Type Systems for Domain-specific Languages

Reiner Jung[1] Christian Schneider[2]
Wilhelm Hasselbring[1]
[1] Software Engineering Research Group, Kiel University
[2] Embedded and Realtime Systems Research Group, Kiel University
{rju,chsch,wha}@informatik.uni-kiel.de

Abstract: Model-driven software development employs models to describe different aspects of a system on different levels of abstraction. These aspects are driven by technology or application domain. Modeling is often done in specific graphical or textual notations, called domain-specific languages (DSL). In recent years such languages became very popular in the modeling community to describe structure and sometimes behavior. In the context of type systems, these structures are called types and the behavior is modeled with expressions. The programming language community has developed many concepts to model and specify type systems and the semantics of expressions. However, in the modeling community this is often neglected when specifying meta models and describing their semantics, what may cause problems in developing checks and generators for DSLs. To address this issue, we present an approach, that provides guidance and support during DSL development based on established knowledge on type systems and generator construction, to ease the integration of type systems in DSL. We evaluate this approach with the Xtext language engineering framework.

1 Introduction

Model-driven Software Development (MDSD) uses models to describe different aspects of a system on different levels of abstraction. Some of these aspects are driven by technology, others are desired to obtain mental links to application domains. To compose those models, domain-specific notations are used. These notations can be textual languages, which are also used to serialize and persist models, and graphical notations. Both are largely summarized under the term domain-specific languages (DSL).

DSLs had much an impact on software development in recent years, as models became more important and, mainly, because the tools and frameworks to develop these languages improved significantly. From DSL developers' points of view, a DSL is just a language which allows us to formulate models. The developers focus on the meta model as a set of classes and references, but do seldom apply methods and concepts common in programming language design. However, the developers do employ type systems in their language, even though they are not aware or not caring about that. Often, doing so is even then not considered to be worth the effort when running into serious problems while realizing reference resolution, semantic checks, and transformations.

Xbase [EEK+12] addresses such type system issues, by incorporating the Java type system into DSLs and provide partial grammars to use Java types and expressions based on Java types. This limits the DSL to be merely a front end for framework and library APIs, but allows fast development of DSLs including expressions. DSLs, which use other type systems or combine multiple type systems, and DSLs not translated into Java are not supported by Xbase.

Driven by own experience in recent projects [GHH$^+$12, Jun12] and the limitations found in Xbase, we developed an approach to guide development of type systems for DSLs. In this paper, we present this approach and show how it can be implemented with the Xtext DSL framework and tooling [AF11].

Our approach alleviates language maintenance by proposing a partitioning of aspects, artifacts and activities. This partitioning allows to integrate DSL development in agile environments, where feature requests and change of requirements occur very often to suit the usual growing needs.

The remainder of this paper is structured as follows. Section 3 discusses the similarities and differences between meta models and type systems. Section 4 introduces and motivates characteristics of DSL. In Section 5 code generation strategies for DSLs are explained. Section 6 introduces our TS4DSL approach, whose evaluation in two case studies is documented in Section 7. Section 2 discusses related work. And finally, Section 8 summarizes the paper and provides an outlook on our next targets.

2 Related Work

Xbase [EEK$^+$12] is a framework for Xtext [AF11] to leverage the integration of the expressions and the Java type system. It provides, a Java type system meta model and a corresponding expression meta model, following the same distinction in models, as TS4DSL. To ease grammar construction, it provides a comprehensive expression grammar, which can be embedded and customized in Xtext DSLs. However, possible types of an Xbase-based DSL must comply with the Java type system. Thus Xbase addresses developers who use DSLs as front ends for Java APIs and helps them to integrate their DSLs in Java projects. TS4DSL generalizes the Xbase approach and proposes a way on how to develop DSLs with custom or domain-specific type systems.

XTS Völter proposes a powerful framework for realizing type checks. The framework provides a declarative API for contributing necessary information on the DSL's meta model classes and the intended type system semantics. XTS [Vö11] supports primitive types, like **boolean**, **int**, **float**), enumeration types, arrays, and sub-typing relations.

Xsemantics is a DSL to describe the semantics, especially the type system, of Xtext-based languages. The DSL targets developers who want to address typing issues from a formal perspective based on type theory. Xsemantics[1] allows to define judgments and de-

[1]Xsemantics project: `http://xsemantics.sourceforge.net/`

duction rules close to notations common in type systems community (compare [CDJ+97]).

Xsemantics and XTS address both the declaration of type systems for Xtext-based languages [BSVC12]. For simple common typing cases, XTS provides a more compact notation then Xsemantics. However, functionality not handled by the XTS API must be implemented in Java or Xtend, losing the advantage of a declarative specification language. Xsemantics is more flexible and allows to formulate arbitrary type systems. In addition to typing, Xsemantics generates validation checks for Xtext with, so called, check rules.

Both type system approaches can be used with our method of designing a well-typed DSL. Since we focus on the development process and building blocks of a well typed DSL, we do not advocate for a particular type system implementation. In this paper we use Xtend to describe simple typing rules. However, for more complex type systems and better re-use, formal notations should be considered.

3 Meta Models and Type Systems

Meta modeling languages, such as MOF [Obj06] or UML, already discuss typing. However, in many other modeling approaches the type system aspect of meta models is not in the focus of developers.

Meta models are used to describe the structure of software, data and persistence, and in some cases, behavior. They consist of classes with properties, either carrying attributes of base types (also called primitive types) or references to other classes. References are used to express containment and/or to relate to other data objects.

There are a couple of meta modeling frameworks, i.e. tools to create meta models and their implementation in code. The most common is the Eclipse Modeling Framework (EMF) [SBPM09], providing the Ecore meta modeling language, which is an EMOF [Obj06] implementation for Java and the basis for many meta modeling tools. Upon the EMF core many tools have been built, e.g., model-to-model transformation languages like QVT [OMG05], ATL [BDJ+03] and Xtend [AG11], DSL and meta model generators like Xtext [AF11] and EMFtext [HJK+09], storage systems like CDO [Ste12], and many more.

In addition to EMF there are, e.g., the ATLAS Model Management Architecture (AMMA) project group [JBC+06], which is realized on the Eclipse platform, too, and the tools GME [LMB+01] and VMTS [MLC06], which are built on Windows.

Type systems have been used to formalize and reason about the structure and semantics of types. They are used to check the soundness [Pie02, p. 95ff] of type declaration and expressions (also called *terms*).

The basis of all type systems is a set of *base types* [Pie02, p. 117], such as string, integer, float or void. While these types may share portions of their value sets, for the type system they are mutually distinct. In addition to these base types, enumerations and references are also common to many programming languages. Build upon these basic typing elements, *complex types*, such as record, list, array, and class, can be composed [Pie02, p. 129ff].

Besides these structural elements, type systems come with semantic specifications for languages. There exist different formalisms to specify semantics. We selected an operational semantics notation (compare [CDJ$^+$97, Pie02]) as a basis for our semantic specifications. Although they have some weaknesses, e.g., cannot address temporal issues, they are easy to understand by non-practiced developers.

Type systems and meta models, both formalize structure of data and provide mechanisms to describe restrictions on the structure. However, their perspectives are quite different. MOF-based meta models by itself do not provide much restriction mechanisms beside limiting the number of values in an array or limit the type of references. To apply further restrictions, they require a constraint language, such as OCL [OMG06]. OCL is a very powerful functional language, which provides a rich set of functionality, but it is not specifically designed to compute typing constraints.

Type system notations formulate the semantics in form of axioms and rules comprising a set of premises and a conclusion. These rules are closely related to the abstract syntax of a language. They can clearly be separated into rules to ensure type safety for structures and those used to provide the same for expressions. In meta models, it is often hard to separate data structures and behavior, as both are modeled with classes and properties, referencing each other.

4 Domain-specific Languages

Domain-specific languages (DSL) are languages tailored to a specific technical or application domain. Their main goal is to express knowledge (structure and behavior) of a domain in a textual or graphical notation suitable for the domain with less code or at least better comprehensible code than in general purpose languages [MHS05].

DSLs can be distinguished in two main groups. DSLs realized in a host language, which are often implemented as a library providing a coherent set of functions for a domain, are called internal DSLs, while DSLs with an own grammar or other means of notation, are considered external DSLs. DSLs can be embedded in other languages, MPS [VP12] for example focuses on that type of DSL, or are fully self contained languages, like Xtend [AG11].

Beside the specification of structure and behavior, DSLs are largely used to build executable software. This can be realized by interpretation or by transforming DSL code into an executable language code.

In our approach, we focus on human-readable textual, external, self contained DSLs, which are used to generate code for an executable target language. Such textual languages profit from the fact that common version control systems are able to handle differences and merges much better for text than for graphical or other representations. In addition, development environments and generative tools, like Xtext [AF11], provide assistance for automated editor generation and specialized APIs for syntactical and semantic checks, code highlighting and content assists, which allows us to develop the tooling for a DSL quickly and help DSL-users to formulate their specifications.

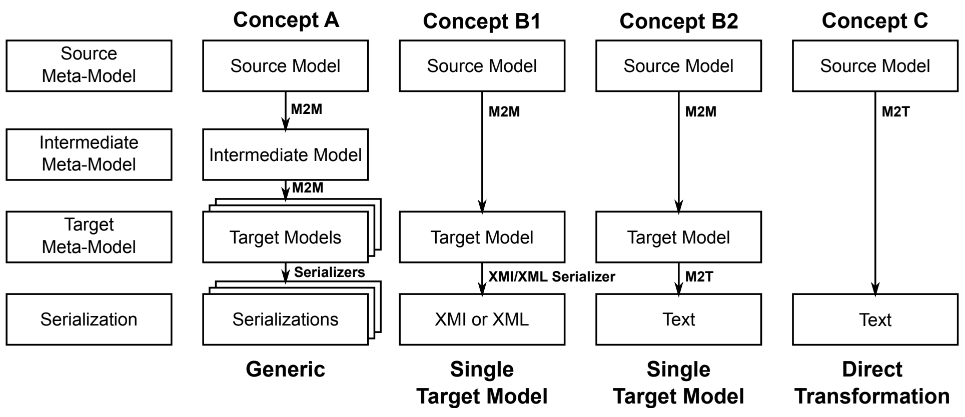

Figure 1: Different generator stack concepts

5 Code Generation Strategies

Generators and compilers scan and parse source code and build an abstract syntax tree (AST) representing the code which is stored in a memory data structure. The AST is then transformed by resolving references (in Xtext this is called linking). The resulting abstract syntax graph may adhere to a meta model for the corresponding language. Or the AST is automatically transformed and stored in the source model. Xtext, the technology we use in this paper, employs an approach where the parse tree is directly transformed into a source model without references resolved, conforming to a AST and in a second step, the references are resolved. Since the concrete steps for obtaining the source model are not in the scope of this work, Figure 1 starts with this source model and omits all previous compiler phases.

The relation between a DSL's meta model (source meta model) and the structure of the generated output can be manifold. Generic code synthesis concepts can be described by four distinct levels (see Figure 1, leftmost column). First, the source meta model, used as storage for artifacts described in a DSL. Second, an intermediate meta model, which resembles a common basis for all target meta models and provides the same type of structural elements to compose types as the target meta models. Third, the target meta models, which are used to represent the generated model in memory. And fourth, the serialization component, which stores target models in files, databases or other means of persistence. Between the three meta models, transformations are used to project the higher level model onto the lower level model.

This expansive structure, well-known in classic compiler construction [ASU88, p. 18], is not always needed. For instance, concepts employing only one target meta model could omit the intermediate meta model. In contexts allowing straight-forward mapping of the source meta model structure to single target code fragments, the target code could be directly generated by a model-to-text transformation. Figure 1 shows four typical DSL-generator model-stacks.

To guide the selection of the different stacks, we established a set of criteria. First, if the target language is a human-readable textual language such as C or Java, not XML or XMI, then the last step can be done by a model-to-text (M2T) transformation. For XML serialization, a meta model for the XML schema definition (XSD) should be generated and used as target meta model. The serialization itself is then done by the XML serializer provided by EMF. For XMI output, the model is serialized with the XMI writer provided by EMF.

Projects using M2T transformations to generate XML encountered serious problems, as M2T transformations write out code in a sequence. While XML supports ids and references to ids (IDREF), references could appear before the id is computed. This requires to implement complex id lookup tables and two pass mechanisms to solve the issue. Using in memory XML models solves the same issue in a more elegant way, as these models could be constructed by model-to-model transformations and then be serialized with XML serializer.

In case the source meta model uses the type system of target meta model, as Xbase [EEK^{+}12] based languages do by employing the Java type system, the source meta model can directly be rendered into a text document.

In case of target meta models, that provide typing structures different from the source meta model, like languages for programmable logic controllers (PLC) [IEC03], or when referencing multiple source meta models, as used in the measure definition language (MDL) of the MAMBA-project [FvHJ^{+}11], more complex composition operations are required. Also, a DSL using sub-typing and dynamic memory allocation with garbage collection, or a target language that supporting just records, then a distinct target meta model is helpful to manage the complexity of the target code generation. This approach is also useful when serializing to XMI or XML, as these serializations make use of cross-references that are hard to keep track of when a source model is directly transformed to text.

6 TS4DSL Approach

TS4DSL is designed to support average DSL developers who use the Xtext framework and tooling to create their languages. These developers often work in an agile environment, provoking changes to meta model and language features in short time. Changes in the meta model and grammar often require changes in typing and validation rules. Such changes to a Java or Xtend-based type and validation system can be complex and time consuming. Furthermore, they cannot be checked for correctness, like formal notations based deduction and reduction rules used to describe operational semantics.

Technologies like, XTS [Vö11] and even more Xsemantics [BSVC12], address the type checking and validation aspects of type system development. While TS4DSL addresses primarily grammar rule design, meta model construction, and generator composition. The approach can be divided in six separate parts. Meta model composition for type systems, integration of base types into the scoping of the DSL, set of Xtext-grammar patterns, type checking, target language integration, and model transformations.

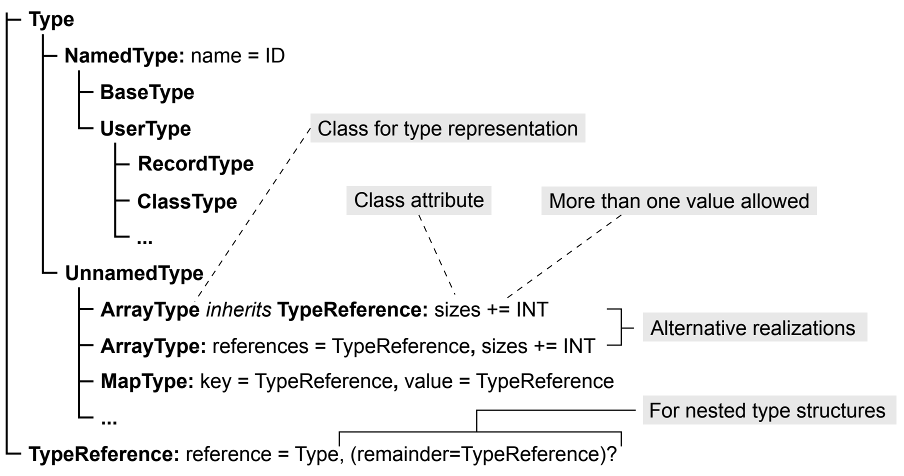
Type
NamedType: name = ID
BaseType
UserType
RecordType
ClassType
...
UnnamedType
ArrayType inherits TypeReference: sizes += INT
ArrayType: references = TypeReference, sizes += INT
MapType: key = TypeReference, value = TypeReference
...
TypeReference: reference = Type, (remainder=TypeReference)?
Class for type representation
Class attribute
More than one value allowed
Alternative realizations
For nested type structures

Figure 2: Taxonomy of Types in a Type System

6.1 Meta Modeling

The meta model of a well-typed DSL can be divided into an expression and type part. For most meta models, it is not necessary to place them into different files or packages, but it is helpful to make this distinction while designing the meta model and the grammar. The awareness of these two aspects in language and meta model design helps later to understand language modifications and the transformation development.

In our approach, we use a class Type to model all types. Based on that, a taxonomy of sub-classes for the different type structures can be build. The taxonomy shown in Figure 2, is our general pattern for the composition of type systems. However, in concrete DSLs certain aspects can be solved differently. For example, the distinction of named and unnamed types, can be either modeled by sub-typing or with multiple inheritance.

In order to represent the base types, the class BaseType is used. All types constructible by modelers are subsumed as UserTypes. This distinction will be helpful later on, e.g. while realizing the type resolution and the identifier (cross reference) resolution.

Our approach uses a ’name’ field for all NamedTypes including the base types. Alternatively each base type could be represented by a separate type class (compare [Vö11]). The advantage of the former method is, type resolution can handle base types like any other user defined type. In addition grammar rules for type references can use one unique rule to describe all kinds of references to named types.

The type reference, in our approach, is modeled with the class TypeReference. It is employed each time a NamedType is referenced, e.g. in declarations of properties, functions and in the composition of UnnamedTypes. UnnamedTypes are types that cannot be identified by a name, instead they are created in place. In many languages arrays are unnamed types, they are composed of a type reference and size constraints, one for each dimension, or at least the number of dimensions.

6.2 Base Type Integration

Base types are the fundamental elements of a type system. The must be available in a language without definition by the language user. Otherwise, types acting as base types cannot be mapped straight-forward to corresponding typing constructs in the target model. We suggest to represent base types by one dedicated class BaseType in the meta model extending the general NamedType class.

To introduce the concrete base types into the language the base type literals should not be hard-wired in the concrete syntax. While this can be done by exploiting Xtext features, it results in mixing different aspects of the language design in one artifact. Instead, we suggest to define base types in a library that is visible by default. In the EMF and Xtext environment, this is best done by providing a dedicated resource containing the necessary instances of BaseType, one for each concrete base type. This resource should not be persisted as a file or another user accessible resource, as it is an integral part of source model semantics. Therefore, we constructed a virtual resource that is implicitly made accessible to the language tooling[2]. Our approach is similar to the type integration for Java types provided by the Xbase framework. However, it is simpler, as we only have to support base types.

The collection of concrete base types can be determined by means of an enumeration, as we have done in our use cases, or in a String array (see Listing 1). The enumeration approach benefits from the natural mutual disjointness of the literals, while a string array would allow to add the same name twice. Therefore, we recommend an enumerated solution, even though it is a bit more complex than a simple string array.

```
public enum BaseTypes {
    BOOLEAN, INT, FLOAT, STRING; /** base types */

    public String lowerCaseName() {
        return this.name().toLowerCase();
    }
}
```

Listing 1: Base Types of the Use Case *Language for App Development*

The class structure of the base type integration, which has been adopted in part from Xbase' type integration, is shown in Figure 3. The key components of the infrastructure are the previously described BaseTypes enumeration, the TypeResource that is invisibly instantiated while loading a model, the TypeProvider that instantiates the resource and acts as a facade of that resource. The TypeGlobalScopeProvider, employed for determining sets of candidates for resolving cross-references, returns so-called *scopes* that are based on the BaseTypeScope, which contributes the known instances of BaseType. These classes are in charge of making the base types visible in concrete Xtext-based language toolings (specific EMF-based resource factories, textual editors).

[2]An example application can be found in the following git repository: `http://build.se.informatik.uni-kiel.de/de.cau.cs.se.lad.git`

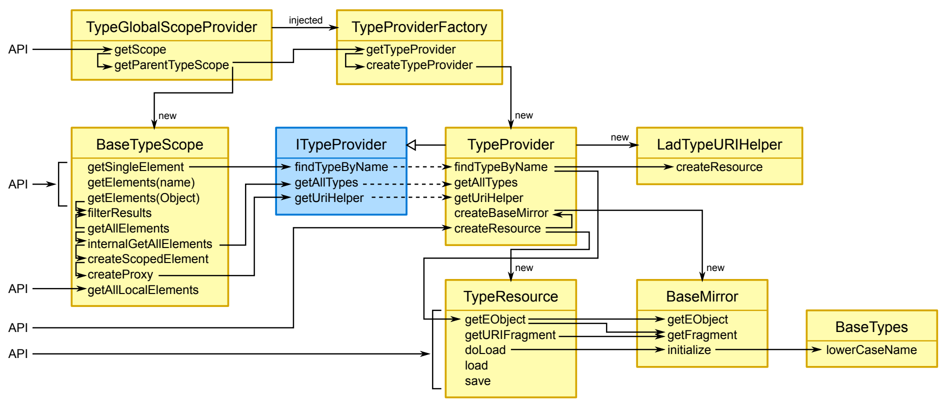

Figure 3: Type System Overview

6.3 Grammar Pattern

Typical rules to construct named and unnamed user types are presented in Listing 2. The first two rules represent the effective taxonomy of the different types. The left most word, is the rule name followed by the corresponding meta model class. If the meta model is automatically generated out of the grammar, the **returns** followed by the meta model class can be omitted. However, for type systems with many different kinds of types, this could lead to meta models hard to comprehend.

```
Type returns type::Type: BaseType | UserType | ArrayType ;
UserType returns type::UserType: ClassType | RecordType ;

TypeReference returns type::TypeReference:
{type::ArrayType} reference = [types::NamedType| ID] ( '[' sizes += INT ']' )+ |
{type::TypeReference} reference = [types::NamedType| ID] ('.' remainder=TypeReference )? ;

MapType: {type::MapType} 'map' '<' key = TypeReference ',' value = TypeReference '>' ;
```

Listing 2: Xtext grammar rules for type representation

After specifying the meta model type, the rule body is defined. The Xtext grammar follows here loosely the EBNF with some extensions. First, names followed by a = are properties of the meta model class associated with the rule. Second, expressions in square brackets are used to express references. For example, in TypeReference the expected token is an ID which is limited to names of type::NamedType of the meta model. Third, meta model classes in curly brackets are used to ensure the creation of an instance of the given type.

```
PropertyDeclaration returns type::PropertyDeclaration:
    modifiers += Modifier type = TypeReference name = ID ;

FunctionDeclaration returns type::FunctionDeclaration:
    modifiers += Modifier type = TypeReference name = ID
    '(' (parameters += ParameterDeclaration (',' parameters += ParameterDeclaration)∗)? ')'
    body = Body ;
```

Listing 3: Xtext grammar rules for properties and functions

```
/** axiom case 1: determines the type of 'true' and 'false', similar for IntValue, Enumeration, etc. */
def dispatch Type getActualType(BooleanValue value) {
    return typeProvider.findTypeByName("boolean");
}
/** recursive case 1/axiom case 2: determines the referenced type of a TypeReference by determining the type
  * of the last component of the (potentially compound) declared type, e.g. A.B[5] */
def dispatch Type getActualType(TypeReference ref) {
    return ref.remainder?.actualType?:ref.reference;
}
/** recursive case 2: determines the type of a value, e.g. 'boolean x = 5;', by determining the declared type */
def dispatch Type getActualType(ValueDeclaration decl) {
    return decl?.typeReference?.actualType;
}
/** recursive case 3: determines the type of an identifier by determining the type of the identified element,
  * e.g. of the declared value 'x' */
def dispatch Type getActualType(ValueReference ref) {
    return ref?.reference?.actualType;
}
/** recursive case 4: determines the type of a declared function by determining its referenced return type */
def dispatch Type getActualType(FunctionDeclaration decl) {
    return decl.type.actualType;
}
/** recursive case 5: determines the type of a function result by determining the called function's return type */
def dispatch Type getActualType(FunctionCall call) {
    return call.functionRef.actualType;
}
/** recursive case 6: determines the type of an assignment by determining the modified value's type */
def dispatch Type getActualType(Assignment assignment) {
    return assignment.target.actualType;
}
```

Listing 4: Type resolution realization by means of Xtend's dispatch extensions (excerpt)

Listing 3 shows general patterns for property declarations as well as the definition of functions, methods, procedures or any other parametrized structure. In many languages, those elements can have modifiers such as **public** or **static**. The subsequently required part TypeReference specifies the return type of definition, the **name** attribute its identifier. The non-shown rule ParameterDeclaration is similar to that of PropertyDeclaration, it merely employs different modifiers. Furthermore, properties are often initialized in its declaration. To enable that the rule PropertyDeclaration must be extended by an optional call of the rules Expression or Literal whose result is assigned to a field called expression or value.

6.4 Implementing Type Resolution

Occurrences of types in declarations are represented by a TypeReference maintaining a non-containment reference to the actual type. Applying this delegation pattern is reasonable, as the reference resolution is limited to instances of TypeReference instead of providing one for each of the available kinds of declaration. In our Xtext-based setting this resolution is realized in terms of a *scope provider* that is supported by the former mentioned TypeGlobalScopeProvider. The implementation follows the usual Xtext scope provider declaration scheme [AF11].

In order to establish type checking of expressions of a DSL, like type compatibility of left and right hand side of assignments or operands of binary operations, we employed the Xtend language. Its features like *null-safe feature call* and *method dispatch* proved very comfortable and allow the compact implementation outlined in Listing 4. Note that ValueDeclaration is a super type of PropertyDeclaration.

Xtend allows a very compact formulation, e.g. **null** checks are expressed by the question marks. Second, Xtend allows suffix notation in case the first parameter of the method, here called *extension*, is type compatible. Using this feature, method name prefixes like 'get' can be skipped, see calls of actualType alias getActualType. Third, the keyword **dispatch** instructs Xtend's code generator to create additional type inference code w.r.t. the extension parameter types. Hence, when calling getActualType(...) for some expression, assignment, or value declaration the extension requiring the most special fitting parameter type is chosen. Thus, by means of such partial inference rules that are dedicated to particular declaration and expression types and that delegate to each other, the type system semantics can be realized very precise, easy to understand and easy to extend. As mentioned in Sec. 2 a dedicated type system framework can by applied alternatively.

6.5 Target Language Integration

The previous sections focus on development of the DSL, the source meta model and the code relevant for editors and source model composition. This section addresses the integration of the target language and target meta model. The simplest method to generate code in a target language is a model-to-text transformation, which is covered by the Xbase framework and well documented and explained in related tutorials [EEK$^+$12].

In this paper, we discuss the more complex target code genesis through transformations into intermediate or target models. The differentiation between an intermediate or target meta model, as shown in Figure 1, is only of a contextual nature, because an intermediate model is subsequently transformed into another model, while the target model, is serialized into text or other kind persistence with a model-to-text or respectively model-to-persistence transformation. These last transformation from a model to a persistence technology is covered by Xtend [AG11] as a model-to-text transformation language and serialization components provided by EMF [SBPM09]. We focus therefore, on the target meta model construction as a requirement for the model-to-model transformation.

The composition of a target meta model requires knowledge of the structure of the type and expression system of the target language, and in addition, knowledge of the storage system, especially when the target model is stored in a database, XML or XMI-file. The retrieval of type and expression system information of a language can be a complex and time consuming task. Therefore, it is good advice to look for existing meta models first. The Eclipse modeling project[3] provides meta models to cover OCL. Java and Xtend meta models can be found in the Xtend repositories[4]. In other scenarios, an XSD might be available to describe the language and its storage model. For example, special computers for automation processes, called programmable logic controllers (PLC), are programmed

[3]`http://eclipse.org/modeling/mdt/`

[4]`http://git.eclipse.org/c/xtend/org.eclipse.xtend.git`

often languages specified in IEC EN 61131-3 [IEC03]. A storage meta model for these languages, formulated in an XSD, is provided by the PLCOpen group[5]. Such XSD can automatically be transformed into an EMF meta model[6]. As the PLCOpenXML specification does not handle the textual languages of the standard, this portion has to be added by hand. However, the meta model genesis from the XSD is still a big time saver, as the graph and structural based languages of the IEC EN 61131-3 are covered.

For some languages, there might not be a meta model or schema available. In that case, it has to be constructed from scratch. To start the construction, the base types of a language and the set of compositional typing rules have to be determined. The meta model for a target language type system is then constructed in the same way, as the DSL's meta model, honoring the collected typing rules.

The resulting target meta model, contains a type system taxonomy, conceptually following the structure from Figure 2, tailored for the target language. It is important to restrain the meta-model to the properties of the target language. Any addition makes the meta model more complex and harder to understand. Furthermore, such additions are often triggered by separate concerns and should therefor be modeled in a separate meta model. However, the target meta model type system can be simpler than the target language type system to restrain transformation development.

6.6 Type Transformations

The transformation of a source model into a target model is the central element of every DSL. The transformation and the semantics of the target language determine the implemented semantics of the DSL, which must adhere to the semantics specified in the DSL design. We identified four relevant activities to guide the transformation development. First, the partitioning of the transformations. Second, realization of source language concepts. Third, mapping of base types. And fourth, the mapping of complex types.

The primary separation of transformations is along type structure and expression. If a language has many different type structures, it is advisable to implement them in separate artifacts. Transformations for expressions can be divided along expression kinds, such as literals, unary and binary operators, method and function calls, or statements. With Xtend as transformation language, the partitioning is best realized by separate Xtend-classes.

Source language concepts which are not present in the target language, like garbage collection, synchronization or memory management, must be provided by a runtime system. Therefore these concepts must be identified, an API must be specified and implemented, and finally used in the transformation.

The mapping of base types is achieved by collecting them for source and target meta model together with their ranges and limitations. Even if types in source and target meta model have similar names, they can be quite different. For example, in Java int is a 32-bit signed

[5]http://www.plcopen.org/

[6]http://yoxos.eclipsesource.com/places/node/org.eclipse.xsd.ecore.converter.feature.group

integer, while in IEC EN 61131-3 INT is 16-bit signed integer and the corresponding type for the Java type would be DINT.

The mapping could become more complex, when a source meta model type, like interval, cannot be directly mapped to a compatible structure in the target meta model. For example, an interval type, based on int, has lower and upper bounds defined by arbitrary integer values. Furthermore, it may either cause an runtime exception when one bound is exceeded, or wrap, like integers do in CPU architectures. Such type can be mapped to a target model base type, however to completely realize the semantics in the target model, the identified assertions must be covered.

To ensure boundaries of source model elements in the target model, all expressions, which compute data must be identified, and checked whether they can cause a transition from a valid state to a state prohibited by the source language semantics. For example, an interval increment cannot be transformed just to a C increment operator variable ++, it must be embedded in an function either triggering an overflow error or implementing a value wrap.

In some cases, the type of the target language is more limited than the type of the source language. A typical case are strings in Turbo-Pascal, which use the first byte of a string to store the length of the string. Therefore, strings cannot exceed 255 characters. Source languages realizing strings without or with a greater capacity limit cannot be mapped to a Pascal string. A solution for this particular case are arrays or null terminated buffers, which are only limited by the underlying hardware model. To implement strings on that basis, every functionality of the source language must have a corresponding API operation.

Complex types can often be mapped to a combination of record or variant types. Of special interest are the transformation of sub-typing and sub-classing to type systems not supporting such facilities. One strategy is to use records holding references to parent types, as implemented in Objective-C [App09], or expand all types and generate records for each type. The first, method has its advantages, however, it requires references or pointers for its realization. Especially, method-calls are mapped to function pointers and held in lookup tables. For environments without pointers, like the IEC EN 61131-3, sub-typing can be realized by the second method.

7 Evaluation

We developed TS4DSL out of experiences made in different DSL development projects and the Xbase project. The evaluation and refinement of the approach was primarily executed in the MENGES project [MEG09] evolving the MENGES-DSL [GHH$^+$12]. Subsequently, we used our approach to develop languages in context of MAMBA [FvHJ$^+$11].

MENGES The project goal was to develop a DSL and tooling for electronic railway control centers utilizing industry grade PLCs. PLCs are often programmed in languages specified in IEC EN 61131-3 [IEC03]. The languages share a common type system, which provides a wide range of integers, bit vectors, records and intervals. It does not provide variant or unit types, dynamic memory management, pointers or references. The latter three are considered dangerous for reliable real-time systems.

The language was developed in an agile environment, where present methods of our project partners were formalized into a specification language. In parallel a code generator was implemented. This lead soon to a hard to maintain generator, grammar and type resolution. To solve these issues, we integrated compiler construction methods into the agile DSL process, which allowed us to continue the development in an agile fashion, by introducing new programming concepts for language users, evaluating these concepts, and modifying them according to user requests.

MAMBA The MAMBA-project addresses software measures and the interpretation of measurements taken during static or dynamic analysis of software. The measures and observations are modeled in an EMF meta model for SMM [Obj11], which is hard to compose by hand. As a solution, two languages the Measure Definition Language (MDL) and the Measure Query Language (MQL), have been developed. An SMM model can reference any EMF model representing an executable specification. Measures in an SMM model reference nodes in an specification expressed by a scope query in OCL. Therefore, MDL and MQL must be able to support the various type systems used by these specifications. The measures declared in MDL return values conforming to types declared in SMM, which consist of a base type and a unit, e.g., count, meter or time. These measures can be used in equations and are instantiable in MQL. They can handle values based on the SMM type system or the type system of the language used to specify a software system.

Early versions of the languages avoided the complexity of the combination of multiple type systems, which made type checks in expressions impossible. Errors could only be detected at runtime. Errors in scope expressions might even be overlooked and remain undetected. Our approach helped to develop better type checks and honor the various involved type systems, which make MDL and MQL applicable for bigger measuring models.

The present application of TS4DSL showed that using type system concepts are helpful to develop DSLs in a safe way and keep them maintainable while neglecting those concepts can lead to languages hard to maintain.

8 Conclusion

TS4DSL allows us to develop more complex DSLs, which exceed simple API front end DSLs. It enables DSL developers to think and use type systems in their DSLs, which allows them to implement checks and generators in a structured way. In our case studies, TS4DSL improved the quality of the developed languages and helped to keep them maintainable and extensible for future developments. Software projects, which use multiple languages to address different aspects of an application, can profit from our approach, which explicitly incorporates type systems in the DSL development process.

The development of type resolution and subsequently, type checking, with Xtend can be made in a more compact way as in Java itself. However, it is only loosely coupled with the grammar description in Xtext and algorithms have to be repeatedly implemented for every language. Therefore, the incorporation of a suitable notation for semantics and type resolution into our approach, like Xsemantics or XTS is part of our future work. In addition, we want to discuss further type concepts, like let and generics, and provide guidelines and best practice for their realization in an public accessible way. Furthermore, a better integration of syntax and semantics would be preferable, as well as better ways to compose languages and meta-models out of grammar fragments.

References

[AF11] Itemis AG and Eclipse Foundation. XText - DSL devlopment framework. Website `http://www.eclipse.org/Xtext/`, 2011.

[AG11] Itemis AG. Xtend 2. Website `http://www.eclipse.org/Xtext/documentation/2_0_0/01-Xtend_Introduction.php`, 2011.

[App09] Apple Inc. *Objective-C Runtime Programming Guide*, October 2009.

[ASU88] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilerbau, Teil 1*. Addison-Wesley, Bonn, 1988. englische Originalausgabe: Compilers—Principles, Techniques and Tools, 1986, 1987 by Bell Telephone Laboratories, Inc. übersetzt von Gerhard Barth und Mitarbeiter.

[BDJ+03] Jean Bézivin, Grégoire Dupé, Frédéric Jouault, Gilles Pitette, and Jamal Eddine Rougui. First experiments with the ATL model transformation language: Transforming XSLT into XQuery. In *2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, 2003.

[BSVC12] Lorenzo Bettini, Dietmar Stoll, Markus Völter, and Serano Colameo. Approaches and Tools for Implementing Type Systems in Xtext. In *Software Language Engineering*, Lecture Notes in Computer Science. Springer, 2012. To Appear.

[CDJ+97] Luca Cardelli, Jim Donahue, Mick Jordan, Bill Kalsow, and Greg Nelson. Type systems. In *The Computer Science and Engineering Handbook*, pages 2208–2236. CRC Press, 1997.

[EEK+12] Sven Efftinge, Moritz Eysholdt, Jan Köhnlein, Sebastian Zarnekow, Robert von Massow, Wilhelm Hasselbring, and Michael Hanus. Xbase: implementing domain-specific languages for Java. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*, GPCE '12, pages 112–121, New York, NY, USA, 2012. ACM.

[FvHJ+11] Sören Frey, André van Hoorn, Reiner Jung, Wilhelm Hasselbring, and Benjamin Kiel. MAMBA: A Measurement Architecture for Model-Based Analysis. Technical Report TR-1112, Department of Computer Science, University of Kiel, Germany, December 2011.

[GHH+12] Wolfgang Goerigk, Wilhelm Hasselbring, Gregor Hennings, Reiner Jung, Holger Neustock, Heiko Schaefer, Christian Schneider, Elferik Schultz, Thomas Stahl, Reinhard von Hanxleden, Steffen Weik, and Stefan Zeug. Entwurf einer domänenspezifischen Sprache für elektronische Stellwerke. In Stefan Jähnichen, Axel Küpper, and Sahin Albayrak, editors, *Software Engineering*, volume 198 of *LNI*, pages 119–130. GI, 2012.

[HJK+09] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Derivation and Refinement of Textual Syntax for Models. In Richard F. Paige, Alan Hartman, and Arend Rensink, editors, *Model Driven Architecture - Foundations and Applications*, volume 5562 of *Lecture Notes in Computer Science*, pages 114–129. Springer Berlin Heidelberg, 2009.

[IEC03] Deutsche Kommission Elektrotechnik Elektronik Informationtechnik im DIN und VDE, Beuth Verlag, Berlin. *IEC EN 61131-3*, 2003-12 edition, 2003.

[JBC+06] Frédéric Jouault, Jean Bézivin, Charles Consel, Ivan Kurtev, and Fabien Latry. Building DSLs with AMMA/ATL, a Case Study on SPL and CPL Telephony Languages. In *Proceeding of the 1st ECOOP Workshop on Domain-specific Program Development (DSPD)*, July 3rd 2006.

[Jun12] Reiner Jung. Introducing Type-Systems in Xtext-Languages. Website `http://se.informatik.uni-kiel.de/news/introducing-type-system-in-xtext/`, October 2012.

[LMB+01] Ákos Lédeczi, Miklós Maróti, Árpád Bakay, Gábor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle, and Péter Völgyesi. The Generic Modeling Environment. In *Workshop on Intelligent Signal Processing*, 2001.

[MEG09] Verbundprojekt 5 im Kompetenzverbund Software Systems Engineering. Technical report, CAU, Institut für Informatik, 2009.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, December 2005.

[MLC06] Gergely Mezei, Tihamér Levendovszky, and Hassan Charaf. Visual Presentation Solutions for Domain Specific Languages. In *Proceedings of the IASTED International Conference on Software Engineering*, Innsbruck, Austria, 2006.

[Obj06] Object Management Group (OMG). Meta Object Facility (MOF) Specification 2.0 Core. formal/2006-01-01, April 2006.

[Obj11] Object Management Group, Inc. Architecture-Driven Modernization (ADM): Structured Metrics Meta-Model (SMM), V. 1.0 Beta 3. `http://www.omg.org/spec/SMM/`, 2011.

[OMG05] OMG. *MOF QVT Final Adopted Specification*. Object Management Group (OMG), June 2005.

[OMG06] OMG. *Object Constraint Language Object Constraint Language, OMG Available Specification, Version 2.0*. Object Management Group (OMG), 2006.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

[SBPM09] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, Boston, MA, 2. edition, 2009.

[Ste12] Eike Stepper. Connected Data Objects (CDO). Website `http://www.eclipse.org/cdo/documentation/index.php`, seen November 2012.

[VP12] Markus Voelter and Vaclav Pech. Language modularity with the MPS language workbench. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 1449–1450, Piscataway, NJ, USA, 2012. IEEE Press.

[Vö11] Markus Völter. Xtext/TS - a Typesystem Framework for Xtext. Website `http://www.infoq.com/articles/xtext_ts`, 2011.