constraints in Modeling Languages. In *Rigorous Methods for Software Construction and Analysis, Essays Dedicated to Egon Börger*, pages 204–218, 2009.

[127] Dimitrios S. Kolovos, Louis M. Rose, James R. Williams, Nikolas Drivalos Matragkas, and Richard F. Paige. A Lightweight Approach for Managing XML Documents with MDE Languages. In *Proceedings of the 8th European Conference on Modeling Foundations and Applications (ECMFA), Kgs. Lyngby, Denmark*, pages 118–132, 2012.

[128] Dimitris S. Kolovos, Patrick Neubauer, Konstantinos Barmpis, Nicholas Matragkas, and Richard F. Paige. Crossflow: A Framework for Distributed Mining of Software Repositories. In *Proceedings of the 16th International Conference on Mining Software Repositories (MSR), Montreal, Canada*, pages 155–159. IEEE / ACM, 2019.

[129] Gabriël D. P. Konat, Lennart C. L. Kats, Guido Wachsmuth, and Eelco Visser. Declarative Name Binding and Scope Rules. In *Revised Selected Papers of 5th International Conference on Software Language Engineering (SLE), Dresden, Germany*, pages 311–331, 2012.

[130] Tomaz Kosar, Saso Gaberc, Jeffrey C. Carver, and Marjan Mernik. Program Comprehension of Domain-specific and General-purpose Languages: Replication of a Family of Experiments Using Integrated Development Environments. *Empirical Software Engineering*, 23(5):2734–2763, 2018.

[131] Tomaz Kosar, Pablo E. Martínez López, Pablo Andrés Barrientos, and Marjan Mernik. A Preliminary Study on Various Implementation Approaches of Domain-specific Language. *Information & Software Technology*, 50(5):390–405, 2008.

[132] Tomaz Kosar, Marjan Mernik, and Jeffrey C. Carver. Program Comprehension of Domain-specific and General-purpose Languages: Comparison Using a Family of Experiments. *Empirical Software Engineering*, 17(3):276–304, 2012.

[133] Tomaz Kosar, Marjan Mernik, Matej Crepinsek, Pedro Rangel Henriques, Daniela Carneiro da Cruz, Maria João Varanda Pereira, and Nuno Oliveira. Influence of Domain-specific Notation to Program Understanding. In *Proceedings of the International Multiconference on Computer Science and Information Technology (IMCSIT), Mragowo, Poland*, pages 675–682, 2009.

[134] Tomaz Kosar, Nuno Oliveira, Marjan Mernik, Maria João Varanda Pereira, Matej Crepinsek, Daniela Carneiro da Cruz, and Pedro Rangel Henriques. Comparing General-purpose and Domain-specific Languages: An Empirical Study. *Computer Science and Information Systems*, 7(2):247–264, 2010.

[135] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: A Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer*, 12(5):353–372, 2010.

[136] Thomas Kühne. What Is a Model? In *Language Engineering for Model-driven Software Development*, 2004.

[137] Thomas Kühne. Matters of (meta-)modeling. *Software and System Modeling*, 5(4):369–385, 2006.

[138] Andreas Kunert. Semi-automatic Generation of Metamodels and Models From Grammars and Programs. *Electronic Notes in Theoretical Computer Science*, 211:111–119, 2008.

[139] Ivan Kurtev, Mehmet Aksit, and Jean Bézivin. Technical Spaces: An Initial Appraisal. In *Proceedings of the International Conference on Cooperative Information Systems (CoopIS)*, 2002.

[140] Ivan Kurtev, Jean Bézivin, and Mehmet Aksit. Technological spaces: An initial appraisal. *International Conference on Cooperative Information Systems (CoopIS), Distributed Objects and Applications (DOA)*, 2002, 2002.

[141] Ralf Lämmel. Grammar Adaptation. In *Proceedings of the International Symposium of Formal Methods for Increasing Software Productivity (FME), Berlin, Germany*, pages 550–570, 2001.

[142] Peter J. Landin. The Next 700 Programming Languages. *Communications of the ACM*, 9(3):157–166, 1966.

[143] Vladimir I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. In *Soviet Physics Doklady*, volume 10, pages 707–710, 1966.

[144] Clayton Lewis and Gary Olson. Empirical Studies of Programmers: Second Workshop. chapter Can Principles of Cognition Lower the Barriers to Programming?, pages 248–263. Ablex Publishing Corp., Norwood, New Jersey, USA, 1987.

[145] Alex Loh, Tijs van der Storm, and William R. Cook. Managed Data: Modular Strategies for Data Abstraction. In *ACM Symposium on New Ideas in Programming and Reflections on Software (Onward!), Part of SPLASH, Tucson, Arizona, USA*, pages 179–194, 2012.

[146] David H. Lorenz and Boaz Rosenan. Cedalion: A Language for Language Oriented Programming. In *Proceedings of the 26th Annual ACM Sigplan Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA) at SPLASH, Portland, Oregon, USA*, pages 733–752, 2011.

[147] Murali Mani, Dongwon Lee, and Richard R. Muntz. Semantic Data Modeling Using XML Schemas. In *Proceedings of Conceptual Modeling - ER, 20th International Conference on Conceptual Modeling, Yokohama, Japan*, pages 149–163, 2001.

[148] Mernik Marjan, Zumer Viljem, Lenic Mitja, and Avdicausevic Enis. Implementation of Multiple Attribute Grammar Inheritance in the Tool LISA. *Sigplan Notices*, 34(6):68–75, 1999.

[149] Thomas J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[150] Pierre André Ménard and Sylvie Ratté. Concept Extraction from Business Documents for Software Engineering Projects. *Automated Software Engineering*, 23(4):649–686, 2016.

[151] Tom Mens, Ragnhild Van Der Straeten, and Maja D'Hondt. Detecting and Resolving Model Inconsistencies Using Transformation Dependency Analysis. In *Proceedings of 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS), Genova, Italy*, pages 200–214, 2006.

[152] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and How to Develop Domain-specific Languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[153] Bart Meyers and Hans Vangheluwe. A Framework for Evolution of Modeling Languages. *Science of Computer Programming*, 76(12):1223–1246, 2011.

[154] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Françoise Le Meur. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, 2010.

[155] Daniel L. Moody. The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Transactions on Software Engineering*, 35(6):756–779, 2009.

[156] Pierre-Alain Muller, Franck Fleurey, Frédéric Fondement, Michel Hassenforder, Rémi Schneckenburger, Sébastien Gérard, and Jean-Marc Jézéquel. Model-driven Analysis and Synthesis of Concrete Syntax. In *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems, Genova, Italy*, pages 98–110, 2006.

[157] Pierre-Alain Muller, Frédéric Fondement, Franck Fleurey, Michel Hassenforder, Rémi Schneckenburger, Sébastien Gérard, and Jean-Marc Jézéquel. Model-driven Analysis and Synthesis of Textual Concrete Syntax. *Software and System Modeling*, 7(4):423–441, 2008.

[158] Pierre-Alain Muller, Frédéric Fondement, and Benoît Baudry. *Concrete Syntax Definition For Modeling Languages*. PhD thesis, École Polytechnique Fédérale De Lausanne, 2007.

[159] Pierre-Alain Muller and Michel Hassenforder. HUTN As a Bridge between Modelware and Grammarware-an Experience Report. In *WISME Workshop at Models/uml*, pages 1–10, 2005.

[160] Christian Nentwich, Wolfgang Emmerich, and Anthony Finkelstein. Consistency Management with Repair Actions. In *Proceedings of the 25th International Conference on Software Engineering, Portland, Oregon, USA*, pages 455–464, 2003.

[161] Patrick Neubauer. Towards Model-Driven Software Language Modernization. In *Joint Proceedings of the Doctoral Symposium and Projects Showcase at the Software Technologies: Applications and Foundations (STAF), Vienna, Austria*, pages 11–20, 2016.

[162] Patrick Neubauer, Alexander Bergmayr, Tanja Mayerhofer, Javier Troya, and Manuel Wimmer. XMLText: From XML Schema to Xtext. In *Proceedings of the ACM Sigplan International Conference on Software Language Engineering (SLE), Pittsburgh, Pennsylvania, USA*, pages 71–76, 2015.

[163] Patrick Neubauer, Robert Bill, Dimitris S. Kolovos, Richard F. Paige, and Manuel Wimmer. Reusable Textual Notations for Domain-Specific Languages. In *19th International Workshop in OCL and Textual Modeling (OCL) at IEEE/ACM 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS), Munich, Germany*, pages 67–80, 2019.

[164] Patrick Neubauer, Robert Bill, Tanja Mayerhofer, and Manuel Wimmer. Automated Generation of Consistency-achieving Model Editors. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), Klagenfurt, Austria*, pages 127–137, 2017.

[165] Patrick Neubauer, Robert Bill, and Manuel Wimmer. Modernizing domain-specific languages with XMLText and IntellEdit. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), Klagenfurt, Austria*, pages 565–566, 2017.

[166] Jakob Nielsen. Iterative User-interface Design. *IEEE Computer*, 26(11):32–41, 1993.

[167] Object Management Group (OMG). Object Constraint Language (OCL), Version 2.2, 2010.

[168] Object Management Group (OMG). Meta Object Facility (MOF) Core Specification, Version 2.5.1, 2016.

[169] Ed Ort and Bhakti Mehta. Java architecture for xml binding (jaxb). Technical article: `https://www.oracle.com/technical-resources/articles/javase/jaxb.html`.

[170] Richard F. Paige, Dimitrios S. Kolovos, and Fiona A. C. Polack. Metamodeling for Grammarware Researchers. In *Revised Selected Papers of 5th International Conference on Software Language Engineering (SLE), Dresden, Germany*, pages 64–82, 2012.

[171] Richard F. Paige, Dimitrios S. Kolovos, and Fiona A. C. Polack. A Tutorial on Metamodeling for Grammar Researchers. *Science of Computer Programming*, 96:396–416, 2014.

[172] Richard F. Paige, Dimitrios S. Kolovos, Louis M. Rose, Nikolaos Drivalos, and Fiona A. C. Polack. The Design of a Conceptual Framework and Technical Infrastructure for Model Management Language Engineering. In *14th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), Potsdam, Germany*, pages 162–171, 2009.

[173] Derek Palma and Thomas Spatzier. Topology and Orchestration Specification for Cloud Applications Version 1.0, 2013.

[174] Terence Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[175] Terence John Parr and Russell W. Quong. ANTLR: A Predicated- *LL(k)* Parser Generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[176] Vaclav Pech, Alex Shatalin, and Markus Voelter. Jetbrains MPS As a Tool for Extending Java. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, Stuttgart, Germany*, pages 165–168, 2013.

[177] Juri Di Rocco Phuong T. Nguyen and Davide Di Ruscio. Enabling Heterogeneous Recommendations in OSS Development: What's Done and What's Next in CROSSMINER. In *Proceedings of the Evaluation and Assessment on Software Engineering (EASE), Copenhagen, Denmark*, pages 326–331, 2019.

[178] Andreas Polzer, Daniel Merschen, Goetz Botterweck, Andreas Pleuss, Jacques Thomas, Bernd Hedenetz, and Stefan Kowalewski. Managing Complexity and Variability of a Model-based Embedded Software Product Line. *ISSE*, 8(1):35–49, 2012.

[179] Jaroslav Porubän and Dominik Lakatoš. YAJCo (Yet Another Java Compiler compiler) is a language parser generator based on annotated model. Github project: `https://github.com/kpi-tuke/yajco`.

[180] Alexander Reder and Alexander Egyed. Computing Repair Trees for Resolving Inconsistencies in Design Models. In *IEEE/ACM International Conference on Automated Software Engineering (ASE), Essen, Germany*, pages 220–229, 2012.

[181] Alexander Reder and Alexander Egyed. Determining the Cause of a Design Model Inconsistency. *IEEE Transactions on Software Engineering*, 39(11):1531–1548, 2013.

[182] Martin P. Robillard, Robert J. Walker, and Thomas Zimmermann. Recommendation Systems for Software Engineering. *IEEE Software*, 27(4):80–86, 2010.

[183] Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. Collaborative Repositories in Model-Driven Engineering. *IEEE Software*, 32(3):28–34, 2015.

[184] Juri Di Rocco, Davide Di Ruscio, Hrishikesh Narayanankutty, and Alfonso Pierantonio. Resilience in Sirius Editors: Understanding the Impact of Metamodel Changes. In *Workshop Proceedings of the ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS), Copenhagen, Denmark*, pages 620–630, 2018.

[185] Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona Polack. Constructing Models with the Human-usable Textual Notation. In *Proceedings of 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS), Toulouse, France*, pages 249–263, 2008.

[186] Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona Polack. The Epsilon Generation Language. In *Proceedings of the 4th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA), Berlin, Germany*, pages 1–16, 2008.

[187] Per Runeson and Martin Höst. Guidelines for Conducting and Reporting Case Study Research in Software Engineering. *Empirical Software Engineering*, 14(2):131–164, 2009.

[188] Beatriz A. Sánchez, Konstantinos Barmpis, Patrick Neubauer, Richard F. Paige, and Dimitrios S. Kolovos. Restmule: Enabling Resilient Clients for Remote Apis. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR), Gothenburg, Sweden*, pages 537–541, 2018.

[189] Andrea Schauerhuber, Manuel Wimmer, Elisabeth Kapsammer, Wieland Schwinger, and Werner Retschitzegger. Bridging Webml to Model-driven Engineering: From Document Type Definitions to Meta Object Facility. *IET Software*, 1(3):81–97, 2007.

[190] Markus Scheidgen. Textual Modelling Embedded into Graphical Modelling. In *Proceedings of the 4th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA), Berlin, Germany*, pages 153–168, 2008.

[191] Douglas C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006.

[192] Ángel Mora Segura, Juan de Lara, Patrick Neubauer, and Manuel Wimmer. Automated Modeling Assistance by Integrating Heterogeneous Information Sources. *Computer Languages, Systems & Structures*, 53:90–120, 2018.

[193] Oszkár Semeráth, Ágnes Barta, Ákos Horváth, Zoltán Szatmári, and Dániel Varró. Formal Validation of Domain-specific Languages with Derived Features and Well-formedness Constraints. *Software and System Modeling*, 16(2):357–392, 2017.

[194] Charles Simonyi, Magnus Christerson, and Shane Clifford. Intentional Software. In *Proceedings of the 21th Annual ACM Sigplan Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA), Portland, Oregon, USA*, pages 451–464, 2006.

[195] Kari Smolander, Kalle Lyytinen, Veli-Pekka Tahvanainen, and Pentti Marttiin. Metaedit - A Flexible Graphical Environment for Methodology Modeling. In *International Conference on Advanced Information Systems Engineering (CAiSE)*, pages 168–193, 1991.

[196] Riccardo Solmi. *Whole Platform*. PhD thesis, University of Bologna, 2005.

[197] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition.

[198] Mark Strembeck and Uwe Zdun. An Approach for the Systematic Development of Domain-specific Languages. *Software: Practices and Experiences*, 39(15):1253–1292, 2009.

[199] Gilbert Tekli, Richard Chbeir, and Jacques Fayolle. A Visual Programming Language for XML Manipulation. *Journal of Visual Languages and Computing*, 24(2):110–135, 2013.

[200] Yentl Van Tendeloo, Simon Van Mierlo, Bart Meyers, and Hans Vangheluwe. Concrete Syntax: A Multi-paradigm Modelling Approach. In *Proceedings of the 10th ACM Sigplan International Conference on Software Language Engineering (SLE), Vancouver, Bc, Canada*, pages 182–193, 2017.

[201] Zoltán Theisz and Gergely Mezei. An Algebraic Instantiation Technique Illustrated by Multilevel Design Patterns. In *Proceedings of the 2nd International Workshop on Multi-level Modelling at the 18th International Conference on Model Driven Engineering Languages & Systems, Ottawa, Canada*, pages 53–62, 2015.

[202] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML Schema Part 1: Structures Second Edition, 2004.

[203] Karsten Thoms and Miro Sprönemann. Ecore to Xtext Grammar Creator Xtend transformation (Ecore2XtextGrammarCreator.xtend). Source code located at `https://tinyurl.com/ybon2dvl`.

[204] Massimo Tisi, Frédéric Jouault, Piero Fraternali, Stefano Ceri, and Jean Bézivin. On the Use of Higher-order Model Transformations. In *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA), Enschede, the Netherlands*, pages 18–33, 2009.

[205] Juha-Pekka Tolvanen and Steven Kelly. Defining Domain-specific Modeling Languages to Automate Product Derivation: Collected Experiences. In *Proceedings of the 9th International Conference on Software Product Lines (SPLC), Rennes, France*, pages 198–209, 2005.

[206] Juha-Pekka Tolvanen, Risto Pohjonen, and Steven Kelly. Advanced tooling for domain-specific modeling: MetaEdit+. In *The 7th Oopsla Workshop on Domain-specific Modeling, Finland*, 2007.

[207] Javier Troya, Hugo Brunelière, Martin Fleck, Manuel Wimmer, Leire Orue-Echevarria, and Jesús Gorroñogoitia. ARTIST: Model-based Stairway to the Cloud. In *STAF Projects Showcase*, volume 1400 of *CEUR Workshop Proceedings*, pages 1–8. CEUR-WS.org, 2015.

[208] M. G. J. van den Brand. Prettyprinting without losing comments. 1993.

[209] Tijs van der Storm, William R. Cook, and Alex Loh. Object Grammars. In *Revised Selected Papers of 5th International Conference on Software Language Engineering (SLE), Dresden, Germany*, pages 4–23, 2012.

[210] Arie van Deursen and Paul Klint. Little Languages: Little Maintenance? *Journal of Software Maintenance*, 10(2):75–92, 1998.

[211] Vladimir Viyović, Mirjam Maksimović, and Branko Perisić. Sirius: A rapid development of DSM graphical editor. In *18th International Conference on Intelligent Engineering Systems (INES)*, pages 233–238, 2014.

[212] Markus Voelter. Language and IDE Modularization and Composition with MPS. In *Revised Selected Papers of International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE), Braga, Portugal*, pages 383–430, 2011.

[213] Markus Voelter and Konstantin Solomatov. Language modularization and composition with projectional language workbenches illustrated with MPS. *Software Language Engineering (SLE)*, 16(3), 2010.

[214] Tim A. Wagner. *Practical Algorithms for Incremental Software Development Environments*. PhD thesis, EECS Department, University of California, Berkeley, Mar 1998.

[215] William M. Waite and Gerhard Goos. *Compiler Construction*. Texts and Monographs in Computer Science. Springer, 1984.

[216] Richard S. Wallace. *The Anatomy of Alice*. Springer, 2009.

[217] Edward D. Willink. Re-engineering Eclipse MDT/OCL for Xtext. *Electronic Communication of the European Association of Software Science and Technology (ECEASST)*, 36, 2010.

[218] Manuel Wimmer and Gerhard Kramler. Doctoral Symposium at International Conference on Model Driven Engineering Languages and Systems (MODELS), Montego Bay, Jamaica, Revised Selected Papers. pages 159–168, 2005.

[219] Niklaus Wirth. Extended Backus-Naur form (EBNF). *ISO/IEC*, 14977:2996, 1996.

[220] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, and Björn Regnell. *Experimentation in Software Engineering*. Springer, 2012.

[221] Andreas Zeller. *Why Programs Fail - A Guide to Systematic Debugging, 2nd Edition*. Academic Press, 2009.

[222] Zarko Zivanov, Predrag S. Rakic, and Miroslav Hajdukovic. Using Code Generation Approach in Developing Kiosk Applications. *Computer Science and Information Systems*, 5(1):41–59, 2008.

[223] Steffen Zschaler, Dimitrios S. Kolovos, Nikolaos Drivalos, Richard F. Paige, and Awais Rashid. Domain-specific Metamodeling Languages for Software Language Engineering. In *Revised Selected Papers of the Second International Conference on Software Language Engineering (SLE), Denver, Colorado, USA*, pages 334–353, 2009.

Appendices

Space transportation service language XML Schema definition

Listing 1: Space transportation service language XML Schema definition (spacetransportationservice.xsd).

```
<?xml version="1.0" encoding="UTF8" standalone="no"?>
<xsd:schema xmlns:sts="http://cs.york.ac.uk/ecss/examples/spacetransportationserviceXsdSource" xmlns:xsd="http
    ://www.w3.org/2001/XMLSchema" targetNamespace="http://cs.york.ac.uk/ecss/examples/
    spacetransportationserviceXsdSource">

  <xsd:element name="Spacecraft" type="sts:Spacecraft"/>
  <xsd:element name="LaunchSite" type="sts:LaunchSite"/>
  <xsd:element name="Stage" type="sts:Stage"/>
  <xsd:element name="PhysicalProperty" type="sts:PhysicalProperty"/>
  <xsd:element name="NamedElement" type="sts:NamedElement"/>
  <xsd:element name="EngineType" type="sts:EngineType"/>
  <xsd:element name="SpaceTransportationService" type="sts:SpaceTransportationService"/>

  <xsd:complexType name="Spacecraft">
    <xsd:complexContent>
      <xsd:extension base="sts:NamedElement">
        <xsd:sequence>
          <xsd:element maxOccurs="unbounded" minOccurs="0" name="stages" type="sts:Stage"/>
          <xsd:element maxOccurs="unbounded" minOccurs="0" name="functions" type="sts:Function"/>
          <xsd:element maxOccurs="unbounded" minOccurs="0" name="physicalProperty" type="sts:PhysicalProperty"/>
        </xsd:sequence>
        <xsd:attribute name="spacecraftId" type="xsd:ID" use="required"/>
        <xsd:attribute name="launchSite" type="xsd:IDREFS" use="required"/>
         <xsd:attribute name="manufacturer" type="xsd:string"/>
        <xsd:attribute name="countryOfOrigin" type="xsd:string"/>
        <xsd:attribute name="relaunchCostInMioUSD" type="xsd:integer" use="required"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <xsd:complexType name="LaunchSite">
    <xsd:complexContent>
      <xsd:extension base="sts:NamedElement">
        <xsd:sequence>
          <xsd:element maxOccurs="unbounded" minOccurs="0" name="physicalProperty" type="sts:PhysicalProperty"/>
          <xsd:element name="operator">
```

```
          <xsd:complexType mixed="true">
            <xsd:sequence>
              <xsd:element name="operatorName" type="xsd:string"/>
              <xsd:element name="operatorService" type="xsd:string"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="launchSiteId" type="xsd:ID" use="required"/>
      <xsd:attribute name="locationLatitude" type="xsd:decimal" use="required"/>
      <xsd:attribute name="locationLongitude" type="xsd:decimal" use="required"/>
      <xsd:attribute name="numberOfLaunchpads" type="xsd:integer" use="required"/>
      <xsd:attribute name="operational" type="xsd:boolean" use="required"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="LaunchSchedule">
  <xsd:sequence>
    <xsd:element maxOccurs="unbounded" minOccurs="0" name="launchEvent">
      <xsd:complexType>
        <xsd:attribute name="missionTitle" use="required">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:maxLength value="42"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>
        <xsd:attribute name="launchSiteId" type="xsd:IDREF" use="required"/>
        <xsd:attribute name="spacecraftId" type="xsd:IDREF" use="required"/>
        <xsd:attribute name="startDateTime">
          <xsd:simpleType>
            <xsd:restriction base="xsd:dateTime">
              <xsd:pattern value="\d{4}-\d\d-\d\dT\d\d:\d\d:\d\dZ"/>
              <! requires UTC as time zone (i.e. indicated by Z),
               e.g.: 20040412T13:20:00Z >
            </xsd:restriction>
          </xsd:simpleType>
```

```
        </xsd:attribute>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="Stage">
  <xsd:complexContent>
    <xsd:extension base="sts:NamedElement">
      <xsd:sequence>
        <xsd:element maxOccurs="unbounded" minOccurs="0" name="physicalProperty" type="sts:PhysicalProperty"/>
      </xsd:sequence>
      <xsd:attribute name="engineTypeId" type="xsd:IDREF" use="required"/>
      <xsd:attribute name="engineAmount" type="xsd:integer" use="required"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="PhysicalProperty">
  <xsd:attribute name="type" type="sts:PhysicalPropertyType" use="required"/>
  <xsd:attribute name="unit" type="xsd:string"/>
  <xsd:attribute name="value" type="xsd:decimal" use="required"/>
</xsd:complexType>
<xsd:complexType abstract="true" name="NamedElement">
  <xsd:attribute name="name" type="xsd:string" use="required"/>
</xsd:complexType>

<xsd:simpleType name="Function">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="MARS_COLONIZATION"/>
    <xsd:enumeration value="EARTH_LUNAR_TRANSPORT"/>
    <xsd:enumeration value="MULTIPLANETARY_TRANSPORT"/>
    <xsd:enumeration value="INTERCONTINENTAL_TRANSPORT"/>
    <xsd:enumeration value="ORBITAL_LAUNCHER"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="PhysicalPropertyType">
  <xsd:restriction base="xsd:string">
```

```
      <xsd:enumeration value="LENGTH"/>
      <xsd:enumeration value="WIDTH"/>
      <xsd:enumeration value="DIAMETER"/>
      <xsd:enumeration value="PERIMETER"/>
      <xsd:enumeration value="AREA"/>
      <xsd:enumeration value="VOLUME"/>
      <xsd:enumeration value="MASS"/>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:complexType name="EngineType">
    <xsd:complexContent>
      <xsd:extension base="sts:NamedElement">
        <xsd:attribute name="engineTypeId" type="xsd:ID" use="required"/>
        <xsd:attribute name="fuelKind" type="xsd:string" use="required"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <xsd:complexType name="SpaceTransportationService">
    <xsd:sequence>
      <xsd:element maxOccurs="unbounded" minOccurs="0" name="launchSite" type="sts:LaunchSite"/>
      <xsd:element maxOccurs="unbounded" minOccurs="0" name="engineType" type="sts:EngineType"/>
      <xsd:element maxOccurs="unbounded" minOccurs="0" name="spacecraft" type="sts:Spacecraft"/>
      <xsd:element maxOccurs="unbounded" minOccurs="0" name="launchSchedule" type="sts:LaunchSchedule"/>
    </xsd:sequence>
  </xsd:complexType>

</xsd:schema>
```

Space transportation service language Ecore metamodel

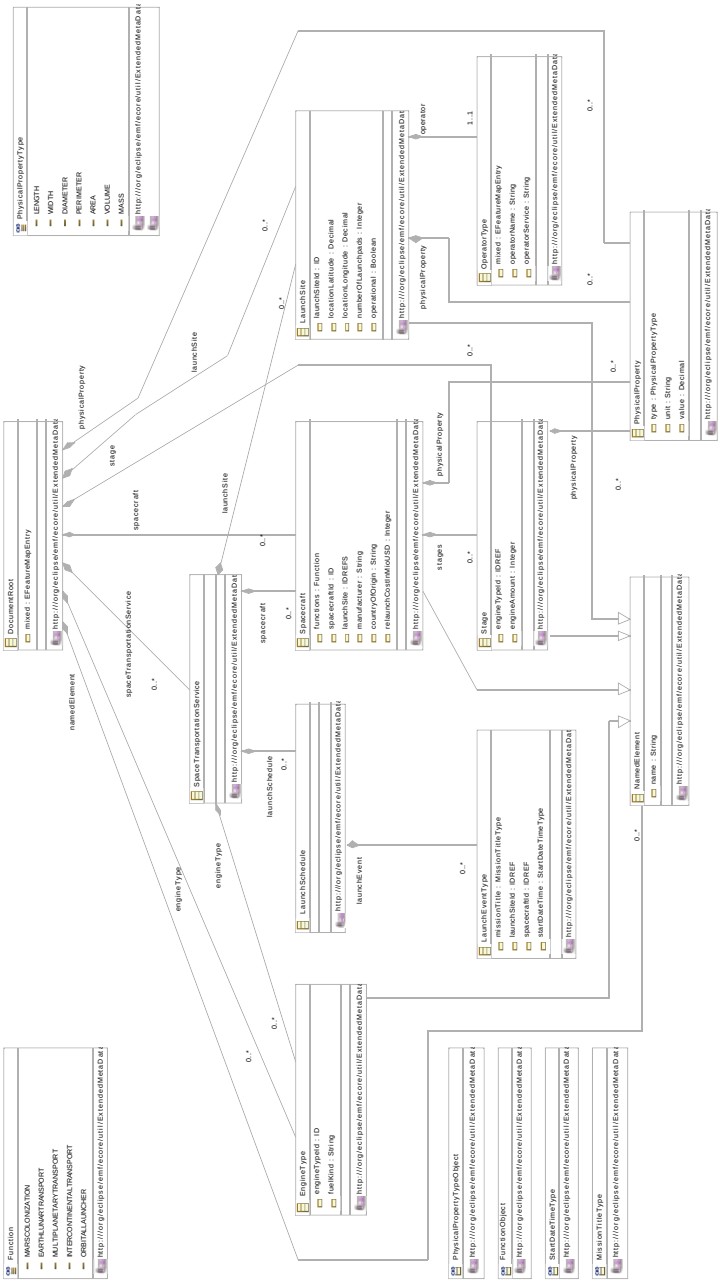

Figure 1: Space transportation service language Ecore metamodel (spacetransportation-service.ecore).

Space transportation service language OCL constraints in Ecore metamodel

Listing 2: Space transportation service language OCL constraints in Ecore metamodel (spacetransportationservice.ecore).

```
import ecore : 'http://www.eclipse.org/emf/2002/Ecore' ;

package spacetransportationservice : sts = 'http://cs.york.ac.uk/ecss/examples/
    spacetransportationservice'
{
   class Spacecraft extends NamedElement
   {
      property stages : Stage[*|1] { ordered composes };
      property launchSites : LaunchSite[*|1] { ordered };
      attribute functions : Function[*|1] { ordered };
      attribute manufacturer : String[?];
      attribute countryOfOrigin : String[?];
      attribute relaunchCostInMioUSD : ecore::EInt[1];
      property physicalProperties : PhysicalProperty[*|1] { ordered composes };
      invariant twoUpperCaseChars:
         countryOfOrigin.size() = 2 and
         countryOfOrigin.toUpperCase() = countryOfOrigin;
   }
   class LaunchSite extends NamedElement
   {
      attribute locationLatitude : ecore::EDouble[1];
      attribute locationLongitude : ecore::EDouble[1];
      attribute operator : OperatorType;
      attribute numberOfLaunchpads : ecore::EInt[1];
      property physicalProperties : PhysicalProperty[*|1] { ordered composes };
      attribute operational : Boolean[1];
      invariant angleDecimal: locationLatitude >= 180 and locationLatitude <= 180;
   }
   class LaunchSchedule
   {
      property launchEvent : LaunchEvent[*|1] { ordered composes };
   }
   class LaunchEvent
   {
      attribute missionTitle : String[?];
      attribute startDateTime : String[1];
      property spacecraft : Spacecraft[1] { ordered };
      property launchSite : LaunchSite[1] { ordered };
      invariant startDateTimeType:
        startDateTime.toString().matches('^\d{4}-\d{2}-\d{2}T\d{2}:\d{2}:\d{2}Z$'
            );
   }
   class Stage extends NamedElement
   {
      property engineType : EngineType[1];
      attribute engineAmount : ecore::EInt[1];
      property physicalProperties : PhysicalProperty[*|1] { ordered composes };
```

```
    }
    class PhysicalProperty
    {
       attribute type : PhysicalPropertyType[1];
       attribute unit : String[?];
       attribute value : ecore::EDouble[1];
    }
    abstract class NamedElement
    {
       attribute name : String[1];
    }
    enum Function { serializable }
    {
       literal MARS_COLONIZATION;
       literal EARTH_LUNAR_TRANSPORT = 1;
       literal MULTIPLANETARY_TRANSPORT = 2;
       literal INTERCONTINENTAL_TRANSPORT = 3;
       literal ORBITAL_LAUNCHER = 4;
    }
    enum PhysicalPropertyType { serializable }
    {
       literal LENGTH;
       literal WIDTH = 1;
       literal DIAMETER = 2;
       literal PERIMETER = 3;
       literal AREA = 4;
       literal VOLUME = 5;
       literal MASS = 6;
    }
    class EngineType extends NamedElement
    {
       attribute fuelKind : String[1];
       invariant lessOrEqualThanMaxChars: fuelKind.size() <= 128;
    }
    class OperatorType
    {
       attribute operatorName : String[1];
       attribute operatorService : String[1];
    }
    class SpaceTransportationService extends NamedElement
    {
       property launchSites : LaunchSite[*|1] { ordered composes };
       property launchSchedule : LaunchSchedule[1] { ordered composes };
       property engineTypes : EngineType[*|1] { ordered composes };
       property spacecrafts : Spacecraft[*|1] { ordered composes };
    }
}
```

Space transportation service default grammar

Listing 3: Space transportation service default grammar.

```
1  // automatically generated by Xtext
2  grammar org.xtext.example.mydsl.MyDsl with org.eclipse.xtext.common.Terminals
3
4  import "http://cs.york.ac.uk/ecss/examples/spacetransportationservice"
5  import "http://www.eclipse.org/emf/2003/XMLType" as type
6  import "http://www.eclipse.org/emf/2002/Ecore" as ecore
7
8  SpaceTransportationService returns SpaceTransportationService:
9     {SpaceTransportationService}
10    'SpaceTransportationService'
11    '{'
12       ('launchSite' '{' launchSite+=LlaunchSite ( "," launchSite+=LaunchSite)* '}'
              )?
13       ('engineType' '{' engineType+=EngineType ( "," engineType+=EngineType)* '}'
              )?
14       ('spacecraft' '{' spacecraft+=Spacecraft ( "," spacecraft+=Spacecraft)* '}'
              )?
15       ('launchSchedule' '{' launchSchedule+=LaunchSchedule ( "," launchSchedule+=
             LaunchSchedule)* '}' )?
16    '}';
17
18
19
20
21
22
23
24
25
26 LaunchSite returns LaunchSite:
27    'LaunchSite'
28    launchSiteId=ID0
29    '{'
30       'name' name=String0
31       'locationLatitude' locationLatitude=Decimal
32       'locationLongitude' locationLongitude=Decimal
33       'numberOfLaunchpads' numberOfLaunchpads=Integer
34       'operational' operational=Boolean
35       ('physicalProperty' '{' physicalProperty+=PhysicalProperty ( ","
             physicalProperty+=PhysicalProperty)* '}' )?
36       'operator' operator=OperatorType
37    '}';
38
39 EngineType returns EngineType:
40    'EngineType'
41    engineTypeId=ID0
42    '{'
43       'name' name=String0
44       'fuelKind' fuelKind=String0
45    '}';
```

```
46
47 Spacecraft returns Spacecraft:
48    'Spacecraft'
49    spacecraftId=ID0
50    '{'
51       'name' name=String0
52       ('functions' '{' functions+=Function ( "," functions+=Function)* '}' )?
53       'launchSite' launchSite=IDREFS
54       ('manufacturer' manufacturer=String0)?
55       ('countryOfOrigin' countryOfOrigin=String0)?
56       'relaunchCostInMioUSD' relaunchCostInMioUSD=Integer
57       ('stages' '{' stages+=Stage ( "," stages+=Stage)* '}' )?
58       ('physicalProperty' '{' physicalProperty+=PhysicalProperty ( ","
             physicalProperty+=PhysicalProperty)* '}' )?
59    '}';
60
61 LaunchSchedule returns LaunchSchedule:
62    {LaunchSchedule}
63    'LaunchSchedule'
64    '{'
65       ('launchEvent' '{' launchEvent+=LaunchEventType ( "," launchEvent+=
             LaunchEventType)* '}' )?
66    '}';
67
68 String0 returns type::String:
69    'String' /* TODO: implement this rule and an appropriate IValueConverter
          */;
70
71 PhysicalProperty returns PhysicalProperty:
72    'PhysicalProperty'
73    '{'
74       'type' type=PhysicalPropertyType
75       ('unit' unit=String0)?
76       'value' value=Decimal
77    '}';
78
79 OperatorType returns OperatorType:
80    {OperatorType}
81    'OperatorType'
82    ;
83
84 ID0 returns type::ID:
85    'ID' /* TODO: implement this rule and an appropriate IValueConverter */;
86
87 Decimal returns type::Decimal:
88    'Decimal' /* TODO: implement this rule and an appropriate IValueConverter
          */;
89
90 Integer returns type::Integer:
91    'Integer' /* TODO: implement this rule and an appropriate IValueConverter
          */;
92
93 Boolean returns type::Boolean:
```

```
94    'Boolean' /* TODO: implement this rule and an appropriate IValueConverter
          */;
95
96  enum PhysicalPropertyType returns PhysicalPropertyType:
97              LENGTH = 'LENGTH' | WIDTH = 'WIDTH' | DIAMETER = 'DIAMETER' |
                    PERIMETER = 'PERIMETER' | AREA = 'AREA' | VOLUME = 'VOLUME' |
                    MASS = 'MASS';
98
99  Stage returns Stage:
100   'Stage'
101   '{'
102      'name' name=String0
103      'engineTypeId' engineTypeId=IDREF
104      'engineAmount' engineAmount=Integer
105      ('physicalProperty' '{' physicalProperty+=PhysicalProperty ( ","
             physicalProperty+=PhysicalProperty)* '}' )?
106   '}';
107
108 enum Function returns Function:
109             MARS_COLONIZATION = 'MARS_COLONIZATION' | EARTH_LUNAR_TRANSPORT = '
                    EARTH_LUNAR_TRANSPORT' | MULTIPLANETARY_TRANSPORT = '
                    MULTIPLANETARY_TRANSPORT' | INTERCONTINENTAL_TRANSPORT = '
                    INTERCONTINENTAL_TRANSPORT' | ORBITAL_LAUNCHER = 'ORBITAL_
                    LAUNCHER';
110
111 IDREFS returns type::IDREFS:
112   'IDREFS' /* TODO: implement this rule and an appropriate IValueConverter
          */;
113
114 IDREF returns type::IDREF:
115   'IDREF' /* TODO: implement this rule and an appropriate IValueConverter */
          ;
116
117 LaunchEventType returns LaunchEventType:
118   'LaunchEventType'
119   '{'
120      'missionTitle' missionTitle=MissionTitleType
121      'launchSiteId' launchSiteId=IDREF
122      'spacecraftId' spacecraftId=IDREF
123      ('startDateTime' startDateTime=StartDateTimeType)?
124   '}';
125
126 MissionTitleType returns MissionTitleType:
127   'MissionTitleType' /* TODO: implement this rule and an appropriate
          IValueConverter */;
128
129 StartDateTimeType returns StartDateTimeType:
130   'StartDateTimeType' /* TODO: implement this rule and an appropriate
          IValueConverter */;
```

Space transportation service ECSS-generated grammar

Listing 4: Space transportation service ECSS-generated grammar.

```
grammar uk.ac.york.cs.ecss.examples.spacetransportationservice.StsLanguage hidden(WS
    , ML_COMMENT, SL_COMMENT)

import "http://cs.york.ac.uk/ecss/examples/spacetransportationservice"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

SpaceTransportationService returns SpaceTransportationService:
   {SpaceTransportationService}
   'SpaceTransportationService' ':'
   BEGIN
      (
         ( 'launchSites' ':' BEGIN launchSites+=LaunchSite ( launchSites+=
             LaunchSite)* END )+ &
         ( 'launchSchedule' ':' BEGIN launchSchedule+=LaunchSchedule END )? &
         ( 'engineTypes' ':' BEGIN engineTypes+=EngineType ( engineTypes+=
             EngineType)* END )? &
         ( 'spacecrafts' ':' BEGIN spacecrafts+=Spacecraft ( spacecrafts+=
             Spacecraft)* END )?
      )
   END
;

LaunchSite returns LaunchSite:
   name=ID ':'
   BEGIN
      (
         ( 'operational' ':' operational=EBoolean ) &
         ( 'locationLatitude' ':' locationLatitude=EDouble)? &
         ( 'locationLongitude' ':' locationLongitude=EDouble)? &
         ( 'operator' ':' operator=OperatorType)? &
         ( 'numberOfLaunchpads' ':' numberOfLaunchpads=EInt)? &
         ( 'physicalProperties' ':' BEGIN physicalProperties+=PhysicalProperty (
             physicalProperties+=PhysicalProperty)* END )?
      )
   END
;

LaunchSchedule returns LaunchSchedule:
   name=ID ':'
   BEGIN
      (
         ( 'launchEvents' ':' BEGIN launchEvents+=LaunchEvent ( launchEvents+=
             LaunchEvent)* END )?
      )
   END
;

LaunchEvent returns LaunchEvent:
   'LaunchEvent' ':'
   BEGIN
```

```
45        (
46           ( 'missionTitle' ':' missionTitle=EString)? &
47           ( 'startDateTime' ':' startDateTime=StartDateTimeType) &
48           ( 'spacecraft' ':' spacecraft=[Spacecraft|EString] ) &
49           ( 'launchSite' ':' launchSite=[LaunchSite|EString] )
50        )
51     END
52 ;
53
54 EngineType returns EngineType:
55     name=ID ':'
56     BEGIN
57        'fuelKind' ':' fuelKind=EString
58     END
59 ;
60
61 Spacecraft returns Spacecraft:
62     name=ID ':'
63     BEGIN
64        (
65           ( 'relaunchCostInMioUSD' ':' relaunchCostInMioUSD=EInt ) &
66           ( 'stages' ':' BEGIN stages+=Stage ( stages+=Stage)* END )? &
67           ( 'launchSites' ':' BEGIN launchSites+=[LaunchSite|EString] ( launchSites
                +=[LaunchSite|EString])* END )? &
68           ( 'functions' ':' BEGIN functions+=Function ( functions+=Function)* END )
                ? &
69           ( 'manufacturer' ':' manufacturer=EString)? &
70           ( 'countryOfOrigin' ':' countryOfOrigin=EString)? &
71           ( 'physicalProperties' ':' BEGIN physicalProperties+=PhysicalProperty (
                physicalProperties+=PhysicalProperty)* END )?
72        )
73     END
74 ;
75
76 EString returns ecore::EString:
77     STRING | ID;
78
79 EDouble returns ecore::EDouble:
80     INT? '.' INT (('E'|'e') INT)?;
81
82 EInt returns ecore::EInt:
83     INT;
84
85 PhysicalProperty returns PhysicalProperty:
86     type=PhysicalPropertyType ':'
87     BEGIN
88        (
89           ( 'value' ':' value=EDouble ) &
90           ( 'unit' ':' unit=EString )?
91        )
92     END
93 ;
94
```

```
95 Stage returns Stage:
96    name=ID ':'
97    BEGIN
98       (
99          ( 'engineAmount' ':' engineAmount=EInt ) &
100         ( 'engineType' ':' engineType=[EngineType|EString] ) &
101         ( 'physicalProperties' ':' BEGIN physicalProperties+=PhysicalProperty (
                physicalProperties+=PhysicalProperty)* END )?
102      )
103   END
104 ;
105
106 enum Function returns Function:
107             MARS_COLONIZATION = 'MARS_COLONIZATION' | EARTH_LUNAR_TRANSPORT = '
                    EARTH_LUNAR_TRANSPORT' | MULTIPLANETARY_TRANSPORT = '
                    MULTIPLANETARY_TRANSPORT' | INTERCONTINENTAL_TRANSPORT = '
                    INTERCONTINENTAL_TRANSPORT' | ORBITAL_LAUNCHER = 'ORBITAL_
                    LAUNCHER';
108
109 EBoolean returns ecore::EBoolean:
110    'true' | 'false';
111
112 enum PhysicalPropertyType returns PhysicalPropertyType:
113             LENGTH = 'LENGTH' | WIDTH = 'WIDTH' | DIAMETER = 'DIAMETER' |
                    PERIMETER = 'PERIMETER' | AREA = 'AREA' | VOLUME = 'VOLUME' |
                    MASS = 'MASS';
114
115 OperatorType returns OperatorType:
116    BEGIN
117       (
118          ( 'operatorName' ':' operatorname=ID ) &
119          ( 'operatorService' ':' operatorService=EString ) &
120          ( any+=AnyGenericElement ( any+=AnyGenericElement)* )?
121       )
122    END
123    ;
124
125 StartDateTimeType returns StartDateTimeType:
126    STRING
127 ;
128
129 AnyGenericConstruct returns AnyGenericConstruct:
130    AnyGenericElement | AnyGenericText;
131
132 AnyGenericElement returns AnyGenericElement:
133    {AnyGenericElement}
134    BEGIN
135       ( elemName=STRING)?
136       (':' elemValue=STRING)?
137       ( anyGenericAttr+=AnyGenericAttribute (
138          anyGenericAttr+=AnyGenericAttribute )* )?
139       ( anyGenericElement+=AnyGenericElement (
140          anyGenericElement+=AnyGenericElement )* )?
```

```
141     END
142     ;
143
144 AnyGenericAttribute returns AnyGenericAttribute:
145     attrName=STRING ':'
146     attrValue=STRING
147 ;
148
149 AnyGenericText returns AnyGenericText:
150     {AnyGenericText}
151     ( textValue=STRING)?
152 ;
153
154 // start: Terminals.xtext
155 terminal ID:
156     '^'? ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
157
158 terminal INT returns ecore::EInt: ('0'..'9')+;
159
160 terminal STRING:
161           '"' ( '\\' . /* ('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\') */ | !('\\'|'
                  "') )* '"'? |
162           "'" ( '\\' . /* ('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\') */ | !('\\'|"
                  '") )* "'"?;
163
164 terminal ML_COMMENT: '/*' > '*/';
165
166 terminal SL_COMMENT: '//' !('\n'|'\r')* ('\r'? '\n')?;
167
168 terminal WS: (' '|'\t'|'\r'|'\n')+;
169
170 terminal ANY_OTHER: .;
171 // end: Terminals.xtext
172
173 // The following synthetic tokens are used for the indentationaware blocks
174 terminal BEGIN: 'synthetic:BEGIN';  // increase indentation
175 terminal END: 'synthetic:END';  // decrease indentation
```

Default notation-template model

Listing 5: Default notation-template model (default.ecss).

```
template attributeTemplate : uk.ac.york.cs.ecss.newproc.AttributeXtendRule;
template singleAttributeTemplate : uk.ac.york.cs.ecss.newproc.
    SingleAttributeXtendRule;
template classTemplate : uk.ac.york.cs.ecss.newproc.AbstractClassBasedXtendRule;
templateGen attributeGenTemplate extends attributeTemplate;
templateGen singleAttributeGenTemplate extends singleAttributeTemplate;
templateGen classGenTemplate extends classTemplate;

rule defaultClassRule: classGenTemplate :: classRules:
   [%= class.name %]" returns " [%= class.name %] ":"
   [% if (loc_subClasses.isEmpty()) { %]
      "{" [%= class.name %] "}" "'" [%= class.name %]"'" [%= booleanDistRules(
          optional[ 0 .. 99]) %] [%= nameDistRules(name[ 0 .. 1]) %] "'{'"
         [%= attributeDistRules(other[ 0 .. 99]) %]
      "'}'"
   [% } else { %]
   for subCl: loc_subClasses join " | " {
         ::classRules(subCl)
   }
   [% } %] ';'
;

rule defaultAttributeDistr: attributeGenTemplate :: attributeDistRules:
   for esf: features {
      [%= attributeRule(esf) %]
   }
;

rule arbitraryAttributeDistr: attributeGenTemplate :: attributeDistRules:
   "("
   for esf: features join ") | (" {
      [%= attributeRule(esf) %]
      [% List<EStructuralFeature> smallerFeatures = new ArrayList(features); %]
      [% smallerFeatures.remove(esf);\nboolean first = true; %]
      "(("
      for EStructuralFeature sub: smallerFeatures join ") & (" {
         "',' [%= attributeRule(sub) %]
      } "))"
   } ")"
;

rule booleanAttributeDistr: attributeGenTemplate :: booleanDistRules:
   for esf: features {
      "(" [%= esf.getName() %] "?= \"" [%= esf.getName() %] "\")?"
   }
;

rule nameAttributeDistr: attributeGenTemplate :: nameDistRules:
   for esf: features {
      "(" [%= esf.getName() %] " = ID)?"
```

```
49    }
50 ;
51
52 rule defaultAttribute: singleAttributeGenTemplate :: attributeRule:
53    [% if (feature.getLowerBound() == 0) {" "(" [% } %]
54       " " [%= feature.name %] " " [%= xtextOperator %] " " [%=
              sensibleTerminal(feature) %] " "
55    [% if (feature.isMany()) { %]
56       "( ',' " [%= feature.name %] " " [%= xtextOperator %] " " [%=
              sensibleTerminal(feature) %] ")*"
57    [% } %]
58    [% if (feature.getLowerBound() == 0) {" ")?" [% } %]
59 ;
60
61 rule defaultTerminal: singleAttributeGenTemplate :: sensibleTerminal:
62    [% if (feature instanceof EReference) { %]
63       [% if (((EReference)feature).isContainment()) { %]
64          //Just the class name
65          ::classRules("feature.getEType()")
66       [% } else { %]
67          //A reference to the class name
68          "[" ::classRules("feature.getEType()") "]"
69       [% } %]
70    [% } else { if (feature.getEType() instanceof EEnum) { %]
71       //Some terminal just take the etype
72       ::enumRules("feature.getEType()")
73    [% } else { %]
74       ::terminalRules("feature.getEType()")
75    [% }} %]
76 ;
77
78 * {
79    slot(name, name): 2.0;
80    slot(*,other): 0.5;
81    slot(*["(not many) and ( eType.name = 'Boolean' or eType.name = 'boolean')
          "],optional): 2.0;
82    template(arbitraryAttributeDistr): 1.5;
83    template(defaultAttributeDistr): 1.5;
84    template(attributeRule): 0.5;
85    //template(attributeRule): 1.5;
86    a: "out";
87    featname: "eigensch";
88    classname: ocl "rule.class.name";
89 }
```

Example space transportation service model

Listing 6: Example space transportation service model.

```
SpaceTransportationService:
   launchSites:
      KennedySpaceCenter:
         operator:
            operatorName: "NASA"
            operatorService: "public-sector service"
         operational: true
         numberOfLaunchpads: 3
         locationLatitude: 28.524058
         locationLongitude: 80.65085

   engineTypes:
      Merlin1D:
         fuelKind: "Subcooled LOX / Chilled RP1"
      Merlin1DVacuum:
         fuelKind: "LOX / RP1"

   spacecrafts:
      FalconHeavy:
         functions:
            ORBITAL_LAUNCHER
         manufacturer: "SpaceY"
         countryOfOrigin: "USA"
         relaunchCostInMioUSD 90
         launchSites: KennedySpaceCenter

         stages:
            FirstStage:
               engineAmount 9
               engineType: Merlin1D
            SecondStage:
               engineAmount 1
               engineType: Merlin1DVacuum

         physicalProperties:
            LENGTH:
               unit m
               value 70.0
            DIAMETER:
               unit m
               value 3.66
            WIDTH:
               unit m
               value 12.2
            MASS:
               unit kg
               value 1420788.0

   launchSchedule:
      LaunchEvent:
```

```
          missionTitle: "GPS III03 navigation satellite deployment"
          startDateTime: "20200131T12:00:00Z"
          spacecraft: FalconHeavy
          launchSite: KennedySpaceCenter
       LaunchEvent:
          missionTitle: "AFSPC44 payload deployment (classified)"
          startDateTime: "20200930T12:00:00Z"
          spacecraft: FalconHeavy
          launchSite: KennedySpaceCenter
```

IntellEdit integration with XMLText

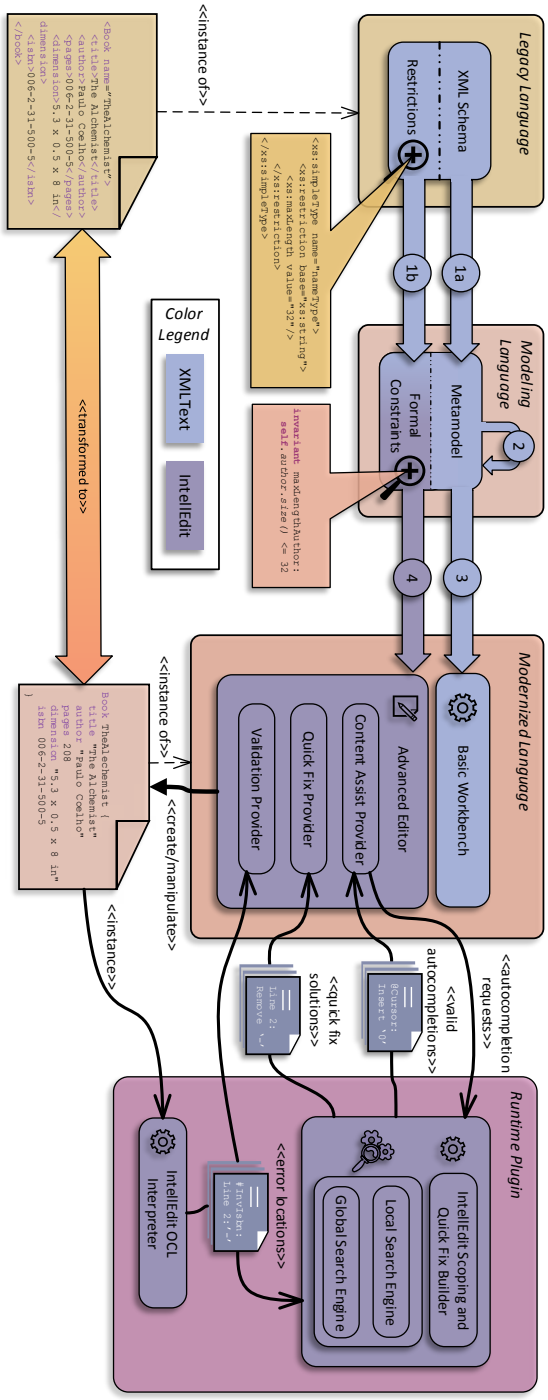

Figure 2: IntellEdit integration with XMLText

Library language XML Schema definition

Listing 7: Library language XML Schema definition (library.xsd).

```
<?xml version="1.0" encoding="UTF8" standalone="no"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <! definition of simple types >
  <xs:simpleType name="nameType">
    <xs:restriction base="xs:string">
      <xs:maxLength value="32"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="sinceType">
    <xs:restriction base="xs:date"/>
  </xs:simpleType>
  <xs:simpleType name="isbnType">
     <xs:restriction base="xs:string">
      <xs:pattern value="[09]{3}[09]{2}[09]{4}[09]{3}[09]"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="dimensionType">
     <xs:restriction base="xs:string">
      <xs:pattern value="([09]|([19][09]+))([.][09]+)? x ([09]|([19][09]+))
          ([.][09]+)? x ([09]|([19][09]+))([.][09]+)? (centimeters|cm|in|inches
          )"/>
    </xs:restriction>
  </xs:simpleType>
  <! definition of complex types >
  <xs:complexType name="libraryType">
    <xs:sequence>
       <xs:element maxOccurs="unbounded" minOccurs="0" name="book" type="
           bookType"/>
       <xs:element maxOccurs="unbounded" minOccurs="0" name="customer" type="
           customerType"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="bookType">
    <xs:sequence>
      <xs:element name="name" type="xs:ID"/>
      <xs:element name="title" type="xs:string"/>
      <xs:element name="author" type="nameType"/>
      <xs:element name="pages" type="xs:int"/>
      <xs:element name="dimension" type="dimensionType"/>
      <xs:element minOccurs="0" name="bookInfo" type="bookInfoType"/>
    </xs:sequence>
    <xs:attribute name="isbn" type="isbnType" use="required"/>
  </xs:complexType>
  <xs:complexType name="noType">
    <xs:sequence>
      <xs:element name="name" type="xs:ID"/>
    </xs:sequence>
    <xs:attribute name="isbn" type="isbnType" use="required"/>
  </xs:complexType>
  <xs:complexType name="customerType">
```

```
    <xs:sequence>
       <xs:element name="firstName" type="xs:string"/>
       <xs:element name="lastName" type="xs:string"/>
       <xs:element name="borrowedBookId" type="xs:IDREF" minOccurs="0"/>
       <xs:element name="borrowedBookSince" type="sinceType" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="bookInfoType">
    <xs:sequence>
       <xs:any maxOccurs="unbounded" namespace="##any" processContents="lax"/>
    </xs:sequence>
  </xs:complexType>
   <! root element >
  <xs:element name="library" type="libraryType"/>
</xs:schema>
```