UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4: Informatik

Linguistic architecture on the workbench

Comprehensive editing features and pluggable semantics for the *MegaL* notation

Bachelorarbeit

zum Erlangen des Grades eines Bachelor of Science in Informatik, vorgelegt von

Lukas Härtel

Erstgutachter: Prof.Dr. Ralf Lämmel
Institut für Informatik

Zweitgutachter: Marcel Heinz
Institut für Informatik

Koblenz, 20. Oktober 2015

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

.....

(Ort, Datum) (Unterschrift)

Abstract

Linguistic architecture complements the existing forms of modeling a software system by focusing on the exercised languages, the used technologies and the defined artifacts, as well as the relationships they participate in. In that sense *megamodeling* is used, viewing the given entities as models. In previous work, the *MegaL* language has been introduced in syntax and semantics [20, 32], it provides a platform for megamodeling. Developing models in that language is not aided by an editor.

In this thesis, strong tool support based on robust frameworks is laid out. The existing syntax will be extended adding features for modularity and reusability, the model and its configuration are unified, inference of facts is supported by a *reasoner* framework, and a method to address *in-memory* objects in a model will be showcased. These features – provided by the developed *Xtext* based and *Eclipse* integrated editor – are demonstrated in the course of a *case study* on *JAXB*, a technology for *Java*/*XML* binding.

Zusammenfassung

Linguistische Architektur bietet einen komplementären Ansatz zur Softwaremodellierung, indem der Fokus auf den verwendeten Programmiersprachen, den benutzten Technologien und der definierten Artefakte liegt, sowie auf deren Beziehungen untereinander. Für diese Art der Architektur wird Megamodellierung verwendet, die obengenannten Dinge werden als *Modelle* angesehen. In vorausgehenden Arbeiten wurde *MegaL* mit ihrer Syntax und der Semantik als Modellierungssprache eingeführt [20, 32]. Erstellung von Architekturmodellen ist darin jedoch nicht durch einen Editor unterstützt.

In dieser Bachelorarbeit wird ein umfangreiches Werkzeug illustriert, basierend auf robusten Frameworks. Die bisherige Syntax der Sprache wird erweitert, dadurch entstehen neue Funktionen wie Modularität und Wiederverwendbarkeit, das Modell und dessen Konfiguration werden vereint, Ableitung neuer Fakten wird durch ein *Reasoner* Framework ermöglicht, und Methoden um Objekte zur Laufzeit eines Programms zu erfassen werden dargestellt. Diese Funktionen – gegeben durch einen *Xtext* basierten und in *Eclipse* integrierten Editor – werden anhand einer Fallstudie über *JAXB*, einer Technologie zur *Java*/*XML* Datenbindung, vorgeführt.

Contents

List of Figures

List of Tables

List of Listings

Acknowledgments

Thanks to the *Softlang* working group for assistance in writing this implementation and for the discussions on *MegaL*, to Andrei Varanovich who helped me setup the language and to Ralf Lämmel who has put much effort into sorting out all the problems we encountered while making *MegaL* happen. My special thanks go to Philipp Schuster who helped me with an external computer scientist's view on megamodeling and on my implementation, and for being supportive when I thought my endeavor had outgrown me.

I want to thank my family, especially my brother, for supporting me during the writing phase and offering their help whenever they can.

1 Introduction

In software engineering, the aspect of modeling a system is key to maintainable and understandable products. There are many kinds of architecture that engineer a system with regards to the requirements it should fulfill and the structure it is implemented in. In addition, the descriptive nature alleviates reuse and makes formal analysis possible [39]. The context of our modeling implementation will be introduced in the following subsections.

Megamodeling and Linguistic Architecture As a complement to the classic approaches to software architecture, we talk about megamodeling as the view of a software system that describes models and their relationships – take for example instance-to-schema conformance or model transformation [18].

Such models have found their way in a wide variety of research fields. This includes management of models and repositories [12, 29], implementation of architecture frameworks based on model relationships [26]. In addition, there are applications focused on metamodels and transformations as relationships[18, 48], some work deals with maintaining properties of models and their elements in transformation processes [42, 27, 34] and others apply megamodeling to implement Big-Data systems [14] and to improve quality of industry software [4].

This thesis will deal with another form of megamodeling, in that we will look at the *linguistic architecture* of a system. As most of modern day software systems use a multitude of languages, concepts and technologies in their implementations, understanding the interplay of their constituents is challenging. Linguistic architecture aims to describe relations of that kind in a well-structured way by using an ontology tailored to the domain, expressing relationships in systems of such entities [20].

Take for example a responsive website: it is based on the concept of a web server to be available in the Internet, the actual *technology* it uses is *Apache*. It comes with native support for the *language* *PHP*, which is translated into *HTML* and carries *JavaScript* to execute programs on the client side, i.e., the viewer's browser. To apply a custom flavor to the looks of, *CSS* is used as a static scripting language. There might be additional technologies used for the implementation of responsiveness and a number of programs to maintain the site.

Transients The existing instances of megamodeling – if they are concerned with resources – are addressing files or sites on the web. When working

with linguistic architecture, we sometimes encounter use cases, where talking about an artifact that does *not* have a manifestation is required. That could for example be reasoning about objects in an object-oriented language are the intermediate result of a concatenated function that is applied to a file and written to another. Take for example $r = (f \circ g)(x)$, where the intermediate result would be $r_i = g(x)$. If r_i bears some properties that we want to talk about in our linguistic model, a way of addressing it – even if it is never manifested as a file – is required. In the course of this work, such an artifact will be referred to as a *Transient*.

Domain specific languages Linguistic architecture can thereby be leveraged to 'read' a software system with regards to where languages have been applied and what concepts are used and exercised — such an architecture however requires a proper notation. Domain specific Languages (DSLs)[24] have been a continued field of research for many years, their primary use case is to focus a programming language on a problem domain, in order to hide complexity of a general-purpose programming language (GPL) and to expose domain specific knowledge for analysis [31]. DSLs have shown to improve quality of models and to help creating them [46], in addition to their beneficial impact on language maintainability [45, 30].

They can be implemented as an *embedded* DSL, which means that they are integrated into a GPL as an API – the project *Guice*[1] by *Google* provides an embedded DSL for configuring a dependency injection framework, their project *Guava*[2] is rich in embedded DSLs for common programming tasks like transformation and filtering of sequences. Another way of DSL implementation is to integrate custom syntax and semantics into a GPL. This is for example exercised in [22], where a number of DSLs is integrated into *Java*. They can however be implemented as standalone languages: there are workbenches that provide support for tool creation (tooling) in the domain of language development [9, 3, 28]. Such workbenches make the DSLs part of Integrated Development Environments (IDEs) in that they – for instance – generate a code editor and provide feedback for statically analyzed language constraint violations.

Linguistic architecture with *MegaL* There have already been some approaches to engineer a language fit for describing linguistic architecture: *MegaL* has been introduced in [20] and its development has been continued at the *Software Languages* department at the *University of Koblenz*. It bears

[1] `https://github.com/google/guice`
[2] `https://github.com/google/guava`

some similarities to the *RDF*[33], as is focused on entities, their relationships, a type lattice and resource bindings. Entity declarations, subtyping and binding symbols to local- or web-resources are however abbreviated, as they will be a core part of a linguistic architecture description, and as the *RDF* syntax is very verbose in that concerns.

Built atop the syntactical definition of *MegaL* is its evaluation. In [32], the execution of a megamodel concerned with linguistic architecture is described. Based on the bindings, resources can be identified. All relationships are checked for validity, e.g., being element of a language is checked by *acceptors*, API referral is inspected, and tool use is examined. As the variety of use-cases in linguistic architecture is rather large, such checks are delegated to *plugins*. In that fashion, domain specific knowledge can be included without obstructing it for reuse.

This implementation is used as a foundation to this thesis. It is an instance of DSL engineering, but it is only partly developed with regard to IDE support. The *ANTLR*[38] parser generator has been used to create the language syntax, the semantics has then been implemented on top of the parsed model. Bindings and plugins has not been integrated into the language, but has rather been delegated to an embedded DSL written in JSON[15]. Modularity, as in creating components of megamodels and reusing them, is not implemented.

A *MegaL* example In order to illustrate the syntax of *MegaL* and the domain of linguistic architecture, the prior example of the responsive website will be notated (in parts) following the *MegaL* format. A subtype, that is, a more specific type b that still is of type a is notated as $b < a$, an instance x of such a type is given as $x : b$, the relationship of that instance to another y is written x r y, where r denotes what relationship they are in.

```
// Programs are executable artifacts
Program < Artifact

// This is the technology that was mentioned
Apache : Technology

// The server, it is part of the technology
ApacheServer : Program
ApacheServer partOf Apache
```

Technology and server program are listed now. A specific subtype for programs was introduced to indicate that it is a runnable artifact. The relationship of server to technology was defined. Now, the linguistics will be

illustrated.

```
// A subset of the named languages
PHP : Language
HTML : Language

// Our main landing page for the website
mainPage : File
mainPage elementOf PHP

// The code sent over the wire to the user
renderedContent : Artifact
renderedContent elementOf HTML
```

The basics of linguistic architecture are laid out now, they are however not giving away more that the existence *HTML* and *PHP*, but their interplay in the described system. In the following fragment, their cooperation will be highlighted.

```
// It was mentioned that the server tanslates PHP into HTML. Formally
// defined as a transformation function with appropriate domain and range
PHPTransformation : PHP → HTML

// When it's applied to the main page, it generates
// code that a browser may render
PHPTransformation(mainPage) ↦ renderedContent

// The function needs to be implemented, in fact, it is a program
// that is part of the server
PHPProcessor : Program
PHPProcessor defines PHPTransformation
PHPProcessor partOf ApacheServer
```

This fragment exposes a few special constructs, one that defines a function and another that specifies an instances of its in- and output. In addition to the *PHP* transformation's definition, its location is described: it resides in a program that is part of the *Apache HTTP* server. These relationships cover the aspects of linguistics on the instance, language and transformation level, as well as the technologies that group them. The syntax is straightforward and will be adapted in great portions.

1.1 Research questions

Arising from the problem domain description and concerning the previous version of *MegaL*, a set of research questions are hereafter listed.

RQ1: How can we improve editing experience for *MegaL*?

In the present state, the implementation serves as a proof of concept approach, which has been evaluated by the previous work [20, 32]. As we have workbenches for DSL engineering, comprehensive support for the language is thereby within reach. Features such as content proposal, auto-completion of names, as well as structural refactoring are beneficial for productivity. Our solution uses the features *Xtext* provides to implement these. Moser et al. for example evaluate the productivity of small teams using refactoring tools [36].

RQ2: How can we effectively provide integration with other tools?

Plugging into an IDE has usually the benefit of having a multitude of tools at hand that work out of the box if you give them an artifact conforming to a specific notation. While there are several ways of making our models accessible, for example, by providing an XML or JSON interface, we are basing our tool on the *EMF* infrastructure. We thereby benefit from a series of tools, including structural editing of the AST [44], difference algorithms comparing the structure as opposed to the syntax [13], and integration with a graphical editor [47].

RQ3: How can we improve the evaluation of a *MegaL* model?

As defined in previous work, the interpretation of a linguistic architecture is given as a composition of well-formedness, correctness and completeness [32]. In that paper, the semantics are defined in a way that checking a relationship or an entity could only see the singleton instance alone and judge if it is valid in that constellation. A relationship, for example, was checked by a function only capable of seeing the left and right bindings as URIs. In the presented implementation, checking is able to traverse relationships in the model, allowing checkers to use encoded facts for their evaluation. In addition to this, we propose an API for such plugin implementations, it is formed around common use cases in linguistic architecture evaluation. Aside from that evaluation of a relationship can generate any number of output messages as a contrast to a `true`/`false` judgment.

RQ4: How can the evaluation of a *MegaL* model be plugged?

As linguistic description is highly domain specific, semantics for the evaluation of a megamodel needs to be configurable by domain experts. Configuration should leverage the modularity features to be exported and reused, e.g., a *language* and its *acceptor* could be both part of a module. An 'artifact element of language' statement could then be checked without needing extra configuration. In our implementation, we use a segment of the ontology to configure the execution: 'plugin' entities engage in relationships with others and are bound to executing code, the entirety of configured plugins constitutes the model semantics.

In the course of answering these questions, the following contributions will be made:

Contributions

1. The *MegaL* language will be remodeled, semantics configuration is now part of the entity and relationship model, eliminating the need for an external configuration model.

2. Megamodels will be made modular by introducing a first-class module and import structure, thereby enabling reuse of existing models, including the configuration. Orchestration is managed through rebinding of symbols, i.e., overwriting the resource binding.

3. Resource binding will be extended, so that transient artifacts can be addressed. This will be done by allowing binding not only to resources, but also to applications of static functions. For some systems, this requires accessibility adaption, but it fit for most use cases.

4. The *Xtext* framework is instrumented to implement editor that can be deployed to the *Eclipse* IDE. This includes (i) integration with other model-ware based on the *Eclipse Modeling Framework*, (ii) direct feedback of static analysis, i.e., type checking, (iii) feedback for the dynamic execution, that is, checking relationship validity through 'plugins', (iv) use of the resource framework provided by *Eclipse*, and (v) directly integrating with *Java* code for plugin development.

Roadmap §2 illustrates the technologies that are uses for the implementation. §3 gives an overview on the field of domain specific languages and megamodeling. §4 describes the observed requirements and lists them. §5 models the software according to the defined requirements and provides an

abstract view o the implementation. §7 analyses the implemented system for its validity and usability. §8 closes the paper in summary, reflections and future work.

2 Background

A variety of technologies is used in the showcased implementation. They are based on the open source IDE *Eclipse*. An illustration of the underlying programs and framework will be given hereafter.

2.1 *Eclipse*

The program *Eclipse* is a *Java* based open-source integrated development environment [17]. It has a strong focus on extensibility, provided are abstractions for text based editors, hooks for the menus and shortcuts, integration with the file-system abstraction and much more. The build system governing that process can be patched so that at a given time, an own compiler can be triggered. With given features it makes for a perfect candidate of implementing an editor for a syntactically simple language such as the one presented in this thesis.

2.1.1 Plugins

The implementation of *Eclipse* is built on top of the *OSGi* service platform, which provides component-oriented architecture of *bundles*, *services* and *dependencies* [37]. It utilizes the *Java Virtual Machine* and maintains lifecycles of the bundles, which themselves are implemented as *Java* classes — in plain *OSGi*, they are subtypes of BundleActivator, in the case of *Eclipse*, a bundle would derive from the Plugin class.

Bundles are managed as components. They can provide an API in the form of an interface. If another bundle requires an instance of this interface, the providing class is instantiated and returned. To initialize a bundle, all its dependencies need to be satisfied, that is, an instance needs to be present before constructing the bundle. Therefore, the dependencies span a *directed, acyclic graph*. As there are some use cases, where components mutually require each other, a method is provided that allows injecting a dependency *after* the requiring bundle has been created, thereby bridging dependencies to *directed graphs*.

Aside from the component and lifecycle management, *Eclipse* provides the mechanism of *extension points*. As IDEs tend to support a lot of tools in one place, instantiating all of them would take too long. *Eclipse* does therefore delay the initialization of its components as far as possible, only instantiating them when needed. To decide what plugins need to be started, they must provide information about what purpose they serve. In *Eclipse*, *extension points* are used, their format is defined in an *XSD* schema, their

implementations are notated in *XML*. For example, there are bindings from file extensions to editors, which allow starting the editor program only if a conforming file is opened. Other extension point implementations are concerned with controlling the debugger: as this function is only required if a program is running, the debugger can lie dormant while programming.

2.1.2 *JDT* and the Classpath

The *Java Virtual Machine* is configured to run code that resides in a *classpath*. This is a collection of classes that are used to resolve the types required when starting a program. Given for example the collection of types $C = \{a, b, c\}$ as a classpath. If a referred to the type x, which is not present in C, a NoClassDefFoundError would arise. Otherwise, instances of types in C can easily be instantiated.

In addition, the *JVM* supports hierarchical composition of a classpath, i.e., given $C_1 = \{a, b, c, d\}$ and $C_2 = \{e, f\}$, a 'composition' $C = C_2 \circ C_1$ is possible. In that manner, a type residing in C_2 can be instantiated and assigned to a reference from a type in C_1. If both C_1 and C_2 provided for example an implementation for x, instances from individual classpaths would be incompatible, composition however is able to bridge between the two classpaths, hierarchy specifies the precedence. This may seem trivial, but this mechanism can be used while in runtime, a class in C_1 could in fact compose C itself, thereby dynamically loading code during execution.

The *Java Development Tools* are a core part of *Eclipse*, they provide the infrastructure for compiling *Java* code, mechanisms for indexing and searching it, a structural model of *Java* that supports manipulation and a context aware content proposal [1]. The compiler component is able to translate *Java* code into class-files incrementally, it reacts to file changes immediately, patching the classpath of *Java*-based projects. In combination with hierarchical composition of a classpath, the incremental compiler is an essential ingredient of the implementation this thesis presents.

2.2 *Eclipse Modeling Framework*

The *Eclipse Modeling Framework* – or *EMF* for short – is a *Java*, *XML* and *Eclipse* based modelware framework. It provides an infrastructure for tools based on structured models [44]. For example, a new model kind can be defined by specifying a metamodel, editors can then be generated and a representation of the model in *Java* classes can be derived. Users of *EMF* benefit from a rich tool structure. Reflective editing, i.e., a form editor for any *EMF* based metamodel, can simply be used to modify a model. It is integrated

with tools of graphical representation and editing, take – for example – *Sirius*, which allows defining viewports for domain specific graphical modeling [47].

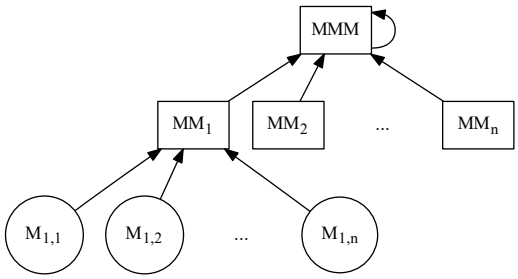

Fig. 1: The hierarchy of *EMF* models. MMM depicts the meta-metamodel. In *EMF*, this is the *Ecore* model, which itself is denoted in *Ecore*. Any user-defined metamodel MM_i conforms to the meta-metamodel. Of a single metamodel MM_i there may be many instances $M_{i,j}$, the models.

There is the *Ecore* part of *EMF*, it defines a structure of top level meta-metamodel, which conforms to itself in a form of 'bootstrapping', there are newly defined metamodels based on it and the latter may have any number of conforming instances, as depicted in Fig. 1.

The metamodels supply a type lattice for *classes* and *interfaces*. They may expose attributes of some primitive types (like text or numbers) and cross references to other classes and interfaces. *EMF* refers to both attributes and references as *structural features*. In addition to properties, it allows specifying *operations*, a pendent to functions or methods. Its generator creates *Java* code that supports a number of usual task like traversal and observation, as well as automatic maintenance of bidirectional references. It comes with an out of the box serialization and deserialization API. Evaluation of an edited document can be done while editing, error annotation is done by annotating the AST.

2.3 *Xtext*

As mentioned in the introduction, *Xtext* is a DSL workbench. It takes a abstract and concrete syntax definition – in the forms of an *EMF* metamodel and a grammar file respectively – to generate an editor plugin for *Eclipse*. The concrete syntax definition is similar to the one *ANTLR* uses: A set of non-terminals and terminals define a grammar. In *Xtext*, the construction of the *EMF* AST is interspersed with *attribute assignments* and constructors. In addition, cross references can be assigned.

2.3.1 Example *Xtext* grammar

In the following fragment, a simple *Xtext* grammar will be showcased. Let's create a new one called 'Library', reusing existing terminals like STRING for text or INT taking numbers, but also rules accepting *C*-style comments. We will also import the AST model – *EMF* binds those to unique resource identifiers – that will export the elements Library, Author and Book.

```
grammar library with xtext.Terminals
import "http://library.example.com/library/1.0"
```

The first rule is always the entry rule of the grammar, as defined per convention of *Xtext*. Our library will consist of some authors and books. We define that they can be interspersed, and that they can occur in any cardinality. Upon being parsed, authors and books are put into corresponding children of the Librarys AST node.

```
Library:
  (authors+=Author | books+=Book)*;
```

We now define an author, it will be given a name and an age. The first statement that the rule lists is however the 'author' keyword — we want to make the fact of defining a new author more prominent, as our documents would otherwise hard to read.

```
Author:
  'author' name=STRING 'born' 'in' age=INT;
```

There's one assignment that should be noted: the attribute 'name' has a special meaning in *Xtext* grammars. *EMF* does not represent a tree, but a graph. therefore, cross references need to be represented somehow. When having a class with an attribute 'name', it immediately becomes referable in *Xtext*, as demonstrated hereafter.

```
Book:
  title=STRING 'published' 'in' published=INT 'by' author=[Author|STRING];
```

Similar to the author, we define a book. The last assignment is special, it says that 'author' is a cross reference assignment, the type is Author, as given by the left side of the pipe, and the actual instance of the author is identified by a STRING.

An example document is given hereafters, it initializes a small library system which conforms to the defined grammar.

```
/**
  * He also wrote a lot of short stories
  */
author "Charles Dickens" born in 1812
```

```
"A Christmas Carol" published in 1843 by "Charles Dickens"

"War and Peace" published in 1869 by "Leo Tolstoy"
"Anna Karenina" published in 1878 by "Leo Tolstoy"

author "Leo Tolstoy" born in 1828
```

2.3.2 *Xtext* features

Xtext provides the integration of a *structural editor* into the host environment *Eclipse*. That means, editing is performed with awareness of the document's underlying structure. Interpreting a code file in this sense has benefits: if a program like a compiler or a source code analyzer needs to comprehend a document, it will most likely not need the excess whitespace and keyword information. Detaching *concrete* and *abstract syntax* also allows changes to a languages grammar, which is helpful when dealing with ambiguous lexing or parsing rules. With *Xtext* editors, the underlying model – or *abstract syntax tree* – is *EMF*. An *ANTLR* parser is generated from a grammar specification, it coordinates the model construction. There's a number of features coming with *Xtext*, some are given hereafter.

1. When a grammar for an *EMF* metamodel is defined, instances models thereof can be serialized. The formatter is preconfigured to generate text conforming to the grammar, it can however be customized to place indentation, new-lines and spaces appropriately.

2. Parallel renaming is automatically provided. The origin of a referred element can be tracked, and all its references can be enumerated. That way, renaming can be applied not only at the currently highlighted instance, but at every location the name is referred to. As the references are semantical, equally named items of other types are not changed.

3. Documenting a model element is a common practice, take for instance *JavaDoc*, which is applied to classes, fields, methods, parameters et cetera. Documentation is here notated in the comment, which usually is ignored in the lexical analysis. *Xtext* is aware of the fact that comments before an element's definition are describing it and associates them. In the previous grammar, commenting on author or a book adds information that would be displayed when hovering above the them, as shown in the example.

4. Integration into *Eclipse*'s build system is alleviated, the meta-information in grammar and configuration are used to automatically generate the

according classes and instantiating the responsible *Eclipse* *extension points*.

5. Syntax highlighting can be configured. By default, the 'keywords' are purple, comments are green, and strings are blue, but associating other lexical rules with different colors is possible. On top of the simple highlighting, the actual model can be used to compute the style. This can be used for conveying semantic knowledge through the definition of special styles — 'instance' and 'static' functions are in *Java* for example distinguished by their font shape.

3 Related work

There's some related work in the context of tool support for megamodeling, tool support in general, integration of editors in aggregations of tools, implementations of text based editors and evaluation – static and dynamic – of domain specific languages. The following paragraphs will give a deeper understanding of this thesis' context, starting off with a more in-depth discussion of the previous implementation.

3.1 Previous *MegaL* implementation

The implementation presented in the thesis is loosely based on the one described in [32]. The grammar for this project is developed according to the previous one and the example files shipped in the existing implementation. This includes language constructs that were later rejected in the course of feature discussion, e.g., the parameter entities given by a prefixed questionmark were unified with the regular ones. We figured out that exercising a linguistic architecture on a system would not let parameters open, and for developing a *truly* parametric model, the holes would have to be tested for at least one instance, thereby already binding it to an example file. In our approach, rebinding is supported as a key to modularity.

The configuration is implemented as an external *JSON* file. For one, it links the entities mentioned in the file. Our version delegates linking to a language-internal construct rather than an external configuration, highlighting the fact that system integration rather than conceptual description is the aim.

In addition to entities, the configuration also binds *checkers* and *reasoners*. The prior are responsible for model validation, checking entities and relationships for appropriateness. The latter provide a mechanism of automatically retrieving information that is encoded as domain knowledge — the project repository of [32] for example provides a RailsResolver, which is aware of the naming conventions in the technology *Rails* and can bind entities appropriately.

In our implementation, both concepts are present in a similar fashion, they are used to implement the semantics and to translate the coded information into manifested data, but are in contrast linked directly in the model. Additionally, our reasoning phase is extended so that reasoners can in fact infer whole model segments including new entities, relationships and types. In short, they are not limited to the bindings. As a final extension to the implementation, reasoners can be dependent of the results that others provide. We refrain from formulating an extensive syntax to analyze dependencies by

applying the reasoners recursively until identity, that is, until their results do not change the model.

The former implementation used parts of the *Java* language – generics in concrete – to associate checkers and reasoners to actual *Java* code. While this has some benefits, like direct association with the type hierarchy, there are some drawbacks that we avoided by explicit linking. For example, including the model into *Java* requires mirroring the entire megamodel type hierarchy, the classes would be purely for type inheritance and not carrying extra information. An alternative to avoiding it would be automatic generation of code files, that however would impede execution of megamodels in portable systems or environments without an appropriate compiler.

Regarding a topic of this thesis – implementing editing features – the existing version is only set up for non-editor development. The process is run on the *MegaL* files themselves and reports the issues tracked. On the opposite, our approach focuses on direct integration within an IDE, and immediate feedback.

Integrated tools In the paper by Sloane et al.[43], a development environment is explained that – in contrast to the more prominent IDEs – is focused on *desintegration*. In their approach, they use a lightweight message protocol carried over network pipes. The latter are abstracted by *ZeroMQ*, which is as technology for automatic management of robust message queues. Formatting, refactoring and compiling is done by small participating programs that collaborate on the state of an artifact. The content proposal mechanisms are implemented by sharing the current selection of an editor window, allowing other contributers to publish potential content. The exchange format can be extended by providing additional services. Seeing that our implementation of *MegaL* will be using a lightweight evaluation engine that can be run in standalone mode, integration with such a system is thinkable. Our parser, which at the moment is provided by *Xtext*, could be deployed as an *ANTLR* parser [38], that way enabling outlining, plotting and other use cases.

The authors of [19] survey the state of language workbenches, which are used to implement one's domain specific or general purpose programming languages. We do rely on *Xtext* as a language workbench, but consider *MegaL* to be one *itself*. Such a claim can be made, as we do provide a fairly open syntax and our evaluation is highly pluggable — there are however problems with the notation of nested structure such as if-clauses or other kinds of loops. As we focus on a graph-like representation, nesting is not an issue for us at the moment.

Megamodel tool support The domain for megamodel tool support is highly distributed, as 'megamodeling' can refer to a variety of architecture and model kinds. However, the tools that are in focus of the past research is applicable to our domain in some degrees. Take for example the work of [27]. In their approach, different languages (DSLs) are woven by a top level model, a pure megamodel thereby. Making the domain specific languages, which are by their nature not graspable in an organized and generalized fashion, program-readable is performed by providing an explicit access method which conforms to a specification. As the presented system is also implemented in *Eclipse*, the access API is specified by an *extension point*. The system ships a set of user interfaces that can also be extended in a similar way. Concerning the features, our system is – by not restricting the access method – universally applicable, every model that can be accessed by some object oriented code is viable. Our interface is in it's current state not more developed than the editor integration, extension point oriented user interfaces are however a track that could be pursued.

In the works of [26], a similar system is described, in that it is concerned with models and relationships between them — for instance, there's model to metamodel conformance. Checking the relationships that are proposed in a system is performed by *OCL*. The *Object Constraint Language* is in fact a domain specific language for model checking. Evaluating the constraints is performable because the megamodel aggregating it's components can point to elements in the models directly. Thereby, *OCL* can navigate the references during constraint validation. In our approach, implementation of such a model access is also possible, we would need a plugin that makes the structure visible and is easy to define. However, most of the time we are dealing with checks that do not consider the involved artifacts as models in the sense that Hillard et al. do. Some of our checks are for instance implemented as calls to appropriate *Java* APIs, like built in *XML* schema conformance.

Comparing our implementation to [27] and [26], which are both build on top of the *AM3* system, we can say that they take benefits from the rigid conditions put on the models. They do gain a strong traceability support. But, as already mentioned, rigidity can impede the generality of a system, and a less constrained framework can still implement the more strict.

A tool that is also based on *AM3* is presented in [29]. They deliver *MoScript*, which serves as a domain specific language for model repository scripts. Their execution is based on an existing model transformation engine, the proposed system translates the queries into code for that engine. In our approach, the implementation of model semantics is not dependent on code generation. We require a plugin that conforms a minimum interface for validity checks and for reasoning. *MoScript* is modeled as a domain specific

language, the implementation is developed in an alternative to *XText*, namely the *Textual Concrete Syntax* environment, further reading on that topic can be delegated to [11]. In our model execution phase, we plan to use a lightweight query script. It's implementation could also be used to answer questions about megamodels.

The topic of traceability is often tied to inter-model element relations. In [42], maintenance on such links is performed during an *Eclipse* build cycle. For this thesis, a case study has been implemented that involves traceability between two internal artifact structures. Our reasoning phase can only extend given models, we can thereby not remove model elements which would be crucial for model traceability maintenance.

Behavior and semantics in DSLs *Xtext* provides mechanisms for direct generation of code from the AST of a domain specific language. That way, defining the semantics of a newly developed language is delegated to an already known language. In [16], the authors present a language for *adaptive distributed applications*. It is based on *JOLIE*, which is focused on the concept of microservices [35], in that it is both extending it's syntax and – by using it as the target of code generation – it's execution engine. In contrast to extending and existing program language by introducing new syntactic constructs and translating them into the original syntax, there are approaches not based on existing syntax. In [40], the authors showcase a language for workflow management. It's execution is performed by generating *Xtend* code, which is similar to *Java*. They are however – in contrast to the previous implementation in [16] – not extending it's syntax.

In our implementation, code generation facilities are not used. Our evaluation is distributed on lots of mostly intrinsic units, i.e., the entities and relationships, which all have to be evaluated in order to execute our model. There are no control flow structures that we could generate from the code, as no execution is conditional and we do not require evaluation in any specified order. Another reason from neglecting code generation is the fact that our execution is defined as *model validity*, we inject our evaluation into the validation phase of *Xtext* to implement the desired behavior.

Another variant of implementing functionality beyond structured data in *EMF* ASTs is to extend the model with functions that can be implemented afterwards. Such methods are defined in the *EMF* metamodel by giving an 'operation' to a model element. When code generation is triggered, the operations are translated into *Java* functions that developers can fill in later — their original implementation is a *stub* that terminates execution in an error. When the operation is implemented, developers will mark the method with

an *annotation*, as to avoid code generation erasing the code and overwrite it with the stub again [44]. We refrain from using this feature, as integrating into large sets of boilerplate code – arising from code generation – is cumbersome and less intuitive than using small and concise units of code. Aside from that, our implementation uses a more triple-oriented structure to represent the data. It alleviates construction and comprehension of models; a detailed discussion will be given in §5.

In addition to fine grained and highly integrated evaluation of *EMF* models, there is also a more structured way of defining validity. The *OCL* has already been identified as being used in [26]. Maintenance of such constraints in *EMF* can be performed with some *OCL* implementation, an instance of this usage is shown in [2]. We wanted to rely on a more intuitive and popular language, as we expect that plugins for *MegaL* will be developed by regular developers. *JVM* based languages seem to be widely spread and therefore are our pick for the constraint definition.

A survey of methods to define static semantics is given in [10]. There are *Java* based approaches, as the one we use for relationship-applicability checking. But there are also implementations in *XSemantics* and *XTS*. The prior is a very general judgment based system that executes deductions rules, the latter is similar but directly aims at type system implementations.

Model comprehension In addition to evaluation of *EMF* models, there are more concepts that need to be mentioned. Our implementation is in parts focused on the access of graph structures — take for instance a lookup of the type hierarchy or checking if one artifact is connected with another one in any way. As our models tend to not exceed a certain level of graphical complexity and size, our lookups are naïvely performed by simple graph algorithms. A system that provides higher performance and would allow for larger graphs is the *EMF-IncQuery* project [7, 6]. As the name suggests, incremental queries are the focus of this work. Another feature is the incremental maintenance of validity constraints. Queries can exceed their use of data providers and trigger – if they are satisfied in any constellation of the graph – an error, that is annotated to the *EMF* model. We would like to benefit from such possibilities, but as our plugin language is *JVM* based and contains great proportions of domain specific knowledge not just concerned with graph lookups, integrating with *IncQuery* is not an option.

4 Requirements

The requirements for *MegaL*s next implementation were retrieved during discussion with users and developers of the previous *MegaL* iterations. Desired features and potential usage scenarios were conceived. They will be elaborated in this section, for they are a foundation of design and implementation.

4.1 Terminology

There have already been some terms that were used in the introduction, background and discussion of related work, they are explained through their usage and the context. However, for the requirements we would like to clarify them explicitly.

Workspace: Our implementation is based on *Eclipse* and its resource architecture. The *workspace* is populated with projects, which are folders conforming to a specific format — usually, the files .project and .classpath are sufficient to define the nature and structure of a *Java* based project.

Workspace relative: This term is used to say that any notation describing resource structure needs to include the *project* in its description. The workspace location itself is not required to identify a resource.

JDT: The *Eclipse*-internal *Java* compiler and associated development tools. Referral to this technology is usually in the context of including user-written code, that is compiled with the *JDT* core component.

Megamodel: The notion of megamodels that has been illustrated in the introduction. It consists of various entities, relationships between those, types of entities, and relationship applicability specification. Entities can be subject to a binding.

Entity: An entity indeed — it describes anything from software system, developing person, concept, format, artifact and the likes. It is typed and has an appropriate, symbolic name.

Entity type: As the name suggest, this element is used to type an entity. It is the actual representation for the before-mentioned list of entity kinds. It consists of a symbolic name and a reference to its supertype, thereby forming a type lattice. The root type is named 'Entity'.

Relationship: Used to connect two entities. It does have a type, which specifies the kind of relations the entities have. Given for example two algebraic sets A and B. If one would like to convey that one is the subset of another, it would be notated as A *subsetOf* B. Relationships can only

applied once for a pair of entities, cardinality is not taken into account.

Relationship type: This is the specification of applicable relationships between a pair of entities, according to the types they have. With regards to the previous definition, the relationship would be listed as $subsetOf < Set \times Set$. Applicability is not limited to the types that are given, but also to their subtypes. Let there for example be a type $Language < Set$. An entity of that kind could also be used as a valid participant of a *subsetOf* relationship

Binding: Given a megamodel based on a number of entities that are purely symbolic. Connecting them to the actual artifacts or locations – in the sense of linked data – is performed by bindings. A binding is attached to an entity by specifying an *unique resource identifier* or short: URI. Artifacts in a local project that is described are given in *workspace relative* notation.

Reasoning: A preparation of the megamodel. To account for encoded knowledge, the reasoning phase can use deduction rules to generate new model elements of any kind.

Evaluation: Facts in the megamodel need to be checked for their validity. We refer to that as *evaluation*, usually not concerned with a model in its entirety but applied on a single model element. The most prominent evaluations are relationship checks.

Plugin: With regard to both *reasoning* and *evaluation*, the executing code is given in the way of linking a plugin. They are defined as functions on megamodel elements and are given a context in which they can navigate the megamodel, generate appropriate response messages and return new model elements in reasoning.

4.2 Scope

The general scope has already been given in the introduction. We are *modeling systems* with a special interest in the *linguistics*, i.e., what languages and technologies thereon are used. In addition to concrete manifestations of artifacts like files, websites and folders, a non-manifested level is included: when talking about a system in that manner, the paradigms and concepts are also of interest. Such entities usually do not have a manifested representation in the system itself, but are documented somewhere on the web.

In particular, we want to improve the existing tools for the previous implementations of *MegaL*. As the given versions were executable but did not have any editor support, this thesis will give the approach that tackles this shortcoming with an integrated editor.

4.3 Stakeholders

There are groups with certain relevance to *MegaL*. In this section, we will list the most important ones as stakeholders.

4.3.1 Researchers (SH-R)

In the context of surveys, teaching and automatized evaluation of software architectures, megamodels can play an important role to researchers. They are therefore listed as stakeholders.

Concern Researchers could use the system as a documentation on something they want to assess, it should therefore be accessible (i) to them in a human-readable form and (ii) to programs they use for automatic analysis.

Relevance Seeing that our focus on linguistic architecture is not as established as other forms of megamodeling, alleviating the use of this architecture form is beneficial to our cause. Access of models should therefore be easy to the research community. In general, notation of a software system can alleviate finding all the involved concepts, thereby making systematic evaluation of software easier. Researches in the field of language engineering can leverage our models to identify use-cases of languages, enabling them to conceive of new DSLs that are more fit to deal with a common pattern. As a consequence, we assign the SHR a high relevance.

4.3.2 Developers (SH-D)

Our own implementation is based on the existing system developed in [32]. The code could not be reused, only to some degree as with grammar definitions. Developers are therefor respected as stakeholders in our engineering, for development of our software might be their obligation. Another involvement of developers could be directly using or even forking our software.

Concern Developers could include our software in their projects, extend the syntax, implement new UI features et cetera. This results in some requirements to our implementation in extensibility and documentation maturity.

Relevance We claim that we have a comprehensive implementation of a language – as opposed to an academic approach. The use cases for other developers are therefore not to be neglected, they are of a similar but not as high relevance to our system as 4.3.1 (SH-R).

4.3.3 Writers (SH-W)

These stakeholders are the primary developers of *MegaL* models. In contrast to researchers, writers are more likely to be regular software engineers or developers. They will have to some degree an understanding of linguistic concepts in software architecture — on the other hand they will bring extensive domain-specific knowledge into the context.

Concern Writers are particularly involved in language design, as we want to make syntax and editor as accessible to them as possible. In addition to the surface description of systems, they are also required to drive the implementation of plugins, which results in design choices for the plugin API.

Relevance As one 'half' of the main users, they do have the highest priority, ignoring their concerns would result in a comprehensive but unused language.

4.3.4 Users (SH-U)

The other core users: any reader of a megamodel, e.g., programmers, engineers or architects that need to know if a technology may be used or what the dependency on an artifact are, for understanding and operationalization.

Concern Readers do not require the extensive support for writing models, they are dependent of the models that writers produce, and thereby subordinate stakeholders to 4.3.3 (SH-W). Exploration features are a more *user*-centric issue.

Relevance Their importance is coupled with that of the writers. DSL usage – especially with a more documentation-oriented language – can be expressed by pairing document writers and the readers, one is existentially dependent of the other. Relevance for the users as stakeholders is therefore the same as in 4.3.3 (SH-W).

In summary, the most important stakeholders are the latter, 4.3.3 (SH-W) and 4.3.4 (SH-U), researches as defined in 4.3.1 (SH-R) may to some degree be distributed on those. The developers, 4.3.2 (SH-D), are respected in the implementation, but as there's a focus on the model structure and a more abstract implementation of the evaluation, they will not be explicitly discussed in this thesis.

4.4 Scenarios

Now that we have listed the core stakeholders, we would like to formulate a set of common tasks that are to be addressed by our implementation. In the following listing, a few scenarios are given, they did occur when using the existing *MegaL* implementation.

4.4.1 Exploration of a model (SC-Exp)

In the *MegaL* notation, a megamodel is given. This model is then used by, e.g, a reader 4.3.4 (SH-U) to inform themselves about a software system in (i) its hidden relationship like conformance or concept instantiation, and (ii) the actual locations of the artifacts a megamodel discusses. The authors of [20] present an exploration view of a megamodel, such functionality is desirable.

4.4.2 Migration of *MegaL* files (SC-Mig)

A lot of files have been written in the old *MegaL* notation. In addition, a lot of checkers were implemented. This scenario describes the process of copying the project with the associated megamodels, correcting notational discrepancies and to translate the old checker into the new plugin framework.

4.4.3 Headless run (SC-HLR)

A megamodel is developed with the help of an editor in an IDE. The created model is then executed in a standalone program.

4.4.4 Creation of a new megamodel (SC-Cre)

With an existing system at hand, a new megamodel shall be created. The artifacts and technologies of interest are to be identified and notated. Then, the plugin framework needs to be orchestrated so that the described properties are validated.

4.4.5 Instrumentation of a megamodel (SC-Inst)

A megamodel created for any system is to be integrated in a new megamodel for a different system, which shares properties with the one to be integrated. Instead of rebuilding the megamodel, the new model should reuse as much of the existing as possible. If the presented system shows multiple instances, the existing model should be allow orchestrating it more than once.

4.4.6 Integration of *MegaL* (SC-Intg)

For a system that displays node networks in a certain kind or uses a triple store to find logical models – such a system would use *MegaL* as a notation or as a database, including some limited usage of the type system and the evaluation.

4.4.7 Identification of a transitive artifact (SC-Tra)

When creating or migrating a *MegaL* model, an arising transitive artifact needs to be captured. That is, it must be exposed as an entity so it is referable to in any relationship.

4.4.8 Implementation of a new relationship type (SC-Rel)

A megamodel is created, including all the facts encoded in relationships. For some, evaluation is defined and imported from other modules. For those who are not implemented, new evaluation functions need to be created.

4.5 Requirements

After establishing users and use cases, the following section will list the requirements that are the foundation to our software. They will be given in the form of a labeled list, so that we can address them in the latter sections, especially in the structured analysis.

4.5.1 Integration (RE-Int)

Editor and execution engine must be integrated in an IDE.

Explanation The language is supported by an editor that is not a standalone product, but in a system of many editors and tools: an *integrated development environment*.

Discussion Integration into an IDE is common and of particular interest for our domain. The scenarios 4.4.1 (SC-Exp) needs navigation features to artifacts, and those are best displayed by designated editors, which is more likely to be integrated in an IDE. Editing models – as required per 4.4.2 (SC-Mig) and 4.4.4 (SC-Cre) – needs navigation as well: features in software systems are best identified when the system is actually readable. Regarding 4.4.8 (SC-Rel), support for programming a plugin is manageable with an IDE.

4.5.2 Stand-alone usability (RE-SAU)

Model retrieval and execution must not be IDE or tool dependent.

Explanation Parsing a model, i.e., translating a file written in *MegaL*, as well as executing it to prove software system properties, can be done without starting an IDE.

Discussion Tool integration is, as given in 4.5.1 (RE-Int), beneficial for language usage. However, such integration should not be limited to the picked IDE or the created tool. This can accommodate for scenarios like 4.4.3 (SC-HLR), which can be of relevance for stakeholders such as 4.3.1 (SH-R), as they would – in automated assessment – not want to execute a model by hand.

4.5.3 Environment independent models (RE-Ind)

A megamodel must not require or be bound to a certain environment.

Explanation Operating system and configuration that exceeds setting up the editor or a tool implementing standalone behavior must not impede the execution.

Discussion *MegaL* will not be used for private documentation, but for sharing knowledge. With many participants, it becomes improbable that the platforms are homogeneous. Overcoming platform heterogeneity is therefore required to support collaboration.

4.5.4 Extensibility (RE-Ext)

The system must provide pluggable mechanisms.

Explanation There must be methods that allow other developers 4.3.2 (SH-D) to implement functionality that has not been conceived of yet.

Discussion With regards to the vast majority of programming languages and the ways to access their artifacts, new code must be easy to include so that such access functions can be plugged — scenario 4.4.8 (SC-Rel) is highly dependent of this feature.

4.5.5 Domain focus of the API (RE-DFA)

The API that is required to implement 4.5.4 (RE-Ext) should be domain focused.

Explanation The plugin infrastructure requires conformance to a certain format. The implementation of plugins needs to incorporate domain specific behaviors.

Discussion When tackling tasks in checking linguistic relationships, recurring tasks may occur. Finding evidence of certain relationship can involve scanning the artifact's text, feeding it to a lexer might require reading it as a stream, conformance to a naming scheme could require file-existence checks and retrieval of web-site documents. Aside from these artifact-centric tasks, there might be other common patterns that should be lifted. Aside from usability, exposing technical details in the API can confuse or encourage abuse thereof.

4.5.6 Descriptive capabilities for runtime state (RE-Tra)

Program internal *in-memory* state must be addressable in a megamodel.

Explanation Most, if not all software maintains a state at runtime that is not observable in files or other manifestations. Such states could be local variables in a class at a given time or return values of an executed function. Their kind must be addressable in a model — we refer to them as *Transients*.

Discussion We want comprehensive description features for software systems. This requires transient artifacts to be included in a description. Especially when describing processes, which are a key ingredient of some software, the intermediate state is crucial for a complete discussion of a system.

4.5.7 Module support (RE-Mod)

Models must be reusable in a modular way.

Explanation A megamodel must be able to export its features, if it is concerned with a recurring structure, and import another one if its exported form is identified in the currently described system.

Discussion To eliminate over-execution of 4.4.8 (SC-Rel), existing models should support their reuse. This helps with keeping the implementation of model semantics small and as a benefit, identifies properties shared among many projects. Sharing between projects can be, for example, leveraged by the 4.3.4 (SH-U) kind of executives to identify cross-project relationships in projects. This can also be used by 4.3.1 (SH-R) for API analysis.

4.5.8 Reasonable syntax (RE-Syn)

The syntax should be minimal, maintaining good readability.

Explanation The syntax of *MegaL* should not include excessive language features, this however must not impede the readability.

Discussion Editors and readers need to understand an artifact at first glance. If there are language concepts, that are not clear – for example if they rely on an non-transparent naming scheme or a hidden implicit structure or non-intuitive syntax – then language comprehension and thereby its usage is impeded.

4.5.9 Reasonable appearance (RE-App)

The language editor should format the code in a reasonable way.

Explanation Code highlighting in color and font-shape needs to convey important, non arbitrary information.

Discussion Reading, especially when writing a model, is aided by a good formatting. Programming language editors usually provide syntax highlighting, they highlight keywords and give other language elements a specific appearance based on what they mean. This benefits should be leveraged in *MegaL* as well.

4.5.10 Editor features (RE-EF)

Editing megamodels must be assisted.

Explanation Creating and mutating a megamodel must me made easy by the means provided in an IDE.

Discussion As a special interest of 4.3.3 (SH-W), creating a megamodel and editing it requires tool-sets for common tasks. In addition, *inline navigation* (in the sense of content proposal) can assist in finding an artifact or a concept that is already defined and can be referred to instead of creating a new one.

4.6 Summary

As a conclusion, we now have a set of requirements arising from scenarios that are relevant to a set of stakeholders. The following listing gives an overview of the requirements.

Integration

- 4.5.1 (RE-Int)

  Editor and execution engine must be integrated in an IDE.

- 4.5.2 (RE-SAU)

  Model retrieval and execution must not be IDE or tool dependent.

- 4.5.3 (RE-Ind)

  A megamodel must not require or be bound to a certain environment.

Language and execution

- 4.5.4 (RE-Ext)

  The system must provide pluggable mechanisms.

- 4.5.5 (RE-DFA)

  The API that is required to implement 4.5.4 (RE-Ext) should be domain focused.

- 4.5.6 (RE-Tra)

  Program internal *in-memory* state must be addressable in a megamodel.

- 4.5.7 (RE-Mod)

  Models must be reusable in a modular way.

Cosmetic

- 4.5.8 (RE-Syn)

  The syntax should be minimal, maintaining good readability.

- 4.5.9 (RE-App)

  The language editor should format the code in a reasonable way.

- 4.5.10 (RE-EF)

  Editing megamodels must be assisted.

5 Design

This section is designated to convey the design choices and the top-level structure of the software. It will give an abstract perspective of the software and its development. The participating models and their relationships will be listed, the semantics will be defined and the project layout will be demonstrated.

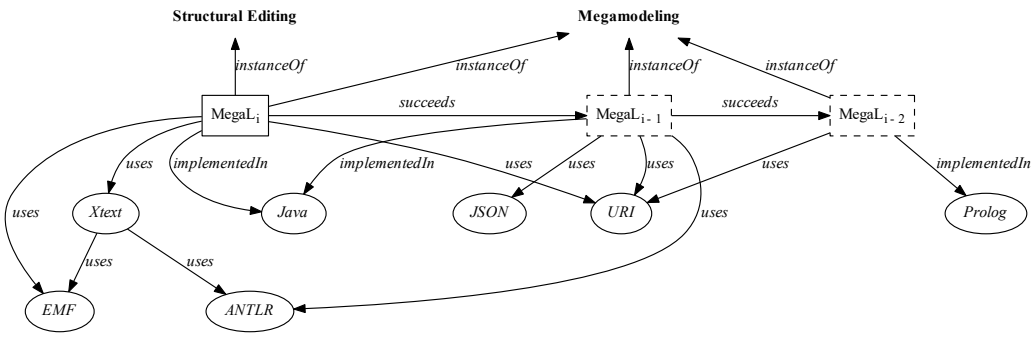

Fig. 2: The meta-structure of the project. Dashed are the previous versions, *MegaL*$_i$ points to the version presented in this thesis. Bold texts are concepts, and technologies are circled.

5.1 Development structure

The software system has been implemented in a set of iterations. This has been required, as there were changes to the syntax, features and other requirements. The *Xtext* grammar is loosely based on the textual syntax presented in *MegaL*$_{i-1}$, that is, the grammar was analyzed and – due to similarity of concrete syntax definition in *ANTLR* and *Xtext* – to some degree translated. Another driving factor for grammar development was the given example files shipped with *MegaL*$_{i-1}$, they exercised notation styles that were desired but not implemented. The structure of development is laid out in Fig. 2.

5.2 Model structure

As discussed, there are a number of models required for the implementation. They are edited by a set of tools, a variety of technologies is used. Figure 3 gives an overview of the participants, a brief discussion for all and an elaborated description for some elements will be given in this segment.

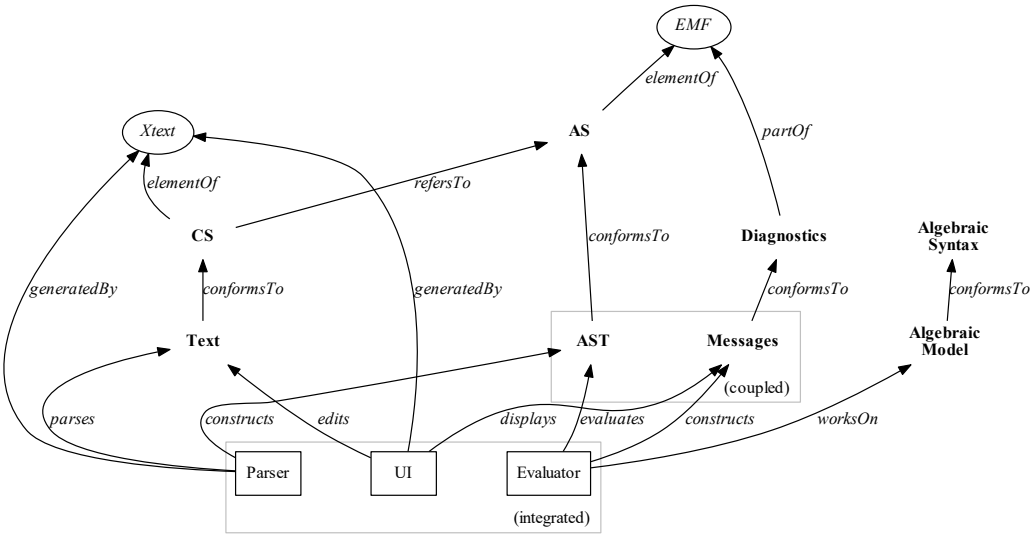

Fig. 3: Given here are the participating models (bold), artifacts and products (regular), technologies and languages (italic) and coupling thereof (clustered).

Technologies and languages The pair of *Xtext* and *EMF* refers to the technologies on one hand and their core languages on the other. As for *Xtext*, the concrete syntax definition format of 'grammar definition' is meant, and with *EMF* we also refer to the metamodeling language. An extensive discussion of both languages can be found in §2 or the corresponding literature [44, 5].

- **CS**: The concrete syntax is defined in the format of an *Xtext*-grammar. It is an element of the language, in that the grammar definition is parsed by the *Xtext*-parser.
- **AS**: The abstract syntax is given as a metamodel that is notated in the *EMF* syntax. Similar to **CS**, it is an element of the core *EMF* language, but in contrast, it is actually an *XML* document conforming to a schema.
- **Diagnostics**: An ingredient of *EMF* that we use as a mechanism of providing feedback is the diagnostics framework. It is a simple annotation model that allows assigning messages and their type – 'info', 'warning' or 'error' – to any element in an *EMF* instance model.

Instances Model instances in the *EMF* system are pairs of the abstract syntax tree (**AST**) and a set of annotations, called **Messages**. The prior

conforms to the abstract syntax definition (**AS**), the latter to **Diagnostics**. The actual megamodels that are visible to a user through the editor are notated in *MegaL* and presented as a code file, this feature is depicted as Parser constructing **AST** by parsing **Text**.

Algebraic model The model used to simplify calculations – especially in the deduction phase – is called the **Algebraic Model**. It is a set based representation of the model facts. We introduce another model, as *EMF* models bear some properties that are unfavorable when implementing a self mutating model. There's for example a number of management tasks that have to be performed and direct modification has to be locked when targeting user code. Another benefit of the introduced representation are the arising properties coming with algebraic data-types. Set union and intersection is highly intuitive in that notation.

Products The main software artifacts that constitute the *MegaL* language are integrated into the *Eclipse* IDE, implementing 4.5.1 (RE-Int). Some are to varying degrees derived from the grammar specification, others are purely hand-coded.

- Parser: The language's concrete syntax specification is transformed into an *ANTLR* grammar as a backend. The rules bear resemblance of the original grammar specification, but there are code fragments for *EMF* instance model construction interspersed. From the generated *ANTLR* grammar, an actual parser is derived, which in turn is used to parse **Text**. The entire component can be used in standalone operation, an important factor with regards to 4.5.2 (RE-SAU).
- UI: The *Xtext* system also derives an editor component, it provides syntax highlighting and code completion, which has to be adjusted in parts to conform 4.5.8 (RE-Syn), 4.5.9 (RE-App) and 4.5.10 (RE-EF).
- Evaluator: A hand-coded component — the devaluator uses the **AST** as an input and translates it into the **Algebraic Model**. It applies the deduction rules and evaluates the stated facts using plugins. To feed the emitted info, warnings and errors back, the **Diagnostics** framework is used: the evaluator writes to the **Messages**.

5.2.1 Abstract syntax

In terms of the abstract syntax, a metamodel containing all the classes, interfaces and properties is defined, so as to fit the presented definition of

megamodeling and to provide the appropriate division of model elements for concrete syntax mapping. The created model is depicted in Fig. 4.

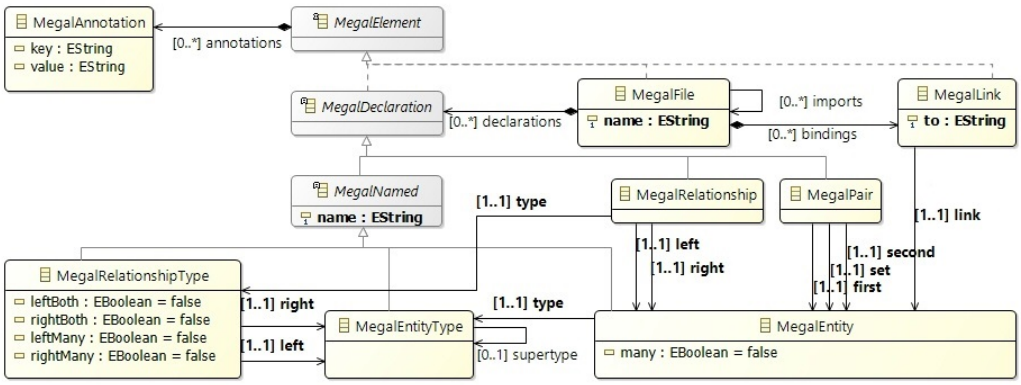

Fig. 4: This graph displays the metamodel for abstract syntax trees in *MegaL*, or, the abstract syntax. It is notated in *EMF*s graphical notation. Classes and interfaces are displayed as boxes with compartments, where the top box gives name and the bottom box lists features. Italic names identify interfaces. Arrows with white, triangular arrowheads display inheritance, the others show cross references. If the latter start with a check, they define *tree properties*: the arrow's origin marks containers, the destination marks children.

There are some judgments that lead to the introduction of abstractions, as well as some classes, interfaces and features that exceed the megamodeling basics:

- MegalAnnotation: To define an entry point for evaluation, there are some ways to connect megamodels and code. There is, for example, the method of connecting megamodel elements and classes directly, in that they are statically bound in the evaluation framework. As we are constantly evolving our foundation model – the *prelude* – another way is needed. The metamodel introduces MegalAnnotation, which maps a key to a value. In that manner, constants can be kept out of code and are attached to the *MegaL* elements directly.

- MegalElement: The MegalAnnotation can be applied on every level, therefore every element in *MegaL* composes a set or annotations. To accommodate for that fact, the MegalElement is introduced. It serves as a superclass to every element in the model, even the model itself. It has a reference of arbitrary cardinality to MegalAnnotation, which

is set to be a composition — annotation instances are thereby direct children of the element.

- MegalFile: The entry point of the abstract syntax tree, it is mapped to one *MegaL* file, as the name suggests. It has the property of a name, that way it can be referred to by other files via import references. That mechanism serves 4.5.7 (RE-Mod).

- MegalDeclaration: The building blocks of megamodels (entity types, relationship types, entities and relationships) are referred to as 'declarations'. This notational fact is encoded in an abstraction, the MegalDeclaration. *MegaL* files compose the declarations in one of their reference.

- MegalNamed: There is a difference between the *applied relationship* and the remaining declarations: while others have a name, e.g., the entity *XML*, defined as an instance of the type *Language*, or the *conformsTo* relationship type, *instances* of the latter do not expose a name. The applied relationship is only connecting three named elements. It is, however, a declaration. We therefore need another layer of abstraction: the interface MegalNamed distinguishes between named and unnamed elements.

- MegalPair: We map every element of a concrete *MegaL* file to an entry in the AST. This is true for legitimate *MegaL* building blocks, but also for *syntactic sugar*. MegalPair is an instance of such a sugared declaration. When we deal with applications of functions in linguistic architecture, like it is the case for an applied model transformation, we need to talk about the input of the function in conjunction with the resulting output.

  This is captured by the MegalPair: a function is seen as a 'relation', that is, a set of tuples — in the most basic case, a pair of input and output. We directly map that notion to a model element. It is observable that the MegalPair exposes a feature set, which is the target function seen as a relation. The other properties denote the input as first and the output as second. Elimination of syntactic sugar is delegated to the translation from AST to algebraic model.

In addition to these new elements that help with keeping the model structured, there are some features that are used to describe cardinalities in megamodels. Entities can bind for example many artifacts instead of one. Take for instance the classpath, it is composed of many class files. Instead of having a ClassFile and a semantically incorrect subtype ClassFiles thereof,

instances are allowed to have the type ClassFile+. This is even accounted for in the relationship type declaration.The left and right entities can take one of the following cardinalities: (i) one, (ii) many or (iii) arbitrary, where the last describes cardinality independence.

5.2.2 Concrete syntax

The concrete syntax is – as mentioned – notated in the *Xtext* grammar description language. A rundown of the concrete syntax definition with the appropriate mapping is given here.

```
grammar org.softlang.megal.language.Megal hidden(WS, COMMENT)
import "http://softlang.wikidot.com/megal"
import "http://www.eclipse.org/emf/2002/Ecore"
```

The defined grammar is given a name in the first line, whitespaces and comments are hidden, as they are part of the lexical analysis but not used in the parsing. In contrast to the example given earlier, there are no imported language features, the *terminal rules* are defined manually at the end of the grammar. In addition to the imported AST metamodel, the data types for *EMF* are imported. The prior is labeled `http://softlang.wikidot.com/megal`, the latter reside in the package assigned to `http://www.eclipse.org/emf/2002/Ecore`; both are symbolic URIs.

The initial rule initializes the MegalFile, its name is assigned and followed by an interspersed collection of declarations, imports and bindings in any cardinality.

```
MegalFile: annotations+=MegalAnnotation*
  'model' name=ID ('import' imports+=[MegalFile]
    | declarations+=MegalEntityType
    | declarations+=MegalRelationshipType
    | declarations+=MegalEntity
    | declarations+=MegalRelationship
    | declarations+=MegalPair
    | bindings+=MegalLink)*;
```

In the following segment, the declarations are defined. They initialize properties with assignments or cross references and define some optionally parsed alternatives.

```
MegalEntityType: annotations+=MegalAnnotation*
  name=ID ('<' supertype=[MegalEntityType] | 'as' 'entity');

MegalRelationshipType: annotations+=MegalAnnotation*
  name=ID
```

```
  '<' left=[MegalEntityType] (leftBoth?='(+)' | leftMany?='+')?
  '*' right=[MegalEntityType] (rightBoth?='(+)' | rightMany?='+')?;

MegalEntity annotations+=MegalAnnotation*
  name=ID ':' (type=[MegalEntityType] many?='+'?
  | domain=[MegalEntity] '->' range=[MegalEntity] ']')?;

MegalRelationship: annotations+=MegalAnnotation*
  left=[MegalEntity] type=[MegalRelationshipType] right=[MegalEntity];
```

After the declarations, links (bindings) are defined as mappings from existing entities to strings. Instead of taking a text for the binding, it would be possible to include the URI syntax. In the presented implementation, this restriction is lifted for simplicity and to allow special use cases exceeding the URI notation — evaluating plugins can view the binding *string* and operate on it in any conceivable way. Following the bindings are annotations and pairs, their definition is straightforward.

```
MegalLink: annotations+=MegalAnnotation*
  link=[MegalEntity] '=' to=STR;

MegalAnnotation: '@' key=ID (value=STR)?;

MegalPair: annotations+=MegalAnnotation*
  set=[MegalEntity] '(' first=[MegalEntity] ')' '|->' second=[MegalEntity];
```

The final segment lists the terminal rules. There are identifiers, strings (text sequences), comments and whitespaces. Identifiers are consecutive characters and numbers, strings are enclosed in single quotes, comments are in the *C* style (double slash for line comments and slash-asterisk/asterisk-slash delimited multi-line comments) and whitespaces erase everything in-between the important constructs.

```
terminal ID:
  '^'? ('a'..'z' | 'A'..'Z' | '_' | '0'..'9')+ ('.' ('a'..'z' | 'A'..'Z' | '_' | '0'..'9')+)
      *;

terminal STRING returns EString:
  "'" ('\\' . | !('\\' | "'"))* "'";

terminal COMMENT:
  '/*'->'*/' | '//' !('\n' | '\r')* ('\r'? '\n')?;

terminal WS:
```

```
(' ' | '\t' | '\r' | '\n')+;
```

5.2.3 Algebraic syntax

As illustrated in the introduction to this section, the algebraic model is introduced as a way to simplify calculations and operations on megamodels. There's two domains of values that originate from the actual *Java* implementation, one is Σ based on *Java*s String for texts. It serves as an identification for all elements in the model. The other is Ω corresponding to Object. Its only requirements are that it is capable of representing URIs (or strings in our notation) as well as the desired transient object values, addressing 4.5.6 (RE-Tra). Both Σ and Ω are listed explicitly in the following equations.

$$\Sigma = \mathsf{String} \tag{1}$$
$$\Sigma_\top = \Sigma \cup \{\top\} \tag{2}$$
$$\Omega = ..., \mathsf{URI} \subseteq \Omega, \text{“}\textit{Transitive values}\text{”} \subseteq \Omega \tag{3}$$

Given the requirements for type and instance level, the model is hereafter built from a set of types for entities and relationships, as well as their instances. The last field in the tuple is the bindings from entities to their manifestation.

$$E_{type} : \Sigma \to \Sigma_\top \tag{4}$$
$$E : \Sigma \to \Sigma_\top \tag{5}$$
$$R_{type} : \Sigma \times \Sigma \times \Sigma \tag{6}$$
$$R : \Sigma \times \Sigma \times \Sigma \tag{7}$$
$$B : \Sigma \to \Omega \tag{8}$$
$$M = (E_{type}, E, R_{type}, R, B) \tag{9}$$

- E_{type}: The entity type is defined as a mapping from the name to its supertypes name — the root of the type lattice is $\top$.
- E: The entities are defined as instances by giving them a name out of Σ and a type. The type is also out of $\Sigma_\top$, they are not required to ignore the type lattice's root.
- R_{type}: The relationship type is declared as a triple relation. As there can be multiple relationship types between a pair of entities, a mapping is not feasible for this signature. The first element names the relationship

type, the second and third entries of the tuple denote left and right entity type respectively.

- R: Applied relationships are given as a relation too, as they are also not required to be singleton for a pair of entities. They are notated as left entity, relationship type name, right entity.
- B: Bindings are simply given as a mapping from entity to the bound object or URI.

Overwriting composition On the algebraic model an operation is defined, it serves the implementation of modularity and orchestration of modules required for 4.5.7 (RE-Mod): the *overwriting* composition $\circ$ of models, dependent on a 'function addition with precedence', labeled $\propto$.

$$m_1 = (e_{t,1}, e_1, r_{t,1}, r_1, b_1) \subseteq M, m_2 = (e_{t,2}, e_2, r_{t,2}, r_2, b_2) \subseteq M$$
$$m_1 \circ m_2 := (e_{t,1} \propto e_{t,2}, e_1 \propto e_2, r_{t,1} \cup r_{t,2}, r_1 \cup r_2, b_1 \propto b_2) \quad (10)$$

The $\propto$ operator takes two functions that share a set of elements in their domains and ranges, it's defined for arbitrary supersets X and Y.

$$F : X_F \to Y_F, G : X_G \to Y_G$$
$$X_F \subseteq X, X_G \subseteq X, Y_F \subseteq Y, Y_G \subseteq Y$$
$$F \propto G := (dom(G) \lhd F) \cup G : \quad (11)$$

The definition of $m_1 \circ m_2$ applies a union on the relation components, the mappings are in fact treated differently. The operator $F \propto G$ is defined as a the union on its operands, Fs domain however is restricted to the arguments not mapped by G, the result is thereby also a uniquely mapping, where the right operand has precedence over the left.

This allows the bindings to be overwritten, but also the entities and their types. In practice, only bindings are overwritten, wellformedness constraints do not hold for the other in general. In the current implementation, overwriting entities or their types is marked as an error, but replacing an entity's type with a more special one can be seen as a valid use-case for overwriting.

Take for example an entity defined in a module that is designated for reuse and orchestration. The defining megamodel might not know about the type lattice defined in the reusing module. In the latter, some type level restrictions might not hold if the exported instance is not special enough. When overriding the entity type coherently with the type lattice, such a substitution could eliminate the type level discrepancies — *covariance* in overwriting is subject to further discussion.

Mapping When editing a *MegaL* file, the AST is created and maintained as a representation. The evaluation is contrast defined on the algebraic syntax. To evaluate a megamodel, the AST needs to be translated into the algebraic syntax. As metamodel classes and categories do not differ very much, this is trivial: imports are chased and all the elements are collected. When traversing a megamodel import, the overwriting composition is applied so that the importing model is of higher precedence, allowing orchestration by rebinding. In addition to the trivial translations,

Evaluation results are relative to the algebraic syntax, messages have to be tracked back to AST nodes so that feedback can be provided. The evaluation is performed on a completely concatenated module, in order to obtain a responsible model element – which could be located in an imported module – the names in highest preceding megamodel are looked up and the imports are gradually descended. That way, an error or warning annotated to an entity in the algebraic model can be tracked back to the corresponding AST node and – by the lexical information given in the *MegaL* file – to the location in the originating text file.

5.2.4 Project structure

Multiple concerns are to be regarded, as the system is comprehensive and based on a multitude of models, the components are clearly separated. The software is therefore divided into three main projects with accompanying side projects.

There is at first the main *MegaL* project. It provides both the **AS** and the **Algebraic Syntax**. It also provides the validation, that is performed by an execution based on plugins. Editors based on the *EMF* model may use *only this plugin* to construct and execute a megamodel.

One of it is the *Xtext* editor of *MegaL*. It is split up into multiple projects, each serving a different purpose. The most important ones are concerned with language and UI. The syntax definition file – presented earlier – is served in the language project, it also supplies the name resolution and applies the validation by executing the translated **Algebraic Model**. The UI project provides the actual editor by filling *Xtext* framework constructs with functionality based on the grammar. It gives for example refactoring options, files common error resolutions and shows a documentation. It's also responsible for syntax highlighting and navigation. Those are assisted by a 'SDK' project that allows compiling a packaged version for users to install — *Eclipse* packages multiple plugins as so-called 'features' [17] . The last *Xtext* originating project is the support for unit-testing the language.

There's another project that assists in the connection of Java source code

that is written parallel to the megamodel, allowing integration of plugins developed just in the same environment. Another project that won't be covered in this thesis is the *Sirius* editor.

5.3 Semantics

In this approach, the semantics are based on the definitions in [32]. The notions of *well-formedness*, *correctness* and *completeness* are a strong influence.

Wellformedness At first, there are some trivial utility definitions, all for an $x \in \Sigma$ and relative to a megamodel instance $(e_t, e, r_t, r, b) \subseteq M$. Capital letters are concerned with the type level, while lower case letters account for the instance level — in the sense of megamodeling this will be entity and relationship types versus entities and relationships.

$$R_x \Leftarrow \exists y : (x, y) \in e_t \tag{12}$$
$$T_x \Leftarrow x = \top \tag{13}$$
$$V_x \Leftarrow R_x \vee T_x \tag{14}$$
$$v_x \Leftarrow \exists y : (x, y) \in e \tag{15}$$

There are four definitions now: Equation (12) defines resolvable entity types, (13) is a shorthand for equivalence to $\top$, i.e., being the 'Entity', the validity in (14) is a composition of those. As there is no corresponding root in the instance level, the validity is directly mapped to the resolvability of an entity name. In the following set of equations, the wellformedness is defined for the individual constituents of the megamodel, starting off with the simple cases.

$$okE_t((t, s) \in e_t) \Leftarrow V_s \tag{16}$$
$$okE((e, t) \in e) \Leftarrow V_t \tag{17}$$
$$okR_t((n, d, r) \in r_t) \Leftarrow V_d \wedge V_r \tag{18}$$
$$okB((e, b)) \Leftarrow v_e \tag{19}$$

In addition to the helpers, there are two predicates that decide if a type is subtype of another. The first holds if there is a direct instance of a subtype relationship if it is transitively reachable. The second includes equality of the parameters.

$$t \twoheadrightarrow^{+} s \Leftarrow (t,s) \in e_t \wedge \exists (t,t_i) \in e_t : t_i \twoheadrightarrow^{+} s \tag{20}$$

$$t \twoheadrightarrow^{*} s \Leftarrow t = s \vee t \twoheadrightarrow^{+} s \tag{21}$$

Now that wellformedness for entity types (16), entities (17), relationship types (18) and the bindings (19) is defined, let's look at relationships.

$$okR((s,p,o) \in r) \Leftarrow \exists (s,t_s),(o,t_o) \in e,(p,d,r) \in r_t : t_s \twoheadrightarrow^{*} d \wedge t_o \twoheadrightarrow^{*} r \tag{22}$$

As observable, relationships require resolving the types of subject s and object o, as well as the relationship types domain d and range r. Hereafter, the resolved instance types t_s and t_o are checked. They must be assignable to their relationship type's domain d and range r, hence the $\twoheadrightarrow^{*}$ application.

At last, there will be the composition of all wellformedness definitions. A megamodel is wellformed, if all its entries are.

$$ok(m) \Leftarrow \begin{cases} \forall i_{e_t} \in e_t : & okE_t(i_{e_t}) \\ \forall i_e \in e : & okE(i_e) \\ \forall i_{r_t} \in r_t : & okR_t(i_{r_t}) \\ \forall i_r \in r : & okR(i_r) \\ \forall i_b \in b : & okB(i_b) \end{cases} \tag{23}$$

Interpretation The interpretation is defined as checking the propositions a megamodel makes, that is, entity and relationship instances. A megamodel interpretation function has been presented in [32], it's defined as binding or resolving the URIs for entities, then checking relationships on the resulting URI pairs. They provide *correctness* and *completeness*, as given by the following quotes.

> Correctness means that an interpretation does not provide any definitions that are not possibly needed by the associated megamodel. Provision of superficial definitions may be acceptable, though, in practice.

(Laemmel et al., [32], pg. 12)

The insight that *superficial provision* is acceptable also applies to the approach of this thesis, especially when having our modularity in mind. Some modules may be developed for the sole purpose of defining the evaluation, others then reuse them to check their statements. Modules are not included in the algebraic model, the definitions are concatenated and overwriting is applied.

> Completeness means that an interpretation suffices to resolve all entities or parameters and to evaluate all relationships for a given megamodel. As discussed, in practice, we do not necessarily require completeness, as we may be unable to resolve certain entities or to evaluate certain relationships, at a given point. However, ambiguities regarding resolution or interpretation should be reported.

(Laemmel et al., [32], pg. 12)

The presented implementation is similar in that regard. There are some facts that are left untouched in by the evaluation. They are annotated with a warning, so that it is explicitly clear that a model fact may not hold, as no evidence of the opposite is found. Ambiguity may also arise, the plugin selection mechanism tries to eliminate such ambiguities by structured rules.

Validity of stated facts in a megamodel is asserted by user defined mechanisms in *plugins*. An element can be *valid* or *invalid* and have be annotated with messages. They are usually combined in the following setup: (Invalid, (Error, Text)), (Valid, (Warning, Text)), (Valid, (Info, Text)) and (Valid, —). Warnings are in this case annotating elements in the model that do not expose the properties that are needed to execute the responsible plugins. Such cases are missing bindings for artifact based entities and relationships they are involved with. Side effects, especially on the file system are not wanted, but avoiding them is not enforced.

Plugin selection Selection of evaluation is done based on the applied relationship type first — a 'conforms to' plugin is not executed on an 'element of' relationship. Therefore, relationship and entity types are accompanied by an *aggregating plugin*, as demonstrated in the following *MegaL* segment.

```
// Declaration of the relationship type
elementOf < Artifact * Language

// The associated 'aggregator'
ElementOfEvaluator : Plugin
```

This is in the most cases not sufficient, more information has to be pulled. This version is at its current state limited to manually querying model structure to select an appropriate plugin. The 'Artifact element of Language' plugin for example is enriched by an *acceptor* based evaluator, which is in turn dependent language acceptors, as demonstrated in the following snippets.

```
// This is not associated to a relationship type, but
// plugged into the aggregator
ElementOfByAcceptorEvaluator : Plugin
ElementOfByAcceptorEvaluator partOf ElementOfEvaluator
```

This is the definition of the *acceptor* based evaluator, it's associated to the top-level aggregator. In the following fragment, it will be extended by a plugin for a language instance.

```
AcceptorXML: Plugin
AcceptorXML partOf ElementOfByAcceptorEvaluator
AcceptorXML accepts XML

AcceptorXML = '<link to an acceptor function>'
```

Let's say we have the language *XML*, as well as an acceptor function for *XML* text. We define an according *XML* acceptor plugin, extend the previous plugin with it and associate it with the language instance. At last, the actual function is linked. In the current implementation, the 'ElementOfByAcceptorEvaluator' would manually go through its plugins and see if they are fit to accept an artifact.

The desired but not implemented mechanism is similar. In contrast to the presented method, there is no plugin code for aggregators involved. Instead, they are enriched with a set of declarative rules that enables them to select their appropriate part, thereby delegating evaluation. This method is called *dispatching*, it is based on queries uniquely identifying facts that are then used to pick an evaluator. Consider for example Tab. 1.

Such a system of queries – as presented in the table **Root** – can be annotated to a first order child of the aggregator plugins. The parts thereof can then be annotated with the parameter names and the instances of them, as given in **D1** and **D2**. It is however not always clear if an appropriate plugin is selected unambiguously, a resolution strategy or error marker can overcome this.

Reasoning In addition to checking, which is to some degree implemented the previous version, we offer reasoning. Any element in the model can be used to generate more model elements. These can then in turn be reentered in this process and generate more models. Reasoning plugins are executed recursively until the identity of input and output. Therefore, they should not unconditionally increment their input. To allow tracking back messages annotated to a 'derived' element, *reasoner* output will be associated with the origin, as depicted in Fig. 5.

Root

Relationship	type(**l**)	type(**r**)	Dispatch on	to
l *elementOf* **r**	Artifact	Language	**r**	**D1**
l *conformsTo* **r**	Artifact	Artifact	**x**,**y** where **l** *elementOf* **x** **r** *elementOf* **y**	**D2**

D1

r	Plugin
Java	JavaAcceptor
XML	XMLAcceptor
XSD	XSDAcceptor

D2

x	**y**	Plugin
XML	XSD	XSDConformanceChecker
Ecore	Ecore	EMFConformanceChecker
XMI	Ecore	EMFConformanceChecker

Tab. 1: Dispatching based on the model. The topmost table is the one that is responsible for the first degree of evaluation. It selects the plugin, that relationship is dispatched to. The bottom tables are on the second level and responsible for language-element checking and instance-to-schema conformance.

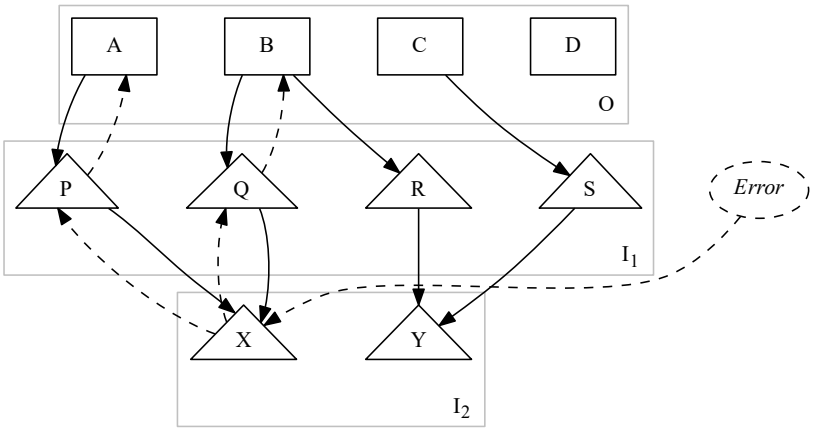

Fig. 5: *Origin tracking* of the *Error*. Boxes are original elements, triangles depict reasoning output and are connected with solid lines to their origin, the dashed line indicates the tracking flow. Q and P are predecessors of X, as their resulting output element X is syntactically equivalent. In the end, A and B are annotated with the error. Ligher boxes accumulating elements are the phases: Original model O, followed by iteration I_1 and I_2, assuming that I_3 would have returned an identical model.

Transients Transients have already been described as a way to model non-manifested artifacts like memory state or execution results. The algebraic signature accounts for such states in its Ω set. Acquisition of the such a state is done by applying an 'accessor' function, i.e., a *reasoner* that results in a new *binding*, in which the memory state is then stored.

Transients were introduced by an example of a *transformation* with an intermediate state. Let's elaborate on this example, an intuitive shorthand notation for function definition and multiple entity instances will be used.

```
Notation, Graph : Language

read : Notation → Graph
transform : Graph → Graph
write : Graph → Notation

// Bindings to the functions
...
```

There are two languages now, one for the file format and another for the in-memory format. Aside that are three functions, one is reading the input, another transforms it, then it's saved by the last function. The entity

bindings are omitted in the example. Now, the application of those will be illustrated.

```
inputFile, outputFile : Artifact
input, output: Transient

read(inputFile) ↦ input
transform(input) ↦ output
write(output) ↦ outputFile
```

The scenario at hand is $outputFile = write(transform(read(inputFile)))$, intermediate stages are however stored, so they can be reasoned about. In that configuration, reasoning will apply the desired behavior in the following operation.

1. In the first iteration, read will be executed, the file is translated by the deserialization function, a 'graph' will be bound to *input*.
2. Application of *transform* requires the input and is therefore delegated to the second iteration, resulting in the transformed graph.
3. As we try to avoid side effects, we assume that the output file is bound to a manifested artifact. The last function will thereby check if its result is – with a certain degree of freedom – equal to the expected output.

After executing the reasoning, both transients will be bound. One could then check that a specific property holds for the output graph, for example that it is a directed, acyclic graph, or that it is a permutation of the input graph. A more elaborated and integrated example will be discussed in the analysis.

5.4 Evaluation

The execution of semantics, divided onto reasoning and checking, is defined in a batch process. The reasoning function is conditionally tail-recursive, feeding back if a single plugin has modified the model and returning if identity has been reached. The checking is in contrast a one-time process that executes all checkers that are defined in a model. Both will be demonstrated in the following segment.

Reasoning Inferring new elements is defined as a function taking an *algebraic model*, containing all the megamodel elements, and an *origin tracking*

set coordinating the behavior depicted in Fig. 5. Input and output overlap, the signature already hints at the recursive nature of the function.

```
reason : Model × Origins → Model × Origins
reason(m, o) {
  // The resulting model and the origin tracking table
  m_next ← m
  o_next ← o

  // Iterate all elements paired with the reasoner functions
  for(x in m)
    for(f in reasonersFor(x, m)) {
```

In the first lines of the method body, the inputs are copied into a new name. This is required, as recursion needs to be skipped if input and output are equal. In succession of these lines, all model elements are handled by the appropriate reasoners in a loop, assuming that the algebraic model enumerates all the definitions accordingly. In the second loop, a function is referred to: reasonersFor(x,m) selects the appropriate reasoners for a given model element in relation to the entire model. As model semantics are configured in the model itself, scanning it is valid. In the implementation at hand, such a lookup would select a multitude of plugins, where only a few are applicable and reject the input in their implementations. The 'dispatching' illustrated in §5.3 can be factored into the reasoner selection function.

The reasoner is notated as a function from model element and entire model to a new model. It is applied on the current element and the original model. All elements it creates are in consequence annotated with their origin and the entire model is appended to the status.

```
      // Apply plugin for the new model elements
      m_new ← f(x, m)

      // Track origin
      o_next ← o_next ∪ {(x↦x_new) |x_new ∈m_new}

      // Expand intermediate model
      m_next ← m_next ∘ m_new
    }
```

When the iteration is completed, a model may not have changed — for example if no reasoner was deemed responsible, or all that were did not generate models that extended the input. Such a case can be tracked when comparing input and output. If there are no changes the status is returned, if there are, the intermediate result is fed back.

```
19   // Apply until recursive identity
20   if(m_next = m ∧ o_next = o)
21     return m, o
22   else
23     return reason(m_next, o_next)
24 }
```

Wellformedness is a delicate matter in the context of reasoning. The used operation of overwriting composition ($\circ$) can change the type-lattice of an entire model, potentially invalidating existing facts. This issue is not very prominent in the currently exercised scenarios, as the type level is not very likely to be inferred. If however there are use-cases that require deriving new type level elements, arising wellformedness violations can be either marked as errors or eliminated by substituting overlapping elements in order to avoid name clashes.

The method bears some potential for parallelization. In fact, both loops in line 8 and 9 are independent of the mutated state, only on the input. Identity must however be checked *after* all parallel processes are completed.

Checking After the model is fully exploded in all inferable facts, it is fed into the checking phase. In semantics design, wellformedness and interpretation were discussed. In the shown implementation, constraints defining wellformedness are interpreted on the fly, e.g., relationship applicability can at a certain point during reasoning not be present, but then be restored by a subsequent reasoner. When entering checking, wellformedness should be checked, the implementation rejects this behavior, as model parts should be checked regardless of other erroneous parts. Thereby, writers 4.3.3 (SH-W) can focus on a fragment assisted by direct feedback, without having to restore wellformedness first.

Checking is defined as a single batch of applications and listed in the following fragments, starting off with the signature taking model and origin tracking. Its output is a set of messages covering information, warning and errors — annotations are collected in a set.

```
1 check : Model × Origins → Annotations
2 check(m, o) {
3   // The result is the set of messages for an element
4   a ← ∅
5
6   // Iterate all elements paired with the checker functions
7   for(x in m) {
8     // Find the responsible elements via origin tracking
```

```
    xs′ ← rootOf(x, o)

    // Execute all checkers and annotate the origin
    for(f in checkersFor(x, m))
      a ← a ∪ {(x′ ↦t) |x′ ∈xs′,t∈f(x, m)}
  }

  // This is a one pass function, return the generated set
  return a
}
```

After initializing the annotation set, all the model elements and their associated plugins are exercised. The latter are treated as functions on element and containing model. Their results are relative to the input elements, they can in fact be derived by a reasoner. The messages therefore need to be tracked back according to Fig. 5. This is done by consulting the root finding function for a given element in the origin tracking set, it backtracks the graph retrieving all elements that outputted a syntactically equivalent element during reasoning. This accounts for multiple ways of deriving a model fact, instead of randomly picking a cause for an arising error, *all* originating elements are annotated.

```
evaluate : Model → Annotations
evaluate(m) {
  // Compose functions, the input model has no origin tracking set
  return check(reason(m, ∅))
}
```

The last fragment depicts the trivial composition of reasoning and checking. After the mapping from AST, no origin is annotated, hence an ∅ is passed as the argument to reasoning.

5.5 Plugin framework

Plugins are an essential ingredient in modeling linguistic architecture, they encode executable domain specific knowledge. The latter fact is a major concern when designing a plugin framework. With that in mind, the API of this implementation was designed.

During development of this *MegaL* iteration, plugins were created without any explicit API, but rather based on the plugin signature and the *Java* core API. As the corpus grew, some significant patterns and functionalities were identified. They are listed in the following summary and will thereafter be addressed individually.

1. File access and existence tests are a common use case. Most relationship checks are concerned with artifacts. Files, folders, websites, web resources etc. play a central role in linguistic architecture. Some relationships require the existence of a certain resource, others may read the content as either bytes or as plain text. To accommodate those use cases, a resource API is needed.

2. Most checks depend on functionality that is applicable if the input is wellformed, in order to communicate mismatching arguments, they throw exceptions. In a regular program, such an exception is caught by a handler or propagated to the referring function's environment. When concerning validity to include wellformed input, such an exception can be used to create an error marker, the API should provide functionality to alleviate behavior of that sort.

3. Strong reliance on control flow is a key insight into the developed plugins: many nested conditional blocks were instrumented to implement checking. While this is feasible for a certain degree of nesting, there are problems arising from excessive usage. Especially when designing utility methods, control flow is hard to surpass the utilities — a return statement can only break the containing function. If a boolean is used as a return value to direct control flow, returning an additional value in the utilities is impossible, workarounds like using 'Optionals' obfuscate the code. Control flow structures need to be relaxed.

File access Artifacts are reflected as an API class: they point to the locations a megamodel specifies and provide methods for reading them as bytes or chars, testing their existence and listing their children, siblings and the parent. They are linked using *bindings*, which can assume any format, but the URI standard is recommended, as there are some helpful properties coming from it.

URIs are specified with regard to use cases in the World Wide Web, namely the functional recommendations for Internet Resource Locators and Universal Resource Names[8]. The fact that they can encode a great variety of specificationss which are still processable by a machine and human-readable to a certain degree makes them a candidate for the bindings. Bearing in mind that they are platform-invariant, they are preferable when implementing 4.5.3 (RE-Ind). In addition to the standard access protocols like *FTP* or *HTTP(S)*, the proposed API provides mechanisms that help identifying local resources. This is done with regard to the *Eclipse* workspace mechanisms: project relative resources may be specified. See for example the following.

```
someFile : Artifact
someFile = 'C:\\User\\Austin\\Projects\\Webapps\\workspace\\org.example
    .webapp\\src\\...\\Hugs.java'
```

The first observation is that file's path is *absolute*. In addition, it is dependent on *Windows* layout. As a result, the path can not be used by a *Linux* user and any other *Windows* user with a different folder layout. Let's consider that other users are working in the same project, such a gap can be overcome by using a relative, platform independent binding, as in the following fragment.

```
someFile : Artifact
someFile = 'workspace:/org.example.webapp/src/.../Hugs.java'
```

Now, workspace resource mechanisms are leveraged. Changing the project layout is an actual change that can have impacts on the execution, where changing the workspace root should usually have no effects. Thereby, locating a file based on the workspace allows (i) safely addressing a file and (ii) pointing out relationships between multiple projects in a workspace. The latter is useful if a software system is distributed among several projects. Aside workspace relative paths, the API supports classpath URIs: addressing a method residing in a class and linking plugins is done in that style.

In addition to the locating mechanisms, the proposed artifact framework provides methods of automatically managed resources. Files or other streams opened throughout the course of a plugin's execution are tracked. When the plugin terminates, all opened resources are closed. Thereby, excessive resource management is lifted.

Automated exception handling Take for example a checker asserting certain structures in a *Java* files based on the *javaparser* project, which serves an exception in case a syntax error occurs, or an *XML* transforming reasoner based on the *Java* built-in *XML* library, it would crash with an exception if the given input would be erroneous.

In both cases, one could argue that an exception at that particular reasoner should be dealt with by propagating the exception as an error message. This disregards the fact that executing said relationships – asserting structure and transforming a document – are not responsible to markup errors, this would be the duty of an 'elementOf' relationship. The latter are in contrast to the others not very numerous, the default behavior should tend to the prior instead.

The presented API accounts for that, developers may subtype classes providing a framework for exception handling. Both reasoners and checkers may in the course of their execution throw any exception. The default behavior is

to swallow the exception, it can be modified as the default handling routine is coordinated by an overwritable method. In the following example, the difference will be illustrated.

```
try {
  DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
  DocumentBuilder builder = factory.newDocumentBuilder();

  try {
    Document document = builder.parse(...);
    document.getDocumentElement().normalize();

    // Work with the document
    ...
  } catch (SAXException e) {
  } catch (IOException e) {
  }
} catch (ParserConfigurationException e) {
}
```

The purpose of this fragment is to obtain a model for an *XML* document, it requires two nested handlers with three handled exception types, unnecessarily obfuscating the purpose and not adding any benefit. Instrumenting the proposed exception handler, the code reads as follows.

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
Document document = builder.parse(...);
document.getDocumentElement().normalize();

// Work with the document
...
```

The code is much more concise than before, it does the same and handles the exceptions appropriately. Let's see how this fragment can be instrumented as an 'element of' test by overwriting the exception coordinator.

```
@Override
protected void onException(Throwable t) {
  getContext().error("Not an element, reason: " + t.getMessage());
}
```

The only exceptions we are expecting arise from mismatching format for the *XML* input, parser configuration errors are ignored anyways and would not occur, as there is no configuration involved. The *SAX* exception indicates

an error in the *XML* file. In contrast to the other two, the IO exception might be of significance, or put in the format of a question: are non-existing files *not element* of a language, or are they configuration errors? The answer to this question can be expressed in the following notation.

Control flow As with exceptions, other control flow mechanisms become prominent when developing checkers and evaluators that describe a certain aspect of a relationship — the status in which they are checkable or executable need to be asserted first. Usually this would be performed by a set of conditional executions in if clauses terminating the function if a certain criterion is not met. This is feasible to some degree but the usage patterns are highly similar among many plugins. Take for example the following code: it checks, if the binding to an entity exists and acquires it.

```
// Terminate if left operand has no binding
if(!rel.getLeft().getBinding().isPresent()) {
  getContext().error("Binding needs to be present for relationship checking.");
  return;
}

// Get the artifact for a present binding
Artifact left =  getContext().getArtifact(rel.getLeft().getBinding().get());
```

It takes some lines to express a requirement for an artifact, refactoring this code is not trivial, as it contains a return statement. This is where the *exception handler* – illustrated above – comes to play again: we declare an exception type invisible to all other classes, it can only be instantiated and handled by its containing class. It is able to carry a termination type, as well as a message through the entire stack trace. This allows refactoring methods that are able to terminate a the checking or reasoning process of a plugin. Utilizing it reduces the code to the following.

```
// Get the artifact for a present binding
Artifact artifact = artifactOf(relationship.getLeft());
```

The used method is part of the following family, all concerned with trivial tasks that may require nested termination.

- when(boolean) An execution can be continued when a condition is satisfied, otherwise it will be terminated silently.
- bindingOf(Entity) Combines the behavior of when(boolean), applied on the presence of a binding, with returning the actual value of the binding. It would return the raw URI bound to the entity and terminate if nothing has been assigned to it.

- artifactOf(Entity) The method does the same as bindingOf(Entity), but instead of returning the binding, the result is wrapped by the artifact abstraction discussed in the *file access* segment.

These building blocks demonstrate the essence of using nested execution terminations, being concerned with termination and avoiding optional return values. To give a deeper insight into this pattern, a new method will be developed in the next fragment.

```
// Defined as a method taking a desired class and requiring
// the actual object to be an instance of it
<T> void T as(Class<T> desired, Object actual){
  // Continue only if the following cast is valid
  when(desired.isInstance(actual));

  // We can only reach this if the cast is OK
  return desired.cast(actual);
}
```

The defined method is very general, it refers to the execution condition defined by when(boolean) and can assure that an object is at least of the *desired type* by checking and casting to it. This can be useful, as bindings are specified to be of any type. If one requires them to be URIs – so that specific judgments about their shape can be executed or they can be read by a stream – casting to the type is required.

5.6 Editor

The editor, based on *Xtext*, provides the interface for writer and reader alike. Writing is widely influenced by the *content proposal* tools and the *refactoring* methods, as well as *formatting* options. The quality of a tool is also dependent of more reader-centered features, namely *exploring* and *code layout*.

Formatting In [25], the authors describe for example the benefits of formatting, it helps to reduce misconceptions about the code meaning. They illustrate that at the hand of a simple arithmetic expression: in `a+b * x`, the operator precedence might be misunderstood, observable when looking at the opposite `a + b*x`. In addition, the importance of indentation for nested programs are hinted at.

Both concerns are not as prominent or non-existent in the our proposed concrete syntax. There are no nested structures, the declarations are side by side in one file, indentation is not necessary. Obfuscation due to misleading

whitespace is not present in the syntax. Automatic formatting is in therefore configured to unify the notation, giving writers some degree of freedom in visually structuring documents, an example is shown in Fig. 6.

```
1     // Unformatted
2 XML:Language
3 XSD : Language
4 XSD   subsetOf XML
5 xsdFiles: File +
6 xsdFiles elementOf  XSD
```

```
1 // Formatted
2 XML : Language
3 XSD : Language
4 XSD subsetOf XML
5 xsdFiles : File+
6 xsdFiles elementOf XSD
```

Fig. 6: These fragments depcit unformatted versus formatted code.

Tools, Refactoring Another automatized modification tool is the basic refactoring tool provided by *Xtext*. When resolving the cross references – as already pointed out in the concrete syntax specification and the *Xtext* illustration – it uses a function for navigation to AST elements, it is called the 'scope'. Usually this function is context sensitive, a *book* is for example not a valid element when assigning an *author*. In our case, the only restriction is the type, as determined by the assigned feature, e.g., the first operand of the relationship needs to be an entity.

The scope function is applied not only to the AST at hand but also to those of the imported and importing modules. Thereby, cross references between modules are possible, however limited to the import structure. All referring elements and references make up a group that is mapped by a name.

When selecting any reference or definition of said name, *Xtext* provides a tool called 'Rename Element', it operates on the group of references and definitions to change all names appropriately. This is particularly useful when refactoring a module without knowing of the entirety of referring modules. As references are automatically tracked, no manual chasing and searching of names needs to be performed by a writer, this operation is shown in Fig. 7.

Additional modifications on the AST that require insight into the semantics can be added to aid in developing megamodels. An example would be an operation that tries too substitute an entity's type with a more general one so that more relationship types are applicable. Such an operation would need to know if the type substitution changes the applied relationship type and therefore the plugin selection and, in conclusion, the semantics of the megamodel. *Xtext* provides mechanisms for easy modification of the AST, projecting them into the concrete syntax by pretty printing according to the

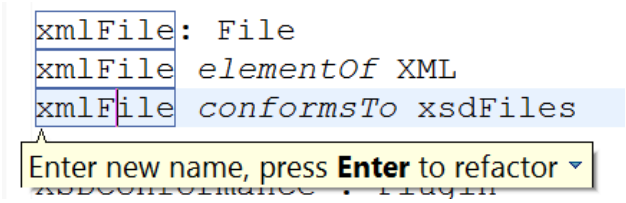

Fig. 7: The context action 'Rename Element' places an overlay on the selected element. All text modfications directly reflect in the depicted boxes, as well as those scattered across referring or referenced modules.

formatter, they are abstracted for multiple purposes as *semantic modifications* [9].

Exploration *Xtext* provides by default a 'Go to definition' option. When a user activates it from the context menu or by pressing a hotkey, the defining file will be opened with the responsible editor and the text location will be highlighted. This accounts for some of the navigation, which is a major part of the exploration. Moreover, *Xtext* exposes the currently selected AST element by tracking the text location to the parsed AST element. Those exposed elements can be accessed from anywhere in *Eclipse*. The presented implementation uses this to provide a mechanism for navigating the links that are assigned as bindings. This is used to explore a system's actual artifacts. In Fig. 8, an *XSD* file will be opened by the associated editor in *Eclipse*.

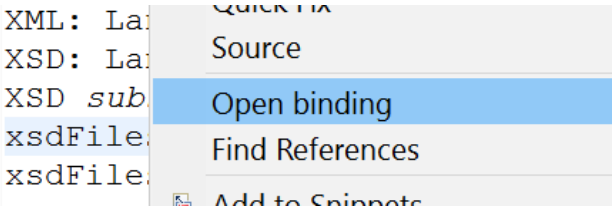

Fig. 8: An entry in the context menu of *Eclipse* is designated to open the binding of an entity. The command is distinguished from 'Go to definition', as the definition is concerned with the symbolic name given to the entity, where the binding is concerned with the real manifestation.

Code layout can also be adjusted. In *MegaL*, a coloring system is used to that allows giving prominent types a more distinguished tint. It is applied by annotating the type, as given by the comparison in Fig. 9.

6 Case study

The comprehensive features of the language will now be demonstrated at the hand of a technology used in industry, the *JAXB* framework for data binding. The executed case study will deal with modularity, orchestration, local and remote bindings, plugin configuration, transients and plugin API usage.

6.1 Preface

JAXB was developed as *JSR* 31 – the specification for *XML* data binding to *Java* – under the *Java Community Process*. The accepted document declares the domain of data binding in the *XML* technology space as being a common problem of non-trivial programs working with *XML* bindings. Understanding and creating *XML* files conforming to a *XSD* schema may require error-prone manipulation of documents on the AST level, i.e., direct *XML* modifications; manually binding data to *Java* objects on the other hand would require maintaining multiple type systems and the according serialization and deserialization methods, doing that is equally error prone [23]. The following quote illustrates this.

> It would be much easier to write XML-enabled programs if we could simply map the components of an XML document to in-memory objects that represent, in an obvious and useful way, the document's intended meaning according to its schema. Of what classes should these objects be instances? [...] In general, [...] classes specific to the schema being used will be required. Rather than burden developers with having to

```
// Uncolored
File < Artifact
Language < Set

XML : Language
familyTree : File
petHistory : File

familyTree elementOf XML
petHistory elementOf XML
```

```
// Colored
@Color 'red' File < Artifact
@Color 'cyan' Language < Set

XML : Language
familyTree : File
petHistory : File

familyTree elementOf XML
petHistory elementOf XML
```

Fig. 9: These fragments depcit uncolored versus colored code.

> write these classes we can generate the classes directly from the schema, thereby creating a Java objectlevel binding of the schema.

(Fialli et al., [23], pg. 2)

As the quote states, binding is concerned with bridging the gap for data in manifested or persisted form and programs that translate them. The technology at hand is concerned with both type and instance level, as it mentions *XML* and *in-memory objects*, as well as classes and the *XSD* schema. Thereby they identify a set of languages and technologies that are essential to software referring to *JAXB* as a method of data binding, as given in the following listing.

- *XML* is – as an essence to the problem domain – defined as an entity, a language that is.
- The given text fragment identifies one of the core specialties of this implementation: *in-memory* objects — in the thesis referred to as *transients*.
- The correspondence of an *XML* document's components to the given *in-memory* objects is explicitly stated.
- A schema that defines *XML* documents in their structure and meaning is listed, by the nature of *XML* technology, this is given as an *XSD* schema.
- *Java* is named as a context and as the target for generating object level bindings. In addition to listing the language as a target, the quote highlights that the *Java* code is *generated* and, in fact, the target of a process.
- As previously given, the *generator* is identified as the ability to generate classes from a schema.
- To some degree, the fact that *XSD* schemata are said to have an 'intended meaning' and that *Java* types are deducted thereof reveals the notion of schema files share a common *problem language* with the *Java* types.

This brief glance at the identified properties – only contained in a single paragraph – will be elaborated in this case study. The *101companies* community project serves as a foundation. In addition to the identified technologies,

programming languages and their relationships, they expose an abstract software concept focused on a fictional *Human Resources Mangagement* scenario. Different languages are used and a multitude of patterns are exercised to illustrate their relations in a common problem [21].

The system is described as follows: a company has a name and is made up of many departments. Such a department is named as well, can aggregate sub-departments, and itself consists of many employees and a single manager. They latter are defined to be of type *Employee*, which is attributed with name, address and a salary. There are predefined functions that allow cutting the employees' salaries. One of the exercised examples is *JAXB* using *composition*, the solution is written in *Java*. Based on that is the presented case study.

6.2 Taxonomy for software linguistics

In order to model a system based, we require to have some basic types, an ontology for software systems. In previous *MegaL* implementations, a set of entity and relationship types was shipped, called *prelude*. It's designed to cover linguistic aspects in megamodeling and has been developed ever since it' inception. The initial approach takes some of its types from inter-model and model/metamodel relationships, and others from *UML* to model referral, dependence or composition. In addition, some notations for properties like being an element or subset of a set are given [20] — a file being written in a specific language is given as an 'elementOf' relationship. These types have been refined when megamodeling was applied to a parser generator system: *ANTLR* [32]. The *prelude* underwent another iteration of assessment and modification, addressing some insights arising from the case study at hand, it is depicted in Fig. 10.

Let's have a look at the entity types and relationships in detail, starting off with the top-most entities.

- **Entity**: This is the root of the type lattice, there's no other meaning assigned to it.
- **Set**: The algebraic notion of a set, that is, a collection of distinct elements. This is an abstraction of the more concrete types of *Languages* and *Functions*, which are listed in the following. Sets are usually not directly referred to, but their properties are shared among the subtypes.
- **Language**: A specialized form of a set that is particularly useful when talking about linguistic architecture, entities of this type have been mentioned in the course of this thesis. Language is not equal to the

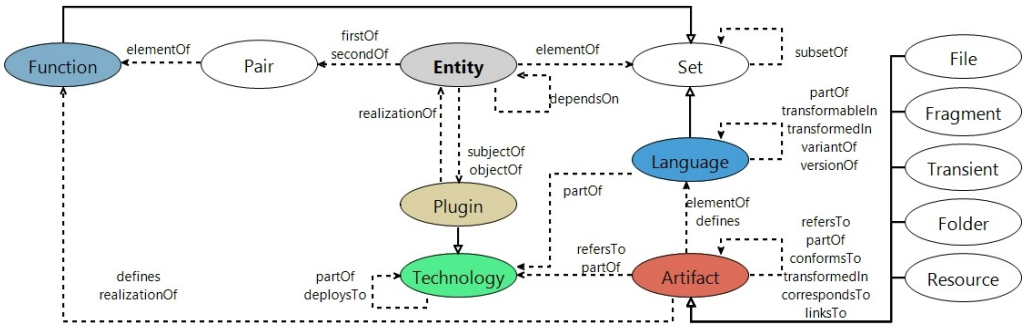

Fig. 10: The prelude is given in visual notation. As *MegaL* is a text-based representation of a graph, reading a diagram thereof is trivial. In the depiction, circles present entity types. The solid arrows show a subtype relationship, e.g., *Function* is a specialization of the *Set*. Dashed arrows list all the applicable relationship types between instances of the respective entity types. In this diagram, subtyping *Entity* is omitted, as this is the basic setup and the arrows would obfuscate the graphic.

notion of sets, there are relationships that do not apply to sets in general. The following example shows a common usage of the type.

```
XML : Language
XSD : Language
XSD subsetOf XML
aFile elementOf XSD
```

- **Function**, **Pair**: A *function* is also a subtype of sets. This is due to the notation of functions as relations with specific properties. A (binary) relation for example is a collection of tuples (or pairs). Functions are mapping an input to a single output, this property does not hold for relations in general.

  The notation of a mapping rule in functions is given as a definition of one particular input and output as a pair. In the *MegaL* syntax, this notation is automatically generated from a syntactic construct, as demonstrated in the following.

```
JavaFormatter : Java → Java
JavaFormatter(aFile) ↦ aPrettyFile
```

  The first line is the specification of domain and range, the second defines a new pair, thereby giving one mapping rule.

- **Technology**: This entity type classifies a group of conceptual entities that aggregate other technologies, tools, languages et cetera.
- **Plugin**: The main coordination entity to assign the semantics to a megamodel. There are a number of relationships that help specifying their meaning and the involvement with other entities in a model, in addition to the potential relationships derived from technologies. Plugins can for example be associated with entities, which offers a high descriptive capability in combination with model/metamodel relationships. An instance of this will be shown in the case study.
- **Artifact**: This type is clarified in a set of specializations and describes all manifested and non-manifested things in software systems. Covered are for example single files or collections thereof, web-based resources, as well as transient artifacts.

In addition to the entity- or node types, the edges between them – the relationship types – are partially listed hereafter, the more trivial types or those mentioned in the entity type guide are omitted.

- *partOf*: The relationship type is defined on a multitude of entity type combinations. It specifies a part/whole relationship — a composition. This is in some cases 'physical', as in files being part of a folder, sometimes conceptual like for languages or technologies. Take for example an embedded domain specific language. It is in fact valid to reason about it as an individual, but it is *part of* a host language.
- *defines*, *realizationOf*: In the domain of artifacts and functions, definition and realization describe two complementary understandings. The prior – *defines* – sees functions as a consequence of a defining artifact, where realization implies that a function is the conceptually strong entity, the artifact is a mere instance of it.
- *variantOf*, *versionOf*: In these relationships the languages are structured in two dimensions. Variants can be seen as a parallel level for a group of similar languages, where versions of a language denote the time and development axis.
- *transformableIn*, *transformedIn*: The relationship type for transformability denotes the transformation *option* itself, as opposed to the actual instance denoted by something being transformed into another. A transformation may either apply to languages or artifacts, where in the prior case, a transformation can change what is accepted by the

language and what is not, and in the latter case, an artifact's form is altered.

- *conformsTo*, *correspondsTo*: The model to metamodel or instance to schema conformance is applied between the artifacts defining them. Correspondence is concerned with strong connections between a pair of artifacts, in that their components on either side are related to each other. In some cases, this can be a bidirectional mapping.

This set of types, our classification hierarchy on software system entities and relationships, is hereafter demonstrated in action as an exemplary *JAXB*-based system is modeled.

6.3 Megamodeling *JAXB*

There are a number of aspects that need to be modeled when describing the linguistic architecture of a system using *JAXB*. There are certain levels of the data-binding technology, especially when having the type system and the instances thereof in mind. The graph in Fig. 11 displays the structure of modules and groups them with regard to their domains.

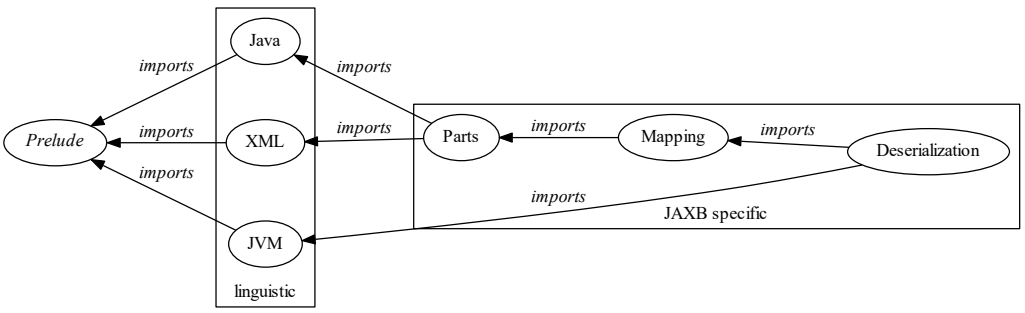

Fig. 11: The imports between modules are listed. The boxes group different aspects, the *linguistic* section is for example applicable to a broader domain of software, where the other is highly problem specific.

In the linguistics section there are three foundation modules for the megamodel: *XML* is given as one side of the data-binding, *Java* and *JVM* as the other. Splitting *Java* and *JVM* is due to the fact that there are a lot of languages besids *Java* that run on the *JVM* — take for example *Clojure* and *Scala*.

The next megamodels at hand are the structural model, called *Parts*, which refers to the languages that *JAXB* is concerned with, i.e., *XML* and

Java. Based on *Parts* are the type and instance level bindings in *Mapping* and *Deserialization* respectively. The complementary case of *Serialization* – as the opposite direction of instance level data-binding – is trivially constructible from thereon.

6.3.1 XML

The prelude has been demonstrated in the previous section, therefore the first model to be explained will be the *XML* module. It is concerned with basic facts of *XML*, focusing on the model/metamodel relationship.

```
model XML
import Prelude

XML : Language
XSD : Language
XSD subsetOf XML
```

The first entities and relationships are trivial, they define both schema and instance language and points out that the latter, *XSD*, is a subset of *XML*. Thereby, the fact that any element of *XSD* will be a valid element of *XML* is established. In the following, an actual example set will be defined, according to the practice for standalone modules.

```
xsdFiles : File+
xsdFiles elementOf XSD

xmlFile : File
xmlFile elementOf XML
xmlFile conformsTo xsdFiles

xmlFile = 'workspace:/mycorp/inputs/sampleCompany.xml'
xsdFiles = 'workspace:/mycorp/inputs/Company.xsd'
```

At first, a group of schemata is listed. *XSD* provides mechanisms of deriving a schema. All the required files can then be combined and used to validate schema conformance [41]. After defining the schemata, the *XML* file is given and conformance is stated. At the end, two files are given as bindings. In the example, only a single schema is used. After stating the facts, they need to be proven or at least supported by evidence. The module at hand is configured as follows.

```
// Evaluate the elementOf relationship for XML files
XMLAcceptor : Plugin
XMLAcceptor partOf StringAcceptor
```

```

XMLAcceptor realizationOf XML

XMLAcceptor = 'classpath:plugins.xml.XMLAcceptor'

// Evaluate the elementOf relationship for XSD files
XSDAcceptor : Plugin
XSDAcceptor partOf StringAcceptor

XSDAcceptor realizationOf XSD

XSDAcceptor = 'classpath:plugins.xml.XSDAcceptor'
```

The 'elementOf' test is delegated to a string acceptor, a plugin that can decide if a file is actually element of a language based on the text it contains. The plugin is enriched by component plugins that are associated to languages. If an artifact is stated to be element of a language, the appropriate 'acceptor' is picked. In the following fragment, the remaining configuration is located.

```
// For the pair of XML and XSD, evaluate the conformsTo relationship
XSDConformance : Plugin
XSDConformance partOf ConformsToEvaluator

XML subjectOf XSDConformance
XSD objectOf XSDConformance

XSDConformance = 'classpath:plugins.xml.XSDConformance'
```

The conformance evaluator is plugged in a similar fashion, the *XML* to *XSD* conformance is associated with the corresponding languages and then linked to the implementation, which is given by a classpath URI. The actual code checking the relationships is based on the *XML* and *XSD* API provided by *Java*.

6.3.2 Java and reasoning

The *Java* module is similar and will not be discussed in detail, as the previous model already illustrated how a language and the element checks are connected. The only differing thing is the binding, instead of a local file, it points to a program on the web. Abstracting such locations allows code to be written without regard of the actual place an artifact resides in.

```
model Java
import Prelude
```

```

// A langauge and one of its files
Java : Language
javaFile : File
javaFile elementOf Java
javaFile = 'http://introcs.cs.princeton.edu/java/11hello/HelloWorld.java'

// The plugin code stating the semantics
AcceptJava : Plugin
AcceptJava partOf StringAcceptor

AcceptJava realizationOf Java

AcceptJava = 'classpath:plugins.java.JavaAcceptor
```

Inferring bindings After listing the two language centric models, there is some hidden knowledge that is encoded and retrieved in the background. It's observable that only the files are bound to artifacts, the languages however are not explicitly linked. There is actually a *reasoner* that is applied, iff the language has not been bound by an author. In our case, the 'LanguageReasoner' will look at the instances and – based on their name – link them to *dbpedia*. For the entity *Java* a binding to `http://dbpedia.org/page/Java_(programming_language)` is generated.

6.3.3 The parts of *JAXB*

Another module where reasoning comes in handy is the following. In here, parts of the example are listed, that is, the components of *JAXB* as a technology. The application for reasoning is given by the naming scheme. Upon closer examination, it's observable that every name has a dot in the middle. This is a shorthand for expressing the 'partOf' relationship. In the reasoning phase $a.b \vDash a.b$ partOf a is executed.

```
model Parts
import XML
import Java

JAXB : Technology
JAXB.Library : Technology
JAXB.Generator : XSD → JAXB.Java
JAXB.Java : Language
```

```
JAXB.Java subsetOf Java
```

Aside from that convenient notation, the actual issue of *JAXB* parts is discussed: itself is listed as a technology, it's accompanied by a function – the generator – and a library it requires to execute the generated code, the latter it is usually some *Java* library dependency, where the function is realized by an executable program that takes *XSD* documents and creates the according *Java* code.

In addition, a fact is hinted at that would go unnoticed otherwise: the code *JAXB* generates is not completely exercising *Java* features, it only uses a specific subset — for example only being concerned with 'classes' and neglecting 'enums' or 'interfaces'. Its therefore specified as a new language – the vocabulary that *JAXB* requires – and set into relation with the full *Java* language.

There are no bindings specified in this model, it only serves as a structural view of the system at hand.

6.3.4 Mapping

Now that the core languages are defined and *JAXB* is laid out, the type level mapping will be discussed. As already mentioned, a program is called that creates *Java* types for *XSD* schemata. The set of resulting *Java* classes is therefor defined. Although we can reuse the 'elementOf' relationship semantics, we need to declare a *group of files* rather than a singleton. As *XSD* schemata usually contain a variety of types, multiple classes will be generated, each in an individual *Java* file.

```
model Mapping
import Parts

// There's only a single Java file in the imported module, declare a group
javaFiles : File+
javaFiles elementOf JAXB.Java

// Bind them to a folder's content, XML and XSD schema stay the same as in
// the example model
javaFiles = 'workspace:/mycorp/src/company/xjc/'

@Binding 'workspace:/mycorp/build.xml'
JAXB.Generator(xsdFiles) → javaFiles
```

The last declaration is interesting: instead of applying the generator function to obtain the *Java* classes, both are already present at the time they are

checked. Finding evidence that the *XSD* schemata are input to the generator, and the *Java* files are its output requires finding traces of the *generator invocation*. We assume that there's a structured call of the function, given by a *build script*. In this particular instance, an *ANT* file (workspace:/mycorp/build.xml) configures the build process of a project. The file contains evidence that *JAXB*'s generator is called, as given in the following fragment of the build script, *XJC* is the generator program.

```
1 <!-- Generates the model classes from the schema -->
2 <exec executable="xjc">
3   <arg value="${xsdfile}" />
4   <arg value="-d" />
5   <arg path="${src}" />
6   <arg value="-p" />
7   <arg value="org.softlang.company.xjc" />
8 </exec>
```

We observe the fact that, regardless of the input and output, both sides of the generator application are concerned with the same domain. To point that out, a new 'virtual' language is introduced, as given in the subsequent model segment.

```
14 // A language that both participants describe
15 aLanguage : Language
16 aLanguage = 'http://www.company.softlang.org/company.xsd'
17 javaFiles defines aLanguage
18 xsdFiles defines aLanguage
```

As we model our example around the *101companies* domain, this language is linked to a related *URI*. Both *Java* and *XSD* files are said to define this language. To check this, both documents (or document groups) are checked for evidence of defining this language. In *XSD*, it can be found in the first lines, as the namespace URI points directly to the one bound to 'aLanguage'. In the folder containing the *Java* files, a package meta-data file states that the surrounding classes are implementing the namespace as well. Therefore, evidence of the virtual language is retrievable.

Not only do the schemata and classes define the same language, the structure of *XSD* components is in fact reflected in the generated *Java* classes. This means that all the metamodel nodes given in *XSD* have a one-to-one mapping to *Java* classes or their members. That fact is encoded by stating correspondence for the two.

```
19 xsdFiles correspondsTo javaFiles
```

The module is completed by the declaration of its plugins, as given hereafter.

```
// Implements finding the call to the generator
FindBuildScriptAsElementOfEvidence : Plugin
FindBuildScriptAsElementOfEvidence partOf ElementOfEvaluator

// Asserts correspondence between schema andclasses
XSDJavaCorrespondence : Plugin
XSDJavaCorrespondence partOf ConformsToReasoner

// Checks if the definition relationship is valid
MatchingLanguageDefinition : Plugin
MatchingLanguageDefinition partOf DefinesEvaluator

// Extract the namespace URI for Java
PackageInfoNSURIExtractor : Plugin
PackageInfoNSURIExtractor realizationOf Java
PackageInfoNSURIExtractor partOf MatchingLanguageDefinition

// Extract the namespace URI for an XSD schema
XSDNSURIExtractor : Plugin
XSDNSURIExtractor realizationOf XSD
XSDNSURIExtractor partOf MatchingLanguageDefinition
XSDNSURIExtractor = 'classpath:plugins.jaxb.XSDNSURIExtractor'
```

6.3.5 JVM

Java and *XML* have been introduced as languages already, as they were essential to the primer models. After we described the static aspects of the system and are heading towards the deserialization, we need to be able to talk about *JVM* as the language of our in-memory objects. The following module will establish the linguistics for this problem.

```
model JVM
import Prelude

JVM : Technology
JVM.Bytecode : Language
JVM.Objects : Language
```

The module describes the central languages: the executed byte-code and the status of a program, the in-memory 'Objects'. There's no reasoning

about the languages on their own, the interesting part comes hereafter, as the instance is declared.

```
objectGraph : Transient
objectGraph elementOf JVM.Objects
```

We had a variety of manifested artifacts up to this point. This is the first time, a module mentions transient objects. We neglect an example binding to assert this module's validity, as we are instrumenting it in the next module: the deserialization.

6.3.6 Deserialization

At last, the *JAXB* core feature of obtaining a runtime object form a serialized file will be tackled. In fact, the example of *101companies* will be executed, a persisted company will be loaded. The first thing that is stated in the module are references to the foundation modules, i.e., those defining mapping and *JVM*. Then, the deserialization is described, as given by the following statements.

```
model Deserialization
import Mapping
import JVM

deserializationCode : File
deserializationCode elementOf Java
deserializationCode defines deserialization
deserializationCode refersTo JAXB.Library
deserializationCode refersTo javaFiles
deserializationCode = 'workspace:/mycorp/src/features/Persistence.java'
```

There's a code file that is written in *Java*. It is the one that defines the deserialization function which maps the *XML* file to a *Java* object graph. The code needs on the one hand *JAXB*'s library to perform the operation, and on the other hand it depends on the generated *Java* classes. The deserialization is then given as a single method that is contained in the linked code file. If applied to the *XML* file, it results in an *JVM* object graph.

```
deserialization: XML -> JVM.Objects
deserialization(xmlFile) |-> objectGraph
deserialization = 'classpath:features.Persistence#deserializeCompany(File)'
```

The stated facts need to be checked now. The 'elementOf' relationship is exported by *Java*. Checking if a function is defined by a *Java* file is performed by looking for the linked signature. Referral to types and libraries is asserted by looking into the import section of the *Java* file.

The transient is now bound, we can argue about it, as the deserialization method provides the in-memory object via reasoning and the plugins can inspect its structure. The following relationships state that the *Java* files are compatible classes, i.e., the in-memory object starts at on of the generated classes. In addition, the correspondence of *XML* file and the in-memory object is stated: traversing the *XML* document reveals the same structure as in the transient value. A byproduct of the correspondence check is a set of trace-links. They connect elements in the document with values in the transient, and are – as execution output can be inspected by other tools – displayable in a 'trace view' shown in Fig. 12.

```
objectGraph conformsTo javaFiles
xmlFile correspondsTo objectGraph
```

xmlFile	objectGraph
/company/department#0	org.softlang.company.xjc.Department@5fd1a6aa
/company/department#0/employee#0	org.softlang.company.xjc.Employee@1a56a6c6
/company/department#0/employee#0/address: Utrecht	Utrecht
/company/department#0/employee#0/name: Erik	Erik
/company/department#0/employee#0/salary: 12345	12345.0
/company/department#0/employee#1	org.softlang.company.xjc.Employee@748e432b

Fig. 12: The trace view depicting links between the file and the actual in-memory object graph. It's observable that the elements in *XML* corresponds to either data-types (like text or numbers) or to classes derived by the mapping.

In the remaining *MegaL* module, the appropriate plugins are specified.

```
// Simple referral test
JavaRefersToJava : Plugin
JavaRefersToJava partOf RefersToEvaluator

JavaRefersToTechnology : Plugin
JavaRefersToTechnology partOf RefersToEvaluator

// Application of a function, associated with the elementOf
// relationship for pairs
FileToObjectFunction : Plugin
FileToObjectFunction partOf ElementOfReasoner

JavaFileDefinesStaticFunction : Plugin
JavaFileDefinesStaticFunction partOf DefinesEvaluator

// Correspondence and conformance
ObjectConformsToJavaFiles : Plugin
```

```
ObjectConformsToJavaFiles partOf ConformsToEvaluator

XMLFileCorrespondsToJavaObject : Plugin
XMLFileCorrespondsToJavaObject partOf CorrespondsToReasoner
```

6.3.7 Metrics

The implementation of *JAXB* as case study took a certain amount of code to describe the linguistic architecture, the semantic configuration and the plugins that evaluate it. They are listed in Tab. 2. A split is applied for the *MegaL* metrics, as some of the code is universally usable. The *Java* metrics are also split, as they include utility code that is in a similar fashion universal.

Module metrics	#Ann	#Dec	#Bind	#Plug
Prelude	91	113	3	104
JDOM	0	5	2	3
Java	0	6	2	4
JVM	0	5	0	0
XML	0	16	5	11
JaxbParts	0	6	0	0
JaxbMapping	0	19	9	17
JaxbDeserialization	0	21	8	18
Total	91	191	29	157
excl. Prelude	0	78	26	53

Java **metrics**	#NOC	#NOM	#NCLOC
Model shell		89	479
Runner		7	235
Plugin utilities	5	88	521
JAXB	16	22	883
Other	11	8	393
Total	34	214	2511
excl. API	32	118	1797

Tab. 2: Module metrics for the case study: number of annotations (#Ann), number of declarations (#Dec), number of bindings (#Bind), number of plugin related declarations or bindings (#Plug). Code metrics of the plugins: number of methods (#NOM), number of classes (#NOC), non-comment lines of code (#NCLOC).

7 Analysis

Now, the implementation will be judged based on requirements and intended contributions. The prior were extracted in discussions with developers of the previous versions, the latter were developed to accommodate for desired features and for observations during the usage of previous implementations. With regard to the development structure that sorts the versions of *MegaL*, new and established technologies in the domain of DSL development and modelware have been adopted. As the previous version used *ANTLR*, some of the grammar could be reused, but a majority was rewritten, as there are requirements for rule to AST mapping and new syntactic constructs.

The proposed algebraic model has been introduced after multiple iterations of plain-*EMF* calculations have proven ineffective or incomprehensible for experts in a certain domain that are not familiar with the concepts this modifications require. Especially in the context of reasoning, the algebraic model has shown to be helpful, for example, creating graph structures was alleviated due to the possibilities of pointing to an element that is created at a later point — in a symbol and constructor based approach, this would require initializing all elements and *then* connecting them.

The development was – after the initial migration and adaption to the new technologies – guided by the implementation of a megamodel in the case study given by § 6. Some of the existing features were extended as unification of different solutions that tackle other problems with current approaches became reasonable. Features of the semantics were developed after certain problems appeared recurring and prominent, the reasoner solution for example was enriched by a secondary mode that allowed skipping a portion of the recursive identity closure.

7.1 Analysis versus the requirements

In the corresponding section, a set of requirements has been defined, according to the desired features. Their realization was discussed extensively in the design section, their fulfillment will be analyzed hereafter.

7.1.1 Analysis of 4.5.1 (RE-Int)

The language and its editor is implemented in *Xtext*, which directly integrates into the IDE *Eclipse*. When a megamodel is executed by the evaluation engine, the results are fed back using an established communication channel: the *EMF* diagnostics framework.

Seeing that editing and exploring a megamodel, as well as running it, are integrated into a standardized and wide-spread IDE, we regard this requirement to be *fully met*.

7.1.2 Analysis of 4.5.2 (RE-SAU)

Xtext provides a standalone parser, it is packageable so that all *Java* programs can refer to it and use it. There's a minimum of setup required to enable both the strongly typed AST and the *MegaL* parser. The execution can be configured accordingly and the *Eclipse* workspace relative URIs are mapped to a certain base path, the resource API of *Eclipse* is lifted. As all features that are available from the IDE are presented to non-IDE programs, this requirement is seen as *fully met*.

7.1.3 Analysis of 4.5.3 (RE-Ind)

The URI mechanism implemented in this language is platform independent, there are no mismatching path conversions or dependencies to absolute folder placement. The *Eclipse* IDE and *Java* are distributed to *Windows*, *OSX* and *Linux*, covering the majority of operating systems.

The usage of platform independent URIs is however not enforced. This helps on the one hand to develop models that are in a 'bad style' but only used for experimentation. On the other hand it might result in a module that is not usable. In conjunction with the few exceptions of systems that do not run *Java*, the requirements is marked as *almost fully implemented*.

7.1.4 Analysis of 4.5.4 (RE-Ext)

The core feature of the implementation is integration of megamodel and plugins. The latter can be implemented by sufficing a small interface. The plugins are however limited to *JVM* based code, other languages and system need to be addressed by wrappers, *JVM* based interpreters or over the network.

In general, *Java* is expressive enough and a lot of dialects and languages basing on it exist. This allows a majority of developers to write plugins for *MegaL* and patching the entire evaluation process — therefore, the requirement is accepted as *fully implemented*.

7.1.5 Analysis of 4.5.5 (RE-DFA)

A driving force behind the development of the API proposed in 5.5 is the *case study* which required a comprehensive description and evaluation. From

this nature alone one can assume that the API is sufficient to address domain specific problems. A problem with this assumption is that software systems are highly diverse, a single case study can not justify the generality. We hope that the modularity of *MegaL* and the accessibility of its editor help to distribute the implementation, thereby making more linguistic models available and as – a result – having a deeper insight into the requirements for such an API. In conclusion, the requirement 4.5.5 (RE-DFA) is considered *present*, but *subject to assessment* delegated to future work.

7.1.6 Analysis of 4.5.6 (RE-Tra)

A core specialty of the implementation is the introduction of transient state, the case study at hand demonstrated this behavior. There are some drawbacks to the method, for example, it is limited to *JVM* based programs. Another major issue is the need of an accessibility method, a gateway into the software. This reduces the generality of our approach as some of the software systems may be proprietary and therefore not be eligible for accessibility methods. The approach does however show the usefulness of transients. The deserialization requires by its nature a description of the object graph it creates. Without transients, such a case study could not be exercised. Similar to 4.5.5 (RE-DFA), this requirement is regarded as *present* but *subject to discussion*.

7.1.7 Analysis of 4.5.7 (RE-Mod)

In the presented implementation, modules are a key milestone. The approach of allowing a module's elements to be overwritten creates the possibility of exporting a self-sufficient example that is then utilized in describing a software system. By implementing a module with an example, its behavior is clarified and demonstrated.

There are however use-cases that could require a module to be instantiated twice, for example if a software system uses more than one *Java* file, which is obviously a common situation. To address such an issue, the current orchestration variant could be enhanced by allowing a module to be imported multiple times in conjunction with individually renaming elements the module exports. This language feature is thereby also in progression and *subject to discussion*, for concise software systems it is however *present*.

7.1.8 Analysis of 4.5.8 (RE-Syn)

The syntax is clearly structured, the syntactic sugar is minimal. Definitions of elements in the type lattice share the concept of $<$, denoting a deriving

definition. The product symbol in the relationship type declaration resembles mathematical specifications. Instantiation of entities is given by a colon, similar to other popular languages, the relationship application is trivial. This issue tends to be subjective, but with regard to other triple based languages, we assume this requirement to be *fully met*.

7.1.9 Analysis of 4.5.9 (RE-App)

As with the syntax definition, the code formatting is a subjective matter. In the presented implementation, the layout is configurable, accounting for different flavors and tastes of megamodeling. This should suffice for a wide variety of layout styles, the requirement is therefore set to be *fully met*.

7.1.10 Analysis of 4.5.10 (RE-EF)

A core ingredient of the implementation is the reliance on *Xtext*. This includes a multitude of editing features that already assist in writing and exploring a model. Complementary to the AST centered features, the *MegaL* implementation provides navigation to the artifacts a model is concerned with. As this is all served in a package for *Eclipse*, readers and writers alike are benefiting from the integration with other domain specific tools. A benefit of this implementation is the integration of *JDT* tools that allow developing plugin code side by side with a megamodel. That way, a rapid prototyping and experimentation based plugin development style is possible. The feedback is reasonably connected to the model editor. In conclusion, we se this feature as *fully met*.

Summary After analyzing the entirety of requirements, it is notable that the language centric features are not as well developed as the technical requirements. This is due to the language being an in-development version. API, modularity and orchestration, as well as transients will be assessed in detail, the language might be adapted after a comprehensive evaluation of the newly provided features.

7.2 Research questions revisited

In the opening of this thesis, four research questions were put. They are highly concerned with the editing experience, thereby their answers relate to the requirements. An explicit listing of the research questions' answers is discussed hereafter.

The **RQ1** asks for ways to improve editing experience. The requirements 4.5.1 (RE-Int), 4.5.8 (RE-Syn), 4.5.9 (RE-App) and 4.5.10 (RE-EF) are explicitly directed at this. The direction illustrated in the question itself were adopted and implemented. As a result its valid to say that editing experience can be improved by providing a tool focused on the problem domain, which in this thesis is done by utilizing a language workbench, namely *Xtext*.

In **RQ2**, integration with other tools is inquired. An explicitly arising requirement is 4.5.1 (RE-Int), as it states that the tool should be integrated in an IDE. Aside being in the same place as other editors, our implementation relies on a common interchange format with a rich tool support: *EMF*. Although an abstract syntax specification might be enough to read a model, having it conform to another level of specification – like it was laid out in Fig. 1 – alleviates other tools to comprehend the model. Simplicity of a model assists development of future editors, the designed model is fairly straightforward.

The question defined in **RQ3** asks for improvement of the evaluation. A step towards an improved evaluation is the plugin interface which may inspect the entire model in an alleviated way. Reasoning is integrated and can help to unify existing resolution behavior and deductions translating inherent knowledge. A great improvement are the *transients*, they can encode key artifacts that were not addressable in the former implementations.

Looking at the performance characteristics, multiple positions exists where parallel execution is possible. Right now, the approach relies on executing a plugin and letting self-determine its termination. When the selection based approach, given as *dispatching* in Tab. 1, appropriateness of a plugin can determined *before* it is executed. The performance sufficed for the executed case study, larger models however need to be subjected to assessment.

In the final question, **RQ4**, plugin capabilities for models are requested. In the requirements, this reflects in 4.5.4 (RE-Ext) and 4.5.7 (RE-Mod). The prior is concerned with the mechanism itself where the latter accounts for orchestration. Unifying the model facts and the plugin configuration allows benefiting from the import semantics. A simple variant of this *orchestration* has been laid out and there are already some insights on multiple instances of an exported model element.

7.3 Threats to validity

In addition to the analysis vs. requirements and the answered research questions, a short set of threats to validity will be listed. Some of them are arising from said analysis, some of them are insights of the development and assessment process. They are divided into internal and external threats, where

the prior are concerned with the implementation design and the latter are focused on the generality.

Int. 1: The semantics have been defined formally in a set of algebraic categories and by a number of algorithms. The semantics they execute is however not asserted for conformance to the definitions in previous *MegaL* implementations. Additionally, reasoning has been woven into the execution and not been formally analyzed.

Wellformedness has however been established by a set of constraints that are close to the original, evaluation does not exceed visiting all model elements, and reasoning is – by being applied ot the unmodified function input – not dependent on side effects.

Ext. 1: There are concerns that the proposed ontology and this tool in particular might not be appropriate to cover all aspects required to model linguistic architecture. The given types of entities and relationships could fail to describe a property that has not been conceived of at the time developing them. This is a common problem with newly introducing a modeling notion and creating tools, the domain of software is highly diverse. Exercising linguistic architectural description for a variety of software might help in identifying shortcomings in the API and the ontology.

Ext. 2: The metrics given in the case study are concerning a single case, there can be no claim for generality, as the example might be simpler to describe than the average software. Given there are some problems that are more complex to describe, the numbers might vary strongly. In the worst case, for every relationship, there's a single plugin checking it.

In some systems, this cannot be avoided. The development process and the API can however eliminate highly complicated functions or at least limit them to the essence. We hypothesize that with a larger corpus of described systems, the reusable parts will accumulate. A structured way of making a plugin reusable should in that case be notated — similar to the pattern of giving an exemplary binding, thereby asserting model appropriateness and showing the intended meaning.

8 Conclusions

The program at hand successfully implements an integrated editor using a parser that is also runnable in standalone mode. The AST constructed by the parser is usable by other applications, as it conforms to *EMF*'s standard. The AST is translated into a string and set based representation, in which its semantics are executed. The latter are defined as a collection of plugins that may be introduced by a users. Their development is alleviated by an embedded DSL that focuses on input and output operations on files and other resources, as well as control flow lifting. The execution of semantics is performed during the model editing as to assist model creation by direct feedback. A case study examines an industry standard system.

8.1 Future work

In the course of this thesis, a few directions have been identified that could guide the future development of the software. In addition to the language and user-experience centric enhancements, there are some possibilities from using some technology features that have been neglected for now, and some potential directions concerned with the entire environment, including programming language and targeted IDE. In this last section, an outlook on future work is given.

Choice of language The system is in its current form implemented in *Java*. This is due to the fact that the core is *Java* based – be it *Eclipse*, the model framework *EMF* or the editor generator system *Xtext*. The evaluation model however is based on paradigms that are more related to functional programming and pattern matching. The readability, maintainability and the performance may be greatly boosted when switching the programming language, at least for the evaluating parts. With regard to the *JVM* nature of the surrounding frameworks, *Scala* may be an option worth considering, as it does not require a too great effort to integrate it with *Java*.

Derivation of features in EMF One of the key features of the system is to derive facts and add them to the model in place. The presented approach implements this by translating the model into the set based representation, expanding the model is a simple operation of set union that – given the nature of persistent data structures – is of high performance in both access time and space.

As this kind of reasoning feature is desirable in multiple systems and the sidetrack of having a set based model is not always a good option, ways of

easily integrating derivation into models should be investigated. There are methods of extending *EMF* data structures even in derived, non-manifested ways, but their application is troublesome and hard to keep in a comprehensible shape.

Empirical evaluation of the software Some requirements are focused on the accessibility of language and editor for users. That requirements are – simply put – focused on the *ease of use*. During the development of the program and when applying it in the case study, we found that the development performance, especially with refactoring options, drastically increased. Such a claim should however be evaluated in a structural way.

There are a number of methods to assess the softwares usefulness, the most significant way would however by the experimental evaluation, where test subjects are given a set of tasks and the different treatments, i.e., using the software as opposed to editing in plain text mode.

Addendum

The implementation is available from *GitHub*, it can be found in `https://github.com/avaranovich/megal-xtext`. Continued development on this branch might lead to different versions than the one discussed in the thesis. A commit of the publication time of this thesis should therefore be picked.

References

[1] Eclipse documentation on JDT Core.

[2] D. Amyot and J. B. Yan. Flexible Verification of User-defined Semantic Constraints in Modelling Tools. In *Proc. of CASCON 2008*, pages 81–95. ACM, 2008.

[3] M. Bacıková, D. Lakatoš, and M. Nosál. Automatized Generating of GUIs for Domain-Specific Languages. In *Proc. of SLE 2012*, volume 7745 of *LNCS*, pages 27–35. Springer, 2012.

[4] M. C. Bastarrica, J. Simmonds, and L. Silvestre. Using megamodeling to improve industrial adoption of complex MDE solutions. In *Proc. of MiSE*, pages 31–36. ACM, 2014.

[5] H. Behrens, M. Clay, S. Efftinge, M. Eysholdt, P. Friese, J. Köhnlein, K. Wannheden, and S. Zarnekow. Xtext user guide. 2008.

[6] G. Bergmann, b. Hegedüs, k. Horváth, I. Ráth, Z. Ujhelyi, and D. Varró. Integrating Efficient Model Queries in State-of-the-Art EMF Tools. In *Objects, Models, Components, Patterns*, volume 7304 of *LNCS*, pages 1–8. Springer, 2012.

[7] G. Bergmann, Z. Ujhelyi, I. Ráth, and D. Varró. A graph query language for EMF models. In *Theory and Practice of Model Transformations*, volume 6707 of *LNCS*, pages 167–182. Springer, 2011.

[8] T. Berners-Lee, R. T. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986, 2005.

[9] L. Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, 2013.

[10] L. Bettini, D. Stoll, M. Völter, and S. Colameo. Approaches and Tools for Implementing Type Systems in Xtext. In *Proc. of SLE 2012*, volume 7745 of *LNCS*, pages 392–412. Springer, 2013.

[11] J. Bézivin, F. Jouault, and I. Kurtev. TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In *Proc. of GPCE 2006*, pages 249–254. ACM, 2006.

[12] J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez. Modeling in the Large and Modeling in the Small. In *Proc. of MDAFA 2003 and MDAFA 2004*, volume 3599 of *LNCS*, pages 33–46. Springer, 2005.

[13] C. Brun and A. Pierantonio. Model Differences in the Eclipse Modeling Framework. *UPGRADE*, 9:29–34, 2008.

[14] S. Ceri, T. Palpanas, E. D. Valle, D. Pedreschi, J. Freytag, and R. Trasarti. Towards mega-modeling: a walk through data analysis experiences. *SIGMOD Record*, 42(3):19–27, 2013.

[15] D. Crockford. The JSON Data Interchange Format. Technical report, ECMA International, 2013.

[16] M. Dalla Preda, S. Giallorenzo, I. Lanese, J. Mauro, and M. Gabbrielli. AIOCJ: A Choreographic Framework for Safe Adaptive Distributed Applications. In *Proc. of SLE 2014*, volume 8706 of *LNCS*, pages 161–170. Springer, 2014.

[17] J. Des Rivières and J. Wiegand. Eclipse: A Platform for Integrating Development Tools. *IBM Systems Journal*, 43:371–383, 2004.

[18] Z. Diskin, S. Kokaly, and T. Maibaum. Mapping-Aware Megamodeling: Design Patterns and Laws. In *Proc. of SLE 2013*, volume 8225 of *LNCS*, pages 322–343. Springer, 2013.

[19] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. D. P. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. A. Vergu, E. Visser, K. van der Vlist, G. H. Wachsmuth, and J. van der Woning. The State of the Art in Language Workbenches. In *Proc. of SLE 2013*, volume 8225 of *LNCS*, pages 197–217. Springer, 2013.

[20] J. Favre, R. Lämmel, and A. Varanovich. Modeling the Linguistic Architecture of Software Products. In *Proc. of MODELS 2012*, volume 7590 of *LNCS*, pages 151–167. Springer, 2012.

[21] J.-M. Favre, R. Lämmel, T. Schmorleiz, and A. Varanovich. 101companies: a community project on software technologies and software

languages. In *Objects, Models, Components, Patterns*, pages 58–74. Springer, 2012.

[22] S. Fehrenbach, S. Erdweg, and K. Ostermann. Software evolution to domain-specific languages. In *Proc. of SLE 2013*, volume 8225 of *LNCS*, pages 96–116. Springer, 2013.

[23] J. Fialli, S. Vajjhala, and K. Kawaguchi. The Java architecture for XML binding (JAXB) 2.2. *Specification Final release*, 2009.

[24] M. Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010.

[25] A. Helene Bagge and T. Hasu. A Pretty Good Formatting Pipeline. In *Proc. of SLE 2013*, volume 8225 of *LNCS*, pages 177–196. Springer, 2013.

[26] R. Hilliard, I. Malavolta, H. Muccini, and P. Pelliccione. Realizing architecture frameworks through megamodelling techniques. In *Proc. of ASE 2010*, pages 305–308. ACM, 2010.

[27] F. Jouault, B. Vanhooff, H. Bruneliere, G. Doux, Y. Berbers, and J. Bézivin. Inter-DSL coordination support by combining megamodeling and model weaving. In *Proc. of SAC 2010*, pages 2011–2018. ACM, 2010.

[28] L. C. Kats and E. Visser. The spoofax language workbench: rules for declarative specification of languages and IDEs. In *Proc. of OOPSLA 2010*, pages 444–463. ACM, 2010.

[29] W. Kling, F. Jouault, D. Wagelaar, M. Brambilla, and J. Cabot. MoScript: A DSL for Querying and Manipulating Model Repositories. In *Proc. of SLE 2011*, volume 6940 of *LNCS*, pages 180–200. Springer, 2012.

[30] P. Klint, T. Van Der Storm, and J. Vinju. On the impact of DSL tools on the maintainability of language implementations. In *Proc. of LDTA 2010*, pages 1–9. ACM, 2010.

[31] D. A. Ladd and J. C. Ramming. Two Application Languages in Software Production. In *Proc. of USENIX 1994*. USENIX Association, 1994.

[32] R. Lämmel and A. Varanovich. Interpretation of Linguistic Architecture. In *Proc. of ECMFA 2014*, volume 8569 of *LNCS*, pages 67–82. Springer, 2014.

[33] O. Lassila and R. R. Swick. Resource description framework (RDF) model and syntax specification. Recommendation 22 February 1999 REC-rdf-syntax-19990222, W3C, 1999.

[34] B. Meyers and H. Vangheluwe. A framework for evolution of modelling languages. *Science of Computer Programming*, 76(12):23–46, 2011.

[35] F. Montesi, C. Guidi, R. Lucchi, and G. Zavattaro. JOLIE: a Java Orchestration Language Interpreter Engine. *ENTCS*, 181:19–33, 2007.

[36] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, and G. Succi. A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team. In *Balancing Agility and Formalism in Software Engineering*, volume 5082 of *LNCS*, pages 252–266. Springer, 2008.

[37] OSGi Alliance. OSGi Service Platform, Release 3. 2003.

[38] T. Parr. The Definitive ANTLR 4 Reference. 2013.

[39] D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architecture. *SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[40] F. Rabbi and W. MacCaull. T□: A Domain Specific Language for Rapid Workflow Development. In *Proc. of MODELS 2012*, volume 7590 of *LNCS*, pages 36–52. Springer, 2012.

[41] J. Roy and A. Ramanujan. XML schema language: taking XML to the next level. *IT professional*, 3:37–40, 2001.

[42] A. Seibel, S. Neumann, and H. Giese. Dynamic hierarchical mega models: comprehensive traceability and its efficient maintenance. *Software and System Modeling*, 9(4):493–528, 2010.

[43] A. M. Sloane, M. Roberts, S. Buckley, and S. Muscat. Monto: A Disintegrated Development Environment. In *Proc. of SLE 2014*, volume 8706 of *LNCS*, pages 211–220. Springer, 2014.

[44] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.

[45] A. Van Deursen and P. Klint. Little languages: Little maintenance? *Journal of Software Maintenance: Research and Practice*, 10(2):75–92, 1998.

[46] A. van Deursen, P. Klint, and J. Visser. Domain-Specific Languages: An Annotated Bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

[47] V. Viyovic, M. Maksimovic, and B. Perisic. Sirius: A rapid development of DSM graphical editor. In *Proc. of INES 2014*, pages 233–238. IEEE, 2014.

[48] V. Zaytsev and A. H. Bagge. Parsing in a Broad Sense. In *Proc. of MODELS 2014*, volume 8767 of *LNCS*, pages 50–67. Springer, 2014.