Interpretation of Linguistic Architecture

Ralf Lämmel and Andrei Varanovich

Software Languages Team
University of Koblenz-Landau, Germany
http://softlang.wikidot.com/

Abstract. The megamodeling language *MegaL* is designed to model the linguistic architecture of software systems: the relationships between software artifacts (e.g., files), software languages (e.g., programming languages), and software technologies (e.g., code generators) used in a system. The present paper delivers a form of interpretation for such megamodels: resolution of megamodel elements to resources (e.g., system artifacts) and evaluation of relationships, subject to designated programs (such as pluggable 'tools' for checking). Interpretation reduces concerns about the adequacy and meaning of megamodels, as it helps to apply the megamodels to actual systems. We leverage *Linked Data* principles for surfacing resolved megamodels by linking, for example, artifacts to *GitHub* repositories or concepts to *DBpedia* resources. We provide an executable specification (i.e., semantics) of interpreted megamodels and we discuss an implementation in terms of an object-oriented framework with dynamically loaded plugins.

Keywords: megamodel, interpretation, technological space, software language, software technology, ontology, *Linked Data*.

1 Introduction

The notion of megamodeling has seen much recent interest specifically in the MDE community with diverse application areas such as model management [2], software architecture [12], and models at runtime [18]. Different definitions of 'megamodel' are in use, see, for example, [4] for a more recent proposal. Usually, it is assumed that a megamodel is a model whose model elements are again models by themselves while the term 'model' is interpreted in a broad sense to include metamodels, conformant models, and transformation models.

In our recent work [7], we have introduced a megamodeling approach that it is not tailored to MDE; it is, in fact, meant to be applicable to arbitrary technological spaces [16]. To this end, we have introduced the megamodeling language *MegaL* for modeling the linguistic architecture of software systems, i.e., a system's architecture in terms of relationships between conceptual entities such as languages and technologies as well as actual entities

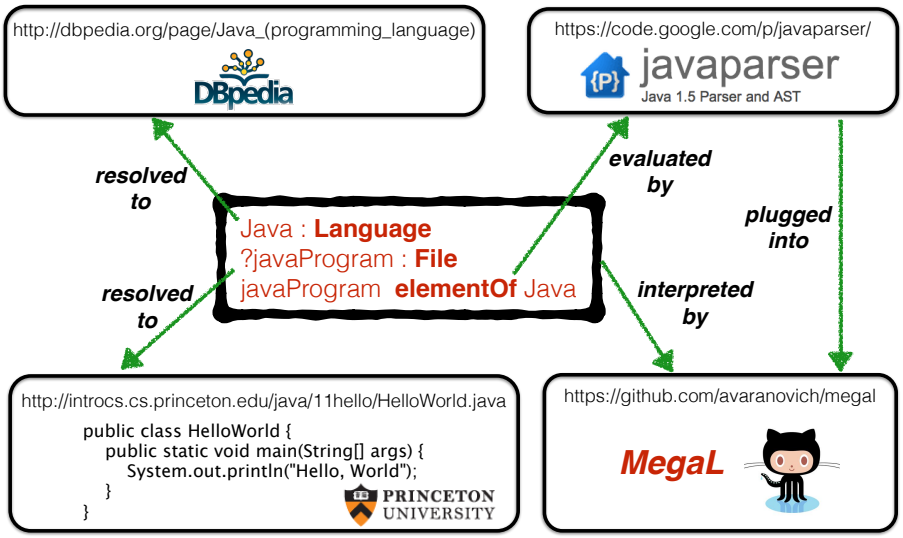

Fig. 1: Interpretation of a megamodel

('artifacts') such as files. Until now, *MegaL* models lacked a proper interpretation which should define how to link megamodel nodes to actual resources (such as system artifacts or documentation) and edges to functionality for checking relationships.

The present paper[1] fills in the notion of interpretation of megamodels. In this manner, we provide a general facility to apply megamodels to actual systems and to validate the claims that are made by megamodels.

Consider Figure 1 for an illustration. The megamodel in the center of the figure declares a language Entity 'Java', a file entity parameter 'javaProgram', and a relationship between these entities such that the latter is an element of the former. Thus, the megamodel essentially describes a trivial Java-based system. The *MegaL* model can be interpreted as indicated in the figure, subject to a configuration and suitable plugins not shown here in detail. The interpretation entails these aspects:

- ◇ The language 'Java' is resolved in terms of the corresponding resource (page) according to the ontology provided by *DBpedia*.
- ◇ The parameter 'javaProgram' is resolved to the on-line version of a 'hello world' program on a web server at the *Princeton University*.
- ◇ The 'elementOf' relationship is evaluated by the Java parser of the *javaparser* project hosted on *Google Code*.

Characteristics of the approach We begin with characteristics of the basic *MegaL* approach, essentially inherited from [7].

[1] The paper's website: `http://softlang.uni-koblenz.de/megal-interpretation/`

- ◇ *Extra models on top of systems*: A megamodel is seen as an abstraction over an existing system, added 'after the fact', as opposed to forming a part of a system or expressing its composition, as in the case of model management.
- ◇ *Flexibility in terms of technological spaces*: Software technologies and systems may involve different technological spaces (such as grammarware or Javaware) without preference for a specific one such as MDE.
- ◇ *Decreased relevance of metamodels*: Metamodels or metamodel-like artifacts (e.g., schemas) are often unavailable or of limited relevance outside clean-room MDE. That is, we often refer to languages instead of metamodels, i.e., to conceptual entities rather than artifacts.

We continue with characteristics of interpretation. These are the **contributions** of the present paper.

- ◇ *Resource-based resolution of entities*: The entities in a megamodel may be resolved to resources that can be addressed with URIs, thereby enabling transparent reuse of existing ontologies (e.g., *DBpedia*) and repositories (e.g., *GitHub* repos). We leverage *Linked Data* principles.
- ◇ *Flexibility in terms of ontologies*: There does not exist a comprehensive ontology for software engineering. Thus, different ontologies, subject to a plugin infrastructure, may be combined to assign meaning to the entity types and the conceptual entities in a megamodel.
- ◇ *Tool-based interpretation of relationships*: Relationships may be interpreted by designated programs ('tools'), e.g., a program implementing the membership test for a given language. This is supported by a plugin infrastructure, without favoring any particular semantics formalism.
- ◇ *Traceability recovery*: The actual semantics of transformation relationships is often unaccessible, as it is buried in software technologies. Thus, it may be preferable to construct a simplified and accessible variant of the actual semantics which provides insight due to its simplicity and through recovered traceability links for the involved artifacts.

Road-map of this paper §2 describes *MegaL* without interpretation; it also develops a relatively simple, illustrative megamodel, which will serve as the running example of the paper. §3 develops the central notion of megamodel interpretation including its implementation as an object-oriented framework. §4 provides an executable specification (semantics) of interpreted megamodels. §5 discusses related work. §6 concludes the paper.

2 Megamodeling with *MegaL*

This section describes the language elements of *MegaL*. We develop a relatively simple, illustrative megamodel, which will serve as the running example of the paper. All original aspects of interpretation are deferred to the next two sections.

2.1 *MegaL* entities

All *entities* in a megamodel must get assigned an *entity type*. These types are also defined in *MegaL*. Entity types are declared as subtypes of the root entity type ***Entity*** or subtypes thereof. In this manner, a classification hierarchy (i.e., a taxonomy or ontology of entity types) is described. Here are some reusable entity types, as declared in actual *MegaL* syntax:

```
Set < Entity // Sets such as languages; see below
Language < Set // Languages as sets, e.g., sets of strings
Technology < Entity // Technologies in the sense of conceptual entities
Artifact < Entity // Artifacts as entities with a physical manifestation
File < Artifact // Files as a common kind of artifact
Function < Set // A function such as the meaning of a program
FunctionApplication < Entity // A particular application of a function
```

Entity types are exercised in entity declarations as those of Figure 1:

```
Java : Language // Entity Java is of type Language
?javaProgram : File // Entity (parameter) javaProgram is of type File
```

We defer the discussion of the exact difference between entities and entity parameters (see the prefix '?') until we deal with resolution in §3.

2.2 *MegaL* relationships

All relationships between entities are instances of appropriate relationship types. Again, these types are defined in *MegaL*. Here are some reusable relationship types, as declared in actual *MegaL* syntax:

```
elementOf < Entity * Set // Membership in the set−theoretic sense
conformsTo < Artifact * Artifact // Conformance in the sense of metamodeling
defines < Artifact * Entity // Such as a grammar defining a language
domainOf < Set * Function // The domain of a function
rangeOf < Set * Function // The range of a function
inputOf < Entity * FunctionApplication // The input of a function application
outputOf < Entity * FunctionApplication // The output of a function application
partOf < Entity * Entity // A physical or conceptual containment relationship
```

Relationship types are exercised in declarations as this one of Figure 1:

```
javaProgram elementOf Java
```

2.3 An illustrative megamodel

Let us capture key aspects of ANTLR usage in a software system. ANTLR[2] is (among other things) a parser generator that targets, for example, Java. Thus, ANTLR can be used to generate Java code for a parser for some language from a grammar given in ANTLR's grammar notation.

[2] `http://www.antlr.org/`

Entities We declare the essential entities of ANTLR usage for parser generation:

```
ANTLR : Technology // The technology as a conceptual entity
Java : Language // The language targeted by the parser generator
ANTLR.Notation : Language // The language of parser specifications
ANTLR.Generator : Function ( ANTLR.Notation → Java )
?aLanguage : Language // Some language being modeled with ANTLR
?aGrammar : File // Some grammar defining the language at hand
?aParser : File // The generated parser for the language at hand
?anInput : File // Some sample input for the parser at hand
```

We leverage a notation for compound entities; see the names *ANTLR.Notation* and *ANTLR.Generator*. That is, ANTLR's notation for grammars is a conceptual constituent of the ANTLR technology as such. ANTLR's generation semantics is also such a constituent. The dot notation implies part-of relationships as follows:

```
ANTLR.Notation partOf ANTLR // Notation is conceptual part of technology
ANTLR.Generator partOf ANTLR // Generator semantics as well
```

We also leverage special notation for function entities; see the declaration of *ANTLR.Generator*. The arrow notation is desugared as follows:

```
ANTLR.Notation domainOf ANTLR.Generator
Java rangeOf ANTLR.Generator
```

Relationships The previously declared entities engage in relationships as follows:

```
aGrammar elementOf ANTLR.Notation // The grammar is given in ANTLR notation
aGrammar defines aLanguage // The grammar defines some language
aParser elementOf Java // Java is used for the generated parser
ANTLR.Generator(aGrammar) ↦ aParser // Generate parser from grammar
anInput elementOf aLanguage // Wanted! An element of the language
anInput conformsTo aGrammar // Conform also to the grammar
```

The declaration of the '↦' relationship is actually a shorthand. We need a designated entity for the function application. Thus, desugaring yields this:

```
ANTLR.GeneratorApp1 : FunctionApplication
ANTLR.GeneratorApp1 elementOf ANTLR.Generator
aGrammar inputOf ANTLR.GeneratorApp1
aParser outputOf ANTLR.GeneratorApp1
```

3 Interpretation of megamodels

Interpretation entails resolution of megamodel entities and evaluation of megamodel relationships. Resolution of entity parameters commences in a 'pointwise' manner in that the parameters are mapped to specific URIs.

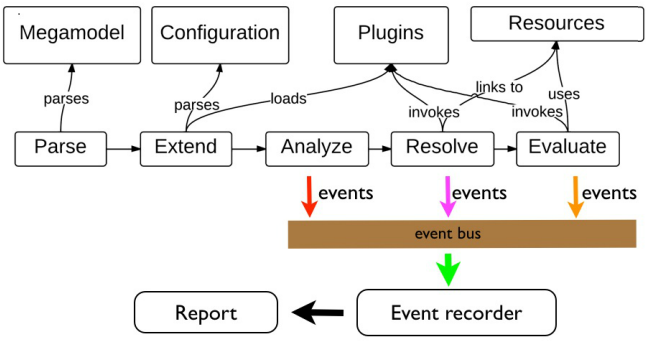

Fig. 2: *MegaL* processing pipeline

Resolution of entities (as opposed to parameters) commences in a schematic manner, subject to 'resolvers' (i.e., programs) for mapping entity names to URIs. Evaluation relies on 'evaluators' (again, programs) for checking the relevant relationships and possibly producing traceability evidence. Pointwise mappings, resolvers, and evaluators are identified in a configuration that goes with a megamodel.

3.1 Megamodel processing

The *MegaL* processor is a Java-based object-oriented framework. Given a megamodel and a configuration, the *MegaL* processor performs the steps summarized in Figure 2.

That is, the megamodel is parsed into an abstract syntax tree based on a suitable object model. In the next step, the configuration file is processed and the corresponding plugins are dynamically loaded and associated with the appropriate AST nodes for entity and relationship types. In the next step, the megamodel and the plugins are analyzed for well-formedness and mutual compliance; see §4 for a precise, formal account. Eventually, resolvers and evaluators are invoked. Resolution determines entity URIs and pings them for availability. Evaluation applies evaluators to the resources (the underlying content) of entities.

Along this pipeline, events are triggered and reported, making the process fully transparent. Any resolution and evaluation problems would also be reported along the way. For instance, the resolution of the 'Java' entity of Figure 1 is reported as follows:

> Looking up entity type *Language*.
< Looked up entity type *Language* successfully.
> Linking entity *Java*.
  ◇ URI located via configuration.
< Linked entity *Java* successfully.

Ideally, all entities of a megamodel should be resolved (successfully) and relationships should be evaluated (successfully). However, this is not always

```
{
  "links" : [ {
      "name": "javaProgram",
      "resource" : "http://introcs.cs.princeton.edu/java/11hello/HelloWorld.java"
    } ],
  "resolvers" : [ { "plugin" : "megal.resolvers.dbpedia" } ],
  "evaluators" : [ {
      "plugin" : "megal.evaluators.FileElementOfLanguage"
      "checkers" : [ { "plugin" : "megal.checkers.languages.Java" } ]
    } ]
}
```

Fig. 3: The configuration for the megamodel in Figure 1

feasible. That is, one may be missing resolvers or evaluators for some of the exercised entities and relationships. In this sense, interpretation may be incomplete, but this would be evident from the event report generated by megamodel processing.

3.2 Configuration of the interpretation

Configuration relies on a simple JSON-based DSL with language elements for URI mapping and registration of mapping resolvers as well as evaluators.

Figure 3 shows the configuration for the introductory Java example. In the ‘links’ section, the parameter ‘javaProgram’ is resolved in a pointwise manner so that it links to the ‘hello world’ program on Princeton University’s web server. In the ‘resolvers’ section, we register a *DBpedia* resolver which is prepared to resolve entity names of the language type to resource URIs on *DBpedia*. In particular, this resolver handles the ‘Java’ entity of the megamodel. In the ‘evaluators’ section, we register an evaluator ‘....FileElementOfLanguage’, which can evaluate ‘elementOf’ relationships when the left operand is a file resource and the right operand is a language. The ‘elementOf’ plugin relies on second-level plugins, ‘checkers’, for individual languages. In the configuration file, we register indeed a checker (i.e., a membership test) for ‘Java’. This checker is a wrapper around the Java parser of the javaparser project. In the *MegaL* project, we aim at collecting all such plugins as consolidated and reusable interpretations of well-defined resources identified through *Linked Data* principles.

3.3 Application to the running example

Let us consider the interpretation of the megamodel for ANTLR, as introduced in §2.3. To begin with, we should pick some software system which exercises ANTLR. Clearly, there is no shortage of such systems. As it happens, the *MegaL* implementation itself also uses ANTLR. Thus, let us apply the *MegaL* model for ANTLR to *MegaL*’s parser.

Entity parameters They are resolved as follows:

aLanguage The language at hand is fixed to be *MegaL*. A link is needed. We choose to link to the language's *GitHub* project.[3]

aGrammar The grammar at hand is the ANTLR-based parser specification of *MegaL*. Thus, we need to link to a specific file *.../MegaL.g4* in said repository.

aParser The parser at hand is a Java source-code file *.../MegaLParser.java* that was generated by ANTLR—again, a file in said repository.

anInput Any *MegaL* source could be linked here. We choose to link to *MegaL*'s prelude with the predefined types, as discuss in §2—again, a file in said repository.

Entities They are resolved as follows:

Java A *DBpedia* resolver is used as explained in §3.2.

ANTLR The *DBpedia* resolver may not be used here because we rely on the fact that *ANTLR* is a compound entity with constituents, as listed below. In the 101companies project [6], software technologies, languages, and concepts are organized in an ontological manner. There is a suitable composition-aware '101companies' resolver for technologies, which links *ANTLR* to a resource.[4]

ANTLR.Notation Use the same resolver as for *ANTLR*.

ANTLR.Generator Use the same resolver as for *ANTLR*.

ANTLR.GeneratorApp1 An application is a pair of the input and output entities. Thus, an application entity is resolved, at a basic level, once input and output are resolved. A more advanced resolution entails the identification of a system artifact's fragment that expresses the application. More specifically, the application of ANTLR's generator could be pinpointed in a build script.

Relationships They are evaluated as follows:

elementOf The evaluator *....FileElementOfLanguage* of §3.2 is enriched by additional second-level plugins (i.e., 'checkers') to serve *aLanguage* (thus, *MegaL*) and *ANTLR.Notation*—in addition to just *Java* previously.

conformsTo Another evaluator *....FileConformsToFile* is needed. It is the language of the right operand which defines the applicable conformance semantics. The result of a conformance test can be richer than just a Boolean value; it may be a set of traceability links between the operands; see §3.4.

defines An evaluator *....Triangle* is used which simply checks that a megamodel with the relationship 'x ***defines*** y' also contains the relationships 'z ***elementOf*** y' and 'z ***conformsTo*** x'. This is Favre's triangle [5].

'$\mapsto$' In fact, we evaluate *ANTLR.GeneratorApp1* ***elementOf*** *ANTLR.Generator* after desugaring. That is, we need to check that *aParser* is the output generated by *ANTLR.Generator* from *aGrammar*. There are several options for checking function applications. As suggested earlier, we may pinpoint the actual application, e.g., in a build script. We could also pinpoint traces of the application, e.g., the Java comment included by ANTLR into the generated source file. We could also apply the function (i.e., run the generator) and compare the result with the existing output artifact. Ultimately, we may analyze input and output and establish problem-specific traceability links based on our understanding of the mapping, thereby also sharing our understanding with others. This is illustrated below.

[3] `https://github.com/avaranovich/megal/`

[4] `http://101companies.org/resources/technologies/ANTLR`

```
// Get methods of interest
val methods = aParser.getMembers()
   .filter(x => x.isInstanceOf[MethodDeclaration])
   .filter(x => ((x.getThrows().map(y => y.getName()).
       contains("RecognitionException"))))
// Get grammar rules
val rules = aGrammar.rules
// Check 1:1 correspondence of names including the same order
val isAlligned = methods.zip(rules).forall(x => x._1.getName().equals(x._2))
```

Fig. 4: Scala-based traceability check for ANTLR's generator

3.4 Traceability recovery

Traceability links may be recovered, for example, for conformance relationships and function application relationships (i.e., 'transformations'). This is illustrated for the application of *ANTLR.Generator*. The input, *aGrammar*, is essentially a list of ANTLR rules with unique nonterminals on the left-hand sides. The output, *aParser*, is essentially a Java file exercising certain code patterns. In particular, for each nonterminal n, there is a corresponding method that implements the rule:

```
public final nContext n() throws RecognitionException { ... }
```

Thus, a suitable approach to traceability recovery is to retrieve nonterminals from the grammar and all relevant methods from the generated Java source and to check for a 1-1 correspondence; see Figure 4 for illustration. For brevity, we show simplified evaluator code that only checks for correspondence, while the actual evaluator collects traceability links (i.e., pairs of URIs) of the following form:

⟨ "http://.../MegaLParser.java/class/MegaLParser/method/megamodel/1" ,
"http://.../MegaL.g4/grammar/megal/rule/megamodel/1" ⟩

The URIs describe the relevant fragments in a language-parametric manner. That is, the URIs start with the actual resource URI for the underlying artifact. The rest of the URI, which is underlined for clarity, describes the access path to the relevant fragment. To this end, syntactical categories of the artifact's language (see 'class' and 'method' versus 'rule') and names of abstractions (see 'megamodel') are used. (We note that 'megamodel' is the first nonterminal, in fact, the startsymbol of the grammar for *MegaL*.)

4 Executable specification of *MegaL*

The following specification of *MegaL* clarifies the meaning of entity resolution and relationship evaluation. The specification assumes an abstract *MegaL* syntax—without convenience notation for functions and function applications and without consideration of compound entities. The specification does also not cover traceability recovery (§3.4).

4.1 Specification style

The specification is a deductive system, as commonplace for type systems and operational semantics. The specification is executable—directly as a logic program in Prolog.[5] *MegaL* is not a regular programming language. Thus, it requires some insight to identify counterparts for what is usually referred to as static versus dynamic semantics.

We assume that interpreted megamodels consist of two parts: the actual megamodel and (the description of) the interpretation—the latter as an abstraction of the configuration, resolvers, and evaluators used in the actual implementation of §3. Given a megamodel *MM* and an interpretation *Interp*, the informal process of Figure 2 is formally described as follows:

```
process(MM, Interp) ⟹
  megamodel(MM), % Inductive syntax definition of megamodels
  okMegamodel(MM), % Well−formedness relation for megamodels
  interp(Interp), % Inductive syntax definition of interpretations
  okInterp(Interp), % (Trivial) well−formedness of interpretations
  correct(MM, Interp), % Correctness of interpretation w.r.t megamodel
  complete(MM, Interp), % Completeness of interpretation w.r.t. megamodel
  evaluate(MM, Interp). % Evaluation of relationships
```

We discuss the contributing judgments in turn.

4.2 Abstract syntax of megamodels

A megamodel is a list of *declarations*. There are declarations for entity-types (***etdecl***s), relationship types (***rtdecl***s), entities (***edecl***s), entity parameters (***pdecl***s), and relationships (***rdecl***s). The declared names are atoms ('ids') and so are all the references to the names. Thus:

```
megamodel(MM) ⟹ map(decl, MM).
decl(etdecl(SubT, SuperT)) ⟹ atom(SubT), atom(SuperT).
decl(rtdecl(R, T1, T2)) ⟹ atom(R), atom(T1), atom(T2).
decl(edecl(E, T)) ⟹ atom(E), atom(T).
decl(pdecl(E, T)) ⟹ atom(E), atom(T).
decl(rdecl(R, E1, E2)) ⟹ atom(R), atom(E1), atom(E2).
```

4.3 Well-formedness of megamodels

Well-formedness is defined as a family of relations, as usual, on the syntactical domains. Well-formedness ensures that all referenced names of entity types, relationship types, and entities (or parameters) are actually declared. (This is part of what we call 'Analyze' in Figure 2.) We omit most of these

[5] The specification is available from the paper's website. Basic logic programming is used, except for higher-order predicates [17] for list processing: *map* (for applying a predicate to the elements of a list), *filter* (for returning the elements that satisfy a predicate), and *zip* (for building a list of pairs from two lists).

routine definitions; a more insightful detail is well-formedness of relationship declarations:

```
okRDecl(MM, rdecl(R, E1, E2)) ⟹
  member(rtdecl(R, Tl1, Tr1), MM), % RType exists
  getEntityType(MM, E1, Tl2), % Type of left entity
  getEntityType(MM, E2, Tr2), % Type of right entity
  subtypeOf(MM, Tl2, Tl1), % Left type Ok
  subtypeOf(MM, Tr2, Tr1). % Right type Ok
```

That is, any declared relationship between two entities E_1 and E_2 must be based on a relationship-type declaration for the same relationship symbol R with entity types Tl_1 and Tr_1 such that the actual entity types Tl_2 and Tr_2 are subtypes of the declared types Tl_1 and Tr_1. Subtyping is defined in terms of the type hierarchy defined by entity-type declarations. This is subtyping like in a single-inheritance OO programming language.

4.4 Abstract syntax of interpretations

We invent a representation of interpretations (say, definitions) of parameters (***pdef***s), entity types (***etdef***s), and relationship types (***rtdef***s). In this manner, we abstract from the plugins of the OO framework and the configuration as discussed in §3. Thus:

```
interp(Interp) ⟹ map(def, Interp).
def(pdef(E, U)) ⟹ atom(E), uri(U).
def(etdef(T, F)) ⟹ atom(T), function(F, [atom], [uri]).
def(rtdef(R, T1, T2, P)) ⟹ atom(R), atom(T1), atom(T2), predicate(P, [uri, uri]).
```

That is, a parameter definition (***pdef***) associates an entity parameter E with a URI U; an entity-type definition (***etdef***) associates an entity type T with a function F mapping entity names to URIs; a relationship-type definition (***rtdef***) associates a relationship type $\langle R, T_1, T_2 \rangle$ with a predicate P on entity URIs. Thus, ***etdef***s and ***rtdef***s model resolvers and evaluators, respectively. We view the aforementioned predicates and functions here as being defined by their extension, i.e., a suitable set of tuples. Thus:

```
predicate(Tuples, Types) ⟹ set(Tuples), map(tuple(Types), Tuples).
function(Tuples, Domain, Range) ⟹ ... % likewise for functions
tuple(Types, Tuple) ⟹ zip(Types, Tuple, TT), map(apply, TT).
```

In the actual implementation, resolvers and evaluators are of course programs that may retrieve resources via the URIs over the internet.

4.5 Correctness and completeness

We present correctness and completeness as two aspects of well-formedness of the megamodel-interpretation couple. (We do not discuss well-formedness of interpretations by themselves, as there are only a few trivial constraints.)

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.

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.

Correctness and completeness are again specified as families of relations. For example, here is the judgment for establishing correctness of relationship-type definitions w.r.t. a megamodel.

```
correctRTDef(MM, rtdef(R, Tl1, Tr1, _)) ⟹
  okT(MM, Tl1), % Left entity type exists
  okT(MM, Tr1), % Right entity type exists
  member(rtdecl(R, Tl2, Tr2), MM), % Relationship type exists
  subtypeOf(MM, Tl1, Tl2), % Definition vs. declaration (left)
  subtypeOf(MM, Tr1, Tr2). % Definition vs. declaration (right)
```

That is, for each relationship-type definition of the interpretation, we can find a corresponding declaration of the megamodel which uses the same or more general entity types.

Let us also consider the counterpart from the family of relations for completeness, i.e., the relation for establishing that a given relationship can be evaluated unambiguously by a definition. This judgement is involved—it is comparable to resolution of names in a non-trivial programming language.

```
% Relationship−type definition unambiguous
completeDecl(MM, Interp, rdecl(R, El, Er)) ⟹
  getRTDef(MM, Interp, R, El, Er, _).

% Determine suitable relationship−type definition
getRTDef(MM, Interp, R, El, Er, RTDef) ⟹
  getEntityType(MM, El, Tl), % Look up left entity type
  getEntityType(MM, Er, Tr), % Look up right entity type
  filter(applicableRTDef(MM, R, Tl, Tr), Interp, RTDefs),
  reduceRTDefs(MM, RTDefs, RTDef).

% Applicability of a relationship−type definition
applicableRTDef(MM, R, Tl1, Tr1, rtdef(R, Tl2, Tr2)) ⟹
  subtypeOf(MM, Tl1, Tl2),
  subtypeOf(MM, Tr1, Tr2).

% Eliminate more general relationship−type definition
reduceRTDefs(_, [RTDef], RTDef). % One rtdef left
reduceRTDefs(MM, RTDefs1, RTDef) ⟹
```

```
member(RTDef1, RTDefs1), % Pick some rtdef
member(RTDef2, RTDefs1), % Pick some rtdef
RTDef1 ≠ RTDef2, % Two different rtdefs
RTDef1 = rtdef(R, Tl1, Tr1, _),
RTDef2 = rtdef(R, Tl2, Tr2, _),
subtypeOf(MM, Tl1, Tl2),
subtypeOf(MM, Tr1, Tr2),
delete(RTDefs1, RTDef2, RTDefs2), % Remove the more general rtdef
reduceRTDefs(MM, RTDefs2, RTDef).
```

This approach is similar to instance resolution in Haskell [11], the one for multi-parameter type classes with overlapping instances specifically [19]. That is, definitions (‘instances’ in Haskell terms) are not proactively rejected by themselves—just because they are overlapping in some sense. Instead, any given relationship is considered as to whether it can be associated uniquely with a definition that is more specific than all other applicable definitions.

4.6 Evaluation of relationships

Evaluation is straightforward at this stage, as all preconditions have been established. That is, entities or parameters thereof can be replaced by URIs and relationships can be evaluated on the URIs for the arguments. Thus:

```
evaluateDecl(MM, Config, rdecl(R, El, Er)) ⟹
  getRTDef(MM, Config, R, El, Er, rtdef(_, _, _, P)),
  getEUri(MM, Config, El, Ul),
  getEUri(MM, Config, Er, Ur),
  applyPredicate(P, [Ul, Ur]).

% Get URI for entity via definition
getEUri(MM, Config, E, U) ⟹
  getEntityType(MM, E, T), % Look up entity type
  member(etdef(T, F), Config), % Look up definition
  applyFunction(F, [E], [U]). % 'Resolve'

% Application of extension−based predicates and functions
applyPredicate(Tuples, X) ⟹ member(X, Tuples).
applyFunction(Tuples, Arg, Res) ⟹ append(Arg, Res, X), member(X, Tuples).
```

Soundness (i.e., alignment between ‘type system’ and ‘semantics’) follows trivially in this approach—as the completeness judgment immediately ensures that all instances of entity resolution and relationship evaluation can be attempted. Thus, the only remaining option for *evaluateDecl* to fail is that a resolution was not successful or a specific relationship failed.

5 Related work

We compare *MegaL* with several approaches to megamodeling. The Atlas MegaModel Management approach (AM3) conveys the idea of modeling in the large, establishing and using general relationships, such as conformance, and metadata on basic macroscopic entities (mainly models and metamodels) [2]. Based on the assumption that all managed artifacts are models conforming to precise metamodels, a solution for typing megamodeling artifacts is proposed in [20]. Model typing is based on the conformance relationship; metamodels are used as types. *MegaL* is clearly not restricted to modeling resources and does not require an existence of metamodels. Also, *MegaL*'s approach to megamodel interpretation provides an open, heterogenous type system.

A formal, graph-oriented view on megamodels is considered in [4]; entities are vertices and relations are edges between them. It is argued, that the semantics of relations are hidden in the type name and are not presented in the megamodel. To fill this gap, the authors zoom into nodes and edges and disassemble them into more elementary building blocks. In the case of *MegaL*, such a formal analysis of the relationships is less relevant, as it is not directly applicable to actual software projects and technologies. Instead, as shown in §3.4, we leverage tool-based relationship evaluators with optional traceability recovery. *MegaL* is also influenced by existing megamodeling patterns and idioms, discovered in theoretical work [8,5,4].

In a comprehensive survey [21] of traceability in MDE, the authors conclude, that traceability practices are still emerging, specifically in the MDE context. *MegaL*'s interpreted megamodels may associate entities in relationships with traceability links, as it was shown in §3.4. This approach is again heterogenous in terms of the technological spaces; it assumes a language-parametric approach to fragment location. Traceability is also used in megamodeling for models at runtime [18], where high-level relationships between models are derived from observable low-level traceability between model elements.

A type system and a type inference algorithm for declarative languages with constraints for MDE are presented in [13]. Elsewhere [1], OCL [10] constraints and ATL rules [14] are used to implement consistency and conformance checking.

Megamodels of metamodels and model transformations are organized into an architectural framework [9], which promotes re-usability of architectural elements and realizes architectural descriptions [12]. We plan to re-implement such descriptions in *MegaL*, thereby providing evidence of its usefulness as an architecture description language.

MegaL relies on the resources to be exposed via HTTP and uniquely identifiable. Such resources can be directly exposed via web servers and web-accessible source control systems. Another promising direction is to

apply *Linked Data* [3] principles, which allows attaching rich metadata. *MegaL* already applies such principles, e.g., in the sense of the *DBpedia* and 101companies resolvers. *Linked Data* principles are also leveraged in [15] in a related manner for the purpose of exposing facts about artifacts in software repositories.

6 Conclusion

We have equipped the megamodeling notion for the linguistic architecture of software systems with a language mechanism for resolving entities, capturing traceability between them, and evaluating relationships. Our approach is not tailored to MDE. We applied the approach to a megamodeling scenario that indeed involves elements of Javaware and grammarware. We formalized the key ideas of interpreted *MegaL* models in a deductive system and described an open-source implementation. Without this enhancement, megamodeling does not provide enough validated insight into actual systems.

The types of megamodeling relationships with the underlying entity types represent patterns of the linguistic architecture of software systems. *MegaL* has already been applied to some typical scenarios of technology usage, as they are demonstrated by software systems in the 101companies chrestomathy [6], thereby capturing important entity and relationship types. It remains to develop a comprehensive megamodeling ontology in a systematic and transparent manner.

Additional topics for future work include these: i) raise the level of abstraction for traceability recovery by establishing a language-independent DSL layer standardizing fact extraction and link composition; ii) support of the evolution of entities and linked resources by including timestamp and version information; iii) search-based instantiation of megamodels for a given software system; iv) ‘megamodeling in the large’ support in the sense of refinement and composition expressiveness; v) ‘megamodeling as a service’ to simplify the setup of the interpreter with its diverse plugins providing support for different technological spaces and relying on different platforms.

References

1. J. Bézivin and F. Jouault. Using ATL for Checking Models. *ENTCS*, 152:69–81, 2006.
2. J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez. Modeling in the Large and Modeling in the Small. In *European MDA Workshops MDAFA 2003 and MDAFA 2004, Revised Selected Papers*, volume 3599 of *LNCS*, pages 33–46. Springer, 2005.
3. C. Bizer, T. Heath, and T. Berners-Lee. Linked data - the story so far. *Int. J. Semantic Web Inf. Syst.*, 5(3):1–22, 2009.

4. 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.
5. J.-M. Favre. Foundations of meta-pyramids: Languages vs. metamodels – Episode II: Story of thotus the baboon. In *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings, 2005.
6. J.-M. Favre, R. Lämmel, T. Schmorleiz, and A. Varanovich. 101companies: A Community Project on Software Technologies and Software Languages. In *Proc. of TOOLS 2012*, volume 7304 of *LNCS*, pages 58–74. Springer, 2012.
7. J.-M. 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.
8. J.-M. Favre and T. NGuyen. Towards a Megamodel to Model Software Evolution through Transformations. *ENTCS*, 127(3), 2004. Proc. of the SETra Workshop.
9. L. Favre and L. Martinez. Formalizing mda components. In *Proc. of ICSR 2006*, pages 326–339, 2006.
10. O. M. Group. *Object Constraint Language Object Constraint Language, OMG Available Specification, Version 2.0*, 2006.
11. C. V. Hall, K. Hammond, S. L. P. Jones, and P. Wadler. Type Classes in Haskell. *TOPLAS*, 18(2):109–138, 1996.
12. R. Hilliard, I. Malavolta, H. Muccini, and P. Pelliccione. Realizing Architecture Frameworks Through Megamodelling Techniques. In *Proc. of ASE'10*, pages 305–308. ACM, 2010.
13. E. K. Jackson, W. Schulte, and N. Bjørner. Detecting Specification Errors in Declarative Languages with Constraints. In *Proc. of MODELS 2012*, pages 399–414, 2012.
14. F. Jouault and I. Kurtev. Transforming Models with ATL. In *Proc. of MODELS 2005, Satellite Events*, volume 3844 of *LNCS*, pages 128–138. Springer, 2006.
15. I. Keivanloo, C. Forbes, A. Hmood, M. Erfani, C. Neal, G. Peristerakis, and J. Rilling. A linked data platform for mining software repositories. In *Proc. of MSR 2012*, pages 32–35. IEEE, 2012.
16. I. Kurtev, J. Bézivin, and M. Akşit. Technological Spaces: an Initial Appraisal. In *Proc. of CoopIS, DOA 2002, Industrial track*, 2002.
17. L. Naish and L. Sterling. Stepwise enhancement and higher-order programming in prolog. *Journal of Functional and Logic Programming*, 2000(4), 2000.
18. A. Seibel, S. Neumann, and H. Giese. Dynamic hierarchical mega models: comprehensive traceability and its efficient maintenance. *Software & Systems Modeling*, 9(4):493–528, 2010.
19. M. Sulzmann, T. Schrijvers, and P. J. Stuckey. Principal Type Inference for GHC-Style Multi-parameter Type Classes. In *Proc. of APLAS 2006*, volume 4279 of *LNCS*, pages 26–43. Springer, 2006.
20. A. Vignaga, F. Jouault, M. Bastarrica, and H. Brunelière. Typing Artifacts in Megamodeling. *Software and Systems Modeling*, pages 1–15, 2011.
21. S. Winkler and J. von Pilgrim. A survey of traceability in requirements engineering and model-driven development. *Software and System Modeling*, 9(4):529–565, 2010.