Xbase: Implementing Domain-Specific Languages for Java*

Sven Efftinge
itemis AG
D-24143 Kiel, Germany
sven.efftinge@itemis.de

Moritz Eysholdt
itemis AG
D-24143 Kiel, Germany
moritz.eysholdt@itemis.de

Jan Köhnlein
itemis AG
D-24143 Kiel, Germany
jan.koehnlein@itemis.de

Sebastian Zarnekow
itemis AG
D-24143 Kiel, Germany
sebastian.zarnekow@itemis.de

Wilhelm Hasselbring
Software Engineering Group
University of Kiel, Germany
wha@informatik.uni-kiel.de

Robert von Massow
Software Engineering Group
University of Kiel, Germany
rvm@informatik.uni-kiel.de

Michael Hanus
Programming Languages and Compiler Construction Group
University of Kiel, Germany
mh@informatik.uni-kiel.de

ABSTRACT

Xtext is an open-source framework for implementing external, textual domain-specific languages (DSLs). So far, most DSLs implemented with Xtext and similar tools focus on structural aspects such as service specifications and entities. Because behavioral aspects are significantly more complicated to implement, they are often delegated to general-purpose programming languages. This approach introduces complex integration patterns and the DSL's high level of abstraction is compromised.

We present Xbase as part of Xtext, an expression language that can be reused via language inheritance in any DSL implementation based on Xtext. Xbase expressions provide both control structures and program expressions in a uniform way. Xbase is statically typed and tightly integrated with the Java type system. Languages extending Xbase inherit the syntax of a Java-like expression language as well as language infrastructure components, including a parser, an unparser, a linker, a compiler and an interpreter. Furthermore, the framework provides integration into the Eclipse IDE including debug and refactoring support.

The application of Xbase is presented by means of a domain model language which serves as a tutorial example and by the implementation of the programming language Xtend. Xtend is a functional and object-oriented general purpose language for the Java Virtual Machine (JVM). It is built on top of Xbase which is the reusable expression language that is the foundation of Xtend.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific architectures*; D.2.13 [**Software Engineering**]: Reusable Software—*Reusable libraries*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Classes and objects*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Inheritance*; D.3.4 [**Programming Languages**]: Processors—*Code generation*

General Terms

Design, Languages

Keywords

Domain-specific languages, Object-oriented programming, Language inheritance

1. INTRODUCTION

The simplicity and application-specific syntax of DSLs increases the readability as well as the density of information. Hence, developers using DSLs cannot only develop systems faster but also understand, evolve, and extend existing systems easier.

There are two different but converging approaches for designing DSLs. On the one hand, there are the so called *external DSLs* [7]. An external DSL is a completely independent language built from scratch, such as the AWK[1] programming language for file processing or the MENGES language

*This work is supported by the German Federal Ministry of Education and Research (BMBF) under grant number 01S10008

GPCE'12, September 26–27, 2012, Dresden, Germany.

[1] `http://www.gnu.org/software/gawk/`

for programming railway control centers [9]. As external DSLs are independent from any other language, they need their own infrastructures like parsers, linkers, compilers or interpreters. More popular examples for external DSLs are SQL [2] or the *Extended Backus-Naur Form* (EBNF) [22] to easily interact with relational databases or representing formal grammars, respectively.

On the other hand, an internal DSL (sometimes also called *embedded DSL*) leverages a so called host language to solve domain-specific problems. This idea originates from the LISP programming language, where several internal DSLs have been designed [10]. Examples of modern internal DSLs are the OpenGL API, the Java Concurrency API, or the Scala parser combinators. An internal DSL is basically an API which pushes the host language's capabilities to make client code as dense and readable as possible.

As internal DSLs are defined on top of a general purpose programming language, their syntax is constrained to what the host language allows. The advantage is that no new language infrastructure needs to be built, because the parser and compiler (resp. interpreter) of the host language is used. Also it is possible to call out to and reuse the concepts of the host language, such as expressions. The downside is the limited flexibility, because internal DSLs have to be expressed using concepts of the host language. Also, internal DSLs typically do not have special IDE support or domain-specific compiler checks.

A domain-specific language (DSL) as we understand the term is a programming language or specification language dedicated to a particular problem or solution domain. Other definitions focus on the aspect that a DSL is explicitly not intended to be able to solve problems outside of that particular domain [7], but from our viewpoint this is not an important distinction.

Creating a DSL and tools to support it can be worthwhile if the language allows one to express a particular type of problem or solution more clearly than an existing language. Furthermore, the kind of problem addressed by the DSL should appear frequently—otherwise it may not be worth the effort to develop a new language.

Xtext[2] is a framework for external textual DSLs. Developing textual DSLs has become straightforward with Xtext because the framework provides a universal architecture for language implementations and well-defined default services. Structural languages which introduce new coarse-grained concepts, such as services, entities, value objects or state-machines can be developed with very little effort. However, software systems do not only consist of structures. At some point a system needs to have some behavior which is usually specified using statements and expressions. Expressions are the heart of every programming language and their implementation is error prone. On the other hand, expressions are well understood and many programming languages share a common set and understanding of expressions. Unfortunately, implementing a language with support for statically typed expressions is complicated and requires a great deal of work. As a consequence, most designers of external DSLs do not include support for expressions in their DSL but provide some workarounds. A typical workaround is to define only the structural information in the DSL and add behavior by modifying or extending the generated code. It is obviously inconvenient and error prone to write, read and maintain information which closely belongs together in two different places, abstraction levels and languages. Additionally, the techniques used to add the missing behavioral information impose significant additional complexity to the system. So far, for most projects, it seems more attractive to apply these workarounds than to implement an expression language from scratch.

The novel contribution of this paper is Xbase. It is a reusable expression language which can be used in any Xtext language. It comes with a complete grammar for a modern Java-like expression language as well as all the required infrastructure:

- A parser and a lexer creating a strongly typed abstract syntax tree (AST).
- A compiler translating the AST to Java source code, or alternatively an interpreter running on the Java virtual machine (JVM).
- A type system model, designed after Java's type system and supporting all Java 5 concepts, including generics.
- A linker that is implemented according to the Java scoping rules and binds against concept of the Java type system.
- Advanced editor features, such as content assist, syntax coloring, code folding and error highlighting.
- Integration with Eclipse, such as call hierarchy view, type hierarchy view and other useful navigation features.
- A debugger allowing one to alternatively step through the DSL or through the generated Java source.

Reusing Xbase in a language can be done by defining two artifacts: A grammar that extends the provided Xbase grammar and a Java type inferrer that maps the DSL concepts to the Java type model.

We first describe language development with Xtext in Section 2. Section 3 introduces the Xbase language by giving an overview on design goals and some examples for Xbase expressions. Two applications of Xbase are presented in Section 4. In Section 5, we take a look at some other work in this field and relate it to Xbase. Finally, we draw our conclusions and give an outlook on some future work in Section 6.

2. THE XTEXT FRAMEWORK

Xtext is an extensible language development framework covering all aspects of language infrastructure such as parsers, linkers, compilers, interpreters and full-blown IDE support based on Eclipse. It is successfully used in academia as well as in industrial applications. Among the industrial users are projects for embedded system development, such as AUTOSAR [15] and enterprise information systems [4, 6].

Xtext is hosted at Eclipse.org[3] and is distributed under the *Eclipse Public Licence* (EPL). With Xtext, the DSL specification is defined in a grammar resembling EBNF from

[2] `http://www.eclipse.org/Xtext/`

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

which parsers, linkers and other parts of the language infrastructure are derived. We give an example of that in Section 2.1. Section 2.2 explains how dependency injection facilitates tool chain adaptability and Section 2.3 explains how language inheritance works with Xtext.

2.1 A Simple Domain Model Language

As an example of how to build DSLs using Xtext, we present a simple language to define domain models. The language is able to describe entities and their relations. Entities have a name and some properties. The type of a property is either a defined entity or some predefined data type. Listing 1 shows an example of a domain model defined with this language.

Listing 1: An example domain model

```
1  datatype String
2
3  entity Person {
4     name : String
5     givenName : String
6     address : Address
7  }
8
9  entity Address {
10    street : String
11    zip : String
12    city : String
13 }
```

The corresponding grammar definition is given in Listing 2. In Line 1, the grammar is declared by giving it a fully qualified name. The file name needs to correspond to the language name with file extension xtext. In Line 3, the Xtext generator is instructed to infer a type-safe AST from the given grammar (nothing more has to be done by the programmer to get the AST). Starting with Line 5, the rules of the grammar are given. A rule is defined using a sequence of terminals and non-terminals. A quoted string declares a terminal. Alternatives are marked by a vertical bar. Assignment operators are used to define how an AST should be constructed by the parser. The identifier on the left-hand side refers to a property of the AST class, the right-hand side is a terminal or non-terminal. Xtext grammars are bi-directional, i.e., a parser as well as an unparser, aka serializer, is derived from an Xtext grammar.

Note that terminal rules such as ID are inherited from the predefined Xtext terminals grammar (see with-clause in Line 1), and do not need to be defined again (see also Section 2.3). Line 15 contains a construct in square brackets which refers to a type of the AST. This defines a cross reference to another element in the parsed model which is automatically linked according to the current scope definition.

Listing 2: A simple domain model language

```
1  grammar org.example.domainmodel.DomainModel with
       org.eclipse.xtext.common.Terminals
2
3  generate domainmodel "http://www.example.org/
       domainmodel/Domainmodel"
4
5  Domainmodel :
6    elements += Type*;
7
8  Type:
9    DataType | Entity;
10
11 DataType:
12   'datatype' name = ID;
13
14 Entity:
15   'entity' name = ID ('extends' superType = [Entity])? '
         {'
16     features += Feature*
17   '}'
18
19 Feature:
20   name = ID ':' type = [Type]
21 ;
```

A class diagram of the AST classes derived from the grammar in Listing 2 is shown in Figure 1.

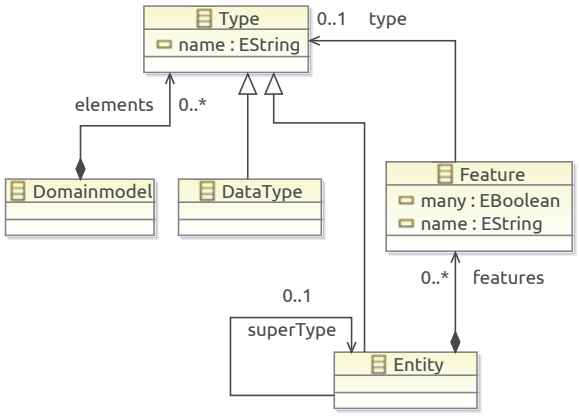

Figure 1: A class diagram of the generated AST model for the domain model language, as specified in Listing 2.

2.2 Tool Chain Adaptability

Not every aspect of a language implementation can be expressed in an Xtext grammar. To allow for arbitrary adaptation of the default functionality, the Xtext framework makes heavy use of the dependency injection pattern [19]. Dependency injection allows one to control which concrete implementations are used in a language configuration. Because of this flexibility, the framework can provide default behavior for almost every aspect of a language infrastructure. If the default behavior does not fit for the language at hand, the implementation can be changed easily by binding an interface to another compatible class type.

2.3 Language Inheritance

Xtext supports *language inheritance* which allows to reuse an existing grammar [3]. Similar to inheritance in object-oriented languages, the symbols (terminals and non-terminals) of the parent grammar can be used as if they were defined in the inheriting grammar. Their definition can also be overridden in the inheriting grammar. This way it is possible to refine and extend arbitrary Xtext grammars. For instance, Line 1 of Listing 2 above shows the import of another language, containing the definition of the terminal ID.

This mechanism is also employed to reuse the Xbase expression language, as described in the following section.

3. DESIGN OF THE REUSABLE EXPRESSION LANGUAGE XBASE

To lower the barrier for users already familiar with the Java programming language, Xbase expressions resemble Java's expressions and statements in terms of syntax and semantics. Additionally, Xbase expressions support advanced concepts such as type inference, extension methods and various forms of syntactic sugar facilitating writing more concise and readable code. The expression language is statically typed and is linked to the Java type system, which renders it fully interoperable with Java. This implies that it is possible to reuse any Java classes, i.e., to subclass them or call out to their methods, constructors and fields.

The implementation consists of a grammar definition, as well as reusable and adaptable implementations for the different aspects of a language infrastructure such as an AST structure, a compiler, an interpreter, a linker, and a static analyzer. In addition, it provides the integration of the expression language with an Eclipse IDE. Default implementations for tool features such as content assistance, syntax coloring, hovering, folding and navigation are automatically integrated and reused within any language inheriting from Xbase.

In the following subsections we introduce some concepts of the Xbase expression language. We start with an introduction to Xbase expressions in Section 3.1, including features like type inference, operator overloading, and lambda expressions. Then we describe the concept of extension methods in Section 3.2. Finally, we show how Xbase integrates with Java in Section 3.3.

3.1 Expressions

Expressions are the main language constructs that are used to express behavior and computation of values. A separate concept of statements is not supported with Xbase. Instead, powerful expressions are used to handle situations in which the imperative nature of statements would be helpful. An expression always results in a value (which might be null). In addition every resolved expressions is of a static type, inferred at compile time. As in other languages, expressions can be formed by composing variables, constants, literals, or expressions with each other using operators or method calls.

3.1.1 Type Inference

Xbase is a statically typed language, based on the Java type system. In Java, type information must often be specified redundantly which is perceived to be verbose in the code. For example, in Java a variable declaration is defined like this:

```
final ArrayList<Integer> ls = new ArrayList(1, 2, 3)
```

Xbase allows one to omit the redundant definition of types, because types can be inferred from the expression on the right-hand side. Instead one uses the keyword **val** to declare a local final variable. The method newArrayList() is a static helper from dependency injection framework Google Guice.[4]

```
val ls = newArrayList(1, 2, 3)
```

While the keyword **val** implies a final variable, the keyword **var** is used to declare mutable variables.

Type inference is also heavily used for type parameters in method and constructor calls and to infer the parameter types as well as the return type of a lambda expression.

3.1.2 Lambda Expressions

A lambda expression is a literal that defines an anonymous function. It also captures the current scope such that any final variables and parameters visible at construction time can be referred to in the lambda expression's body. Lambdas can be used wherever a type with a single method is expected. This includes all functional interfaces as defined in the current draft of JSR 335 (Project Lambda) [13]. A lambda expression will result in an instance of that interface, implementing the single method using the given expression.

Consider, for example, the following Java method signature as defined in java.util.Collections:

```
public static <T> List<T> sort(List<T> list, Comparator<
    T> comparator)
```

With Xbase, it can be invoked using a lambda expression such as:

```
sort(listOfPersons, [p1, p2 | p1.name.compareTo(p2.name)])
```

All needed type information is automatically inferred from the current context.

Lambda expressions allow for convenient usage of higher-order functions [12] and also simplify the use of many existing Java APIs which are designed for use with anonymous classes in Java.

3.1.3 Operator Overloading

In contrast to Java, the semantics and the applicability of operators is not hard-coded into the language. Instead, all operators are delegated to methods with a specific signature depending on the operator. This allows for defining and implementing operators for any Java type.

One could, for instance, support arithmetic on a custom type "distance". Together with a skillful application of extension methods, the following code can be written:

```
12.cm + 44.mm == 164.mm
```

Just like in Java, the precedence and associativity of the operators is predefined and cannot be changed.

By extending the Xbase language in a DSL definition, it is also possible to introduce new operators and define a

[4] http://code.google.com/p/google-guice/

method mapping for them. Predefined operators can also be removed by overriding and deactivating their default definitions in the grammar.

3.2 Extension Methods

An extension method is a method that can be invoked as if it were a member of a certain type, although it is defined elsewhere. The first argument of an extension method becomes the receiver. Xbase comes with a small standard library that defines so-called extension methods for existing types from the Java Runtime Environment (JRE).

For example, an extension method for java.lang.String is defined which turns the first character of a given string to upper case leaving the others as is:

```
public static String toFirstUpper(String receiver)
```

Because Xbase puts this static Java method as an extension method on the scope of java.lang.String, it can be used like this:

```
"some text".toFirstUpper()
```

Xbase's standard library comes with many useful extension methods adding support for various operators and higher-order functions for common types, such as java.lang.String, java.util.List, etc.

Languages extending Xbase may contribute additional methods and can change the whole implementation as it seems fit. There are various ways in which extension methods can be added. The simplest possibility is an explicit listing of types that provide extensions to existing classes. The language designer can predefine this list. This implies that language users cannot add additional library functions.

An alternative is to have them looked up by a certain naming convention. A third option is to allow users to import extension methods on the client side. This is often the most valuable way especially for general purpose languages. The approach can be seen in the language Xtend, where extension methods are added using a special keyword on field declarations, see Section 4.2.

The precedence of extension methods in Xbase is lower than real member methods, i.e., one cannot override member features. Moreover, extension members are not invoked polymorphically. For example, if there are two extension methods available (foo(Object) and foo(String)) the expression (foo as Object).foo would bind and invoke foo(Object). The binding rules are the same as for overloaded Java methods that are statically bound according to the compile time type of an expression. That is because extension invocations are actually shortcuts for method calls with parameters. myValue.extensionCall() is equivalent to extensionCall(myValue).

3.3 Model Inference

The method calls, constructor invocations, and field references are linked against Java's type model. That type model is populated from real Java classes but may also be instantiated as a representation of arbitrary models. They may be based on the AST of a DSL script. This allows for the linking of non-Java elements. The Java type model in Xbase consists of all structural concepts of Java, such as classes, interfaces, enums, and annotations and also includes all possible members such as methods, fields, constructors, or enum literals. Also Java generics are fully supported.

Xbase uses the type model during type and method resolution. In order to refer to concepts from a DSL, there must be a mapping to the Java type model. For example, a property in the domain model language might be translated to a Java field and two Java methods, one for the getter and one representing the setter method. Given such a translation, an expression can now invoke the inferred field, getter or setter method. The visibility constraints are applied according to the Java language specification. Also a generic code generator is available which automatically transforms the Java type model to compilable Java source code.

In addition, the variable scope of an expression is defined by associating it with a certain Java element. For example, consider the domain model language where a Java field, a getter and a setter method is inferred from a property. Imagine one would want to specify a custom implementation for the setter of a property using the following notation:

```
entity Person {
    name : String set {
            if (name == null)
              throw new NullPointerException()
            fName = name
          }
}
```

To define a proper scope for the expression in the set block, one would only have to associate the expression with the derived setter method. Doing this automatically makes any declared parameters and access to the instance (**this**) available: The expression is now logically contained in the derived setter method. The scope of the expression is defined accordingly. Also the expected type is declared through the return type of the method, so incompatible return expressions will be marked with an error. Xbase expressions can be associated with derived Java methods, constructors, and fields. The generic code generator not only generates declarations for the inferred Java model but will also generate proper Java statements and expressions for the associated Xbase expressions.

Within the explained Java model inference, a Java model is inferred from the DSL's AST and all used expressions are associated with elements in the inferred Java model. Doing so provides the information needed to define the scopes of the expressions and how the DSL is translated to executable Java code. The Java model inference is a central aspect when implementing a language inheriting from Xbase. The model inference together with a grammar definition is sufficient to define executable languages with Xbase, including advanced IDE support.

A Java model inference is usually implemented in Xtend, a statically typed programming language, which itself is built with Xbase and presented in Section 4.2.

4. APPLICATIONS OF XBASE

In this section we present two examples of how Xbase can be applied to develop DSLs. The first example is an extension to the introductory Xtext example presented in Section 2. We add support for operations, the Java type system, and for building web services. In the second example we show how the Xtend programming language is implemented based on Xbase.

4.1 The Extended Domain Model Example

In the introductory example in Section 2, we presented a simple language to define entities, datatypes and relationships. The language does not support expressions and uses a very primitive type system, consisting of data types and entities. Because we initially did not allow expressions, the type system did not contribute compatiblity rules. In this section, support for the Java type system, including generics, and expressions are added to this language using Xbase.

4.1.1 The Extended Grammar

At first the grammar needs to inherit from org.eclipse.xtext.xbase.Xbase, see Line 1 in Listing 3. This allows one to use the rules defined in the Xbase grammar, that is, for instance, needed to implement operations using these expressions. Second, as we intend to add Java type system support, we no longer need our own types. Hence, we replace the symbol Type by a new symbol AbstractElement, see Lines 8–9 in Listing 3. This non-terminal denotes a package declaration, an entity, or an import. The package declaration and the contained entities resemble the Java package structure.

As we now intend to create and reuse Java classes, we also modify the definition of Entity, starting with Line 19 in Listing 3. Instead of being able to extend other entities, we now want to allow arbitrary Java classes as the super type of entites. To do this, we simple exchange the [Entity] reference with JvmParametrizedTypeReference. This rule is inherited from Xbase and allows to declare full Java type references, e. g. java.lang.String or java.util.Map<Integer, String>.

Next, we want to add support for operations to the domain model language. In the first example, entities could only declare properties. As an entity now can contain operations as well, we introduce a new symbol called Feature that delegates to a Property or an Operation. Finally, we have to define a syntax for the operations. The signature of an operation starts with the keyword **op** followed by a name, a list of parameters and, separated by a colon, a return type. The body of the operation is defined to be an XBlockExpression which is a non-terminal defined as part of Xbase. The full grammar is given in Listing 3. For details on the employed syntax and the reused Xbase non-terminals, please refer to the Xtext documentation.[5]

4.1.2 Java Type Model Inference

To be able to integrate the entities into the Java type system, we have to map the concepts of the domainmodel DSL to the Java types. The DomainmodelJvmModelInferrer, which takes care of this, is given in Listing 4. It is implemented in Xtend, see Section 4.2.

For each entity it defines a new Java class (Line 8) that declares several members (Line 14). For properties, this is a private field and the respective getter and setter method. For operations, we convert the signature from the DSL to a Java method signature and make it public. As the method's body, we simply assign the XBlockExpression as the logical container, see Line 32 in Listing 3. The platform takes care of that in the body assignment. Creating a JVM model also enables the compiler to generate the full Java code from the model.

[5] http://www.eclipse.org/Xtext/documentation/

Listing 3: The grammar of the domain model language

```
1  grammar org.eclipse.xtext.example.domainmodel.
       Domainmodel with org.eclipse.xtext.xbase.Xbase
2
3  generate domainmodel "http://www.xtext.org/example/
       Domainmodel"
4
5  DomainModel:
6          elements+=AbstractElement*;
7
8  AbstractElement:
9          PackageDeclaration | Entity | Import;
10
11 Import:
12         'import' importedNamespace=
               QualifiedNameWithWildCard;
13
14 PackageDeclaration:
15         'package' name=QualifiedName '{'
16                 elements+=AbstractElement*
17         '}';
18
19 Entity:
20         'entity' name=ValidID ('extends' superType=
               JvmParameterizedTypeReference)? '{'
21                 features+=Feature*
22         '}';
23
24 Feature:
25         Property | Operation;
26
27 Property:
28         name=ValidID ':' type=JvmTypeReference;
29
30 Operation:
31         'op' name=ValidID '(' (params+=
               FullJvmFormalParameter (',' params+=
               FullJvmFormalParameter)*)? ')' ':' type=
               JvmTypeReference
32                 body=XBlockExpression;
33
34 QualifiedNameWithWildCard :
35         QualifiedName ('.' '*')?;
```

Listing 4: The Java Model inferrer of the domain model language

```
class DomainmodelJvmModelInferrer extends
      AbstractModelInferrer {

    @Inject extension JvmTypesBuilder
    @Inject extension IQualifiedNameProvider

    def dispatch infer(Entity entity,
          IJvmDeclaredTypeAcceptor acceptor, boolean
          prelinkingPhase) {
        acceptor.accept(
            entity.toClass( entity.fullyQualifiedName)
        ).initializeLater [
            documentation = entity.documentation
            if (entity.superType != null)
                superTypes += entity.superType.
                      cloneWithProxies

            for ( f : entity.features ) {
                switch f {

                    Property : {
                        members += f.toField(f.name,
                              f.type)
                        members += f.toGetter(f.
                              name, f.type)
                        members += f.toSetter(f.name
                              , f.type)
                    }

                    Operation : {
                        members += f.toMethod(f.
                              name, f.type) [
                            documentation = f.
                                  documentation
                            for (p : f.params) {
                                parameters += p.
                                      toParameter(p.
                                      name, p.
                                      parameterType)
                            }
                            body = f.body
                        ]
                    }
                }
            }
        ]
    }

}
```

Listing 5: A domain model with behavior

```
package my.social.network

import java.uil.List

entity Person {
  firstName : String
  lastName : String
  friends : List<Person>

  op getFullName() : String {
    return firstName + "␣" + lastName
  }

  op getSortedFriends() : List<Person> {
    return friends.sortBy[fullName]
  }
}
```

Based on these two artifacts, one is now able to define entities that link against Java types and may include definitions of operations including their behavior. At the same time, model definitions are translated to valid Java source code. Listing 5 shows an example of such a model definition.

4.1.3 Building Web Services

In addition to the aforementioned extensions, another extension for compiling RESTful web services and *Java Persictence API* (JPA)[6] compatible entity classes has been developed as an evaluation scenario for the Xbase language and tools.

The extensions are implemented by adding mappings to the JVM model inferrer, which generates two additional classes for each entity, in addition to the original DomainmodelJvmModelInferrer. One class is a so called *data access object* (DAO) class, that encapsulates database logic to create, retrieve, update and delete persisted entities via JPA's database access functionality. The other class provides RESTful web service bindings for each of the operations provided by the DAO class. These web services are defined using the JAX-RS,[7] the Java API for RESTful web services, which means that methods and classes are annotated with REST-specific annotations for URL and parameter mapping as well as HTTP content negotiation. This extension is available as open source software.[8]

4.2 Xtend

Xtend[9] is a statically typed, functional, and object-oriented programming language for the JVM. Xtend is developed at Eclipse.org and is freely available under the Eclipse Public License (EPL), similar to Xtext. Its primary goal is to provide a significantly more concise notation than Java, without switching to a whole new kind of language. The language improves on Java by removing the need for writing redundant and superfluous information. In addition, it adds some new language features. For instance, Xtend's template expressions are designed to concatenate strings, e.g.,

[6] http://jcp.org/en/jsr/detail?id=317
[7] http://jcp.org/en/jsr/detail?id=311
[8] https://github.com/RvonMassow/Xrest
[9] http://www.eclipse.org/xtend/

Listing 6: Multi-methods in Xtend

```
class MultiMethods {
  def dispatch overloaded(String s) { return 'string' }
  def dispatch overloaded(Object s) { return 'object' }
}
```

for generating web pages. Xtend inherits all of the features from Xbase but adds an additional expression for multi-line strings and allows for declaring classes, methods, and so on.

4.2.1 Main Concepts

One of the main design goals of Xbase and also Xtend was to eliminate the need to write boiler plate code by providing reasonable defaults. For instance, classes and methods are public by default and type inference for variables and method declarations allows to infer the return type of operations, thus, eliminating the need to define that redundantly.

Generally, Xtend reuses the concepts and even the keywords of Java, such that new users can feel comfortable immediately. Like Java classes, Xtend classes can declare members, i.e., fields and methods. Fields have almost the same syntax as in Java. The only difference is that the field's name is optional if the field is used as an extension provider. That means, a local field can provide extension methods, which is a powerful feature especially when used in conjunction with the dependency injection pattern. Static imports can act as extension providers, too.

Xtend derives its name from its extensive support of extension methods.

4.2.2 Xtend Template Expressions

Xtend adds one expression to the set of Xbase expessions: the template expression. It is a multi-line string literal which supports string interpolation (i.e., embedded expressions). The template expression is defined by extending the Xbase grammar and adding several additional rules.

They also require additional scoping rules and an implementation in the compiler and interpreter. Finally, the type provider needs to know the type of a template expression. All these aspects are implemented by extending the original services provided by Xbase and by adding another case for template expressions. The necessary adaptions are hooked in by means of dependency injection only, i.e., without touching any line of the existing Xbase code.

4.3 Dispatch methods

Like for any other Xbase language, a model inferrer is implemented mapping the various concepts from Xtend to Java. In most cases the translation is a straight mapping for Xtend. In one case, however, the mapping is more interesting: Xtend supports multi-methods which allow one to overload a method for different parameter types. In contrast to regular method resolution done at compile time, a multi-method is bound at run time based on the run-time type. This behavior is implemented through a synthetic dispatch method using the most common parameter types of all overloaded declarations.

For example, given the definition in Listing 6, the Java code in Listing 7 is generated. This mapping is implemented in the Java type model inferrer.

Listing 7: Multi-methods in Xtend

```
public class MultiMethods {

  protected String _overloaded(String s) { return "
      string"; }
  protected String _overloaded(Object s) { return "
      object"; }

  public String overloaded(Object s) {
    if (s instanceof String)
      return _overloaded((String)s);
    else
      return _overloaded(s);
  }
}
```

5. RELATED WORK

This paper presents Xtext as a framework for implementing external DSLs. Below, we take a look at some related work. First, we compare Xtext to other frameworks for external DSL implementation. Second, we take a look at other JVM languages supporting the definition of internal DSLs.

5.1 External DSL Frameworks

Several frameworks for external DSL development exist besides Xtext [14, 17, 20]. In this section, we exemplary compare the Xtext framework with the Meta Programming System developed by Jetbrains.

The *Meta Programming System* (MPS) by JetBrains[10] is another DSL development tool. In contrast to Xtext, MPS uses projectional editors for both, DSL development itself and the IDE for that DSL. This means, languages are not based on a plain text grammar. Instead, the developer creates a meta model of the AST by defining concepts. For the representation to the user, the DSL developer creates editors, which can be textual or even graphical, showing a representation of the AST. These editors operate directly on the underlying AST.

Similar to Xtext, MPS comes with a base language. This base language can be reused in any MPS DSL. It provides support for expressions, statements, typing, and operations (i.e., feature calls) and can also be directly compiled to Java statements, expressions, and feature calls.

Dissimilar to Xtext, the use of projectional editing prevents the tooling from stumbling over syntactic ambiguities in the DSL. Thus, it is possible to compose arbitrary languages. While the AST is always unambiguous, this is not necessarily the case for the projected syntax, which can be confusing when working with MPS.

5.2 Internal DSLs

In contrast to external DSLs, for which special editors and compilers have to be developed, it is also possible to embed DSLs into a general purpose language which serves as the host for the DSL. For such approaches, the host language's infrastructure can be reused completely. The downside of this approach is that the infrastructure can rarely be tailored specifically for the DSL.

[10] http://www.jetbrains.com/mps/

For developing internal DSLs, some languages have proved to be well-suited, due to their flexible syntax. One of the oldest and at the same time most powerful languages which are used to develop internal DSLs is Lisp. Later on, functional languages, like Haskell, have been often advocated to implement internal DSLs, e.g., for pretty printing [21], music composition [11], or financial contract analysis [18].

In the following, we discuss the approaches for DSL development in Groovy and Scala and compare this to the Xtext approach.

5.2.1 Groovy

Groovy is a dynamically typed language that runs on the Java Virtual Machine [16].

Like Xtend, Groovy has support for operator overloading and lambda expressions. Because Groovy supports run-time meta-programming, it is possible to add new methods to classes at run time. Categories are a convenient feature to add new methods to existing classes. A special DSL for the Eclipse plug-in allows one to tell the IDE about applied categories. As a result, users will get rudimentary content assistance.

Groovy's compile-time meta-programming capabilities allow for rewriting the AST at compile time as well.

5.2.2 Scala

Scala is a statically typed language for the JVM. It is a blend of functional and object-oriented concepts and is often used as a host language for DSLs. It offers various possibilities to adapt the syntax to the needs of the DSL, e.g., by operator overloading, implicits, and closures [8]. These features allow one to define very concise internal DSLs.

In addition, the Scala group is working on so-called *language virtualization*. Language virtualization, which combines polymorphic embedding of languages and a staged compilation process, supports the application of domain-specific optimizations [1]. This mechanism is achieved by decoupling the DSL definition from its execution. The DSL is defined in an abstract manner, by defining the types and operations on them. These DSL concepts are mapped to actual implementations in several stages in which domain-specific optimizations can be applied in the form of AST rewriting. Another very similar approach is covered in SIP-16 "Self Cleaning Macros" [5], which introduces a convenient way to declare AST transformations.

5.2.3 Summary

These efforts demonstrate how general purpose languages are enhanced to support the advantages of external DSLs. While tools for external DSLs, such as Xtext, move toward supporting the advantages of internal DSLs by allowing for the reuse of sub-languages such as Xbase, general purpose languages introduce compile-time meta programming and language virtualization to support the advantages of an external DSL, such as compiling to arbitrary platforms. The two approaches seem to converge eventually.

6. CONCLUSIONS AND FUTURE WORK

In this paper we presented Xbase, an expression language library which can be integrated into domain-specific languages built with the Xtext language development framework. We discussed the main concepts of the design and implementation of the Xbase language and demonstrated its application to two different languages. Xbase significantly reduces the effort to implement domain-specific languages with behavioral aspects running on the Java Virtual Machine.

Although Xbase has been available for less than a year, it is already employed in industry and open-source projects. Examples are Jnario,[11] a language for executable specifications, and a domain-specific language to script CAD systems, developed by Sandvik Coromant which has been presented at EclipseCon 2012.[12]

For the future, we plan to support translations of Xbase expressions to other target languages, such as JavaScript, C and Objective-C.

7. REFERENCES

[1] Hassan Chafi, Zach DeVito, Adriaan Moors, Tiark Rompf, Arvind Sujeeth, Pat Hanrahan, Martin Odersky, and Kunle Olukotun. Language Virtualization for Heterogeneous Parallel Computing. Technical report, EPFL, 2010.

[2] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13:377–387, June 1970.

[3] Sven Efftinge, Moritz Eysholdt, Jan Köhnlein, Sebastian Zarnekow, and Contributors. *Xtext 2.2 Documentation*, December 2011.

[4] Sven Efftinge, Sören Frey, Wilhelm Hasselbring, and Jan Köhnlein. Einsatz domänenspezifischer Sprachen zur Migration von Datenbankanwendungen. In *Datenbanksysteme für Business, Technologie und Web (BTW 2011)*, volume P-180 of *Lecture Notes in Informatics*, pages 554–573, Kaiserslautern, March 2011.

[5] Eugene Burmako, Martin Odersky, Christopher Vogt, Stefan Zeiger, Adriaan Moors. *Self Cleaning Macros (SIP 16)*, March 2012.

[6] M. Eysholdt and J. Rupprecht. Migrating a large modeling environment from XML/UML to Xtext/GMF. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 97–104. ACM, 2010.

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

[8] Debasish Ghosh. *DSLs in Action*. Manning Publications, pap/psc edition, December 2010.

[9] Wolfgang Goerigk, Reinhard von Hanxleden, Wilhelm Hasselbring, Gregor Hennings, Reiner Jung, Holger Neustock, Heiko Schaefer, Christian Schneider, Elferik Schultz, Thomas Stahl, Steffen Weik, and Stefan Zeug. Entwurf einer domänenspezifischen Sprache für elektronische Stellwerke. In *Software Engineering 2012*, volume P-198 of *Lecture Notes in Informatics (LNI)*, pages 119–130. GI, March 2012.

[10] Peter Henderson. Functional geometry. In *Symposium on LISP and Functional Programming*, pages 179–187, 1982.

[11] P. Hudak. Describing and interpreting music in Haskell. In J. Gibbons and O. de Moor, editors, *The*

[11] `http://jnario.org/`

[12] `http://www.eclipsecon.org/2012/sessions/desagn-dsl-engineer-order`

Fun of Programming, pages 61–78. Palgrave Macmillan, 2003.

[12] J. Hughes. Why functional programming matters. In D.A. Turner, editor, *Research Topics in Functional Programming*, pages 17–42. Addison Wesley, 1990.

[13] Java Community Process. *Lambda Specification (Early Draft Review #1)*, November 2011.

[14] Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench. Rules for declarative specification of languages and IDEs. In Martin Rinard, editor, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 444–463, October 2010.

[15] Olaf Kindel and Mario Friedrich. *Softwareentwicklung mit AUTOSAR*. dpunkt.verlag, 2009.

[16] Dierk Koenig, Andrew Glover, Paul King, Guillaume Laforge, and Jon Skeet. *Groovy in Action*. Manning Publications Co., 2007.

[17] Holger Krahn, Bernhard Rumpe, and Steven Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12:353–372, September 2010.

[18] S.L. Peyton Jones and J.-M. Eber. How to write a financial contract. In J. Gibbons and O. de Moor, editors, *The Fun of Programming*, pages 105–129. Palgrave Macmillan, 2003.

[19] Dhanji R. Prasanna. *Dependency Injection*. Manning Publications Co., 2009.

[20] Markus Voelter and Konstantin Solomatov. Language modularization and composition with projectional language workbenches illustrated with MPS. In Mark van den Brand, Brian Malloy, and Steffen Staab, editors, *Software Language Engineering, Third International Conference, SLE 2010*, Lecture Notes in Computer Science. Springer, 2010.

[21] P. Wadler. A prettier printer. In J. Gibbons and O. de Moor, editors, *The Fun of Programming*, pages 223–243. Palgrave Macmillan, 2003.

[22] Niklaus Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Commun. ACM*, 20:822–823, November 1977.