Comparing Language Workbenches

Roman Stoffel
University of Applied Sciences Rapperswil (HSR), Switzerland

MSE-seminar: Program Analysis and Transformation

December 23, 2010

Abstract

To create a domain specific language basic tools like the parser, compiler or interpreted have to implemented. Additionally a integrated development environment (IDE) for a language is expected today. Therefore tools which allow you to define and implement domain specific languages including the IDE start to show up. In this paper three different approaches to tackle these challenges are introduces and compared.

1 Introduction

Domain specific languages (DSL) have been around for ages. Recently creating and using DSLs has become popular again. Embedded DSLs, also known as internal DSL, are written in a host language. They cleverly use constructs of the host language to create the DSL. External DSLs are not embedded in an existing language. They are separate languages which bring with their own syntax and toolchain. In this paper I take a look at tools to create external DSLs.

There have been tools for creating external DSLs for decades. However, this is not enough. Today developers expect more than just a raw compiler or interpreter. They expect a integrated development environment (IDE) with error checking, code completion, refactoring support and more assisting functionality. This is where language workbenches come into the picture. Language workbenches [20] allow you to create and modify DSLs and also directly use them. It assists you with creating a full-fledged IDE for the created DSLs. In this paper I'm going to compare two language workbenches which follow different approaches to tackle the problem and a tool for domain driven code generation.

2 Criteria For Language Workbenches

In order to compare the different tools, I lay out some basic criteria for a language workbench. It is a collection of features and capabilities which I expect from a development environment and functionality which is required to create DSLs.

Integration with other tools: Programming tools are rarely used standalone and in isolation. Usually a whole collection of tools is used for different purposes. This means that a language workbench should work together with other tools. The most important and widely used tools are version control systems, build systems and testing environment. Therefore, I'm going to compare how the different language workbenches work together with these three categories of tools.

Creating DSLs: An important feature is

support for creating DSLs. There are different aspects to it. You need to be able to define the grammar and the semantic meaning of your DSL. In order execute your DSL a transformation to the language or runtime system is required. Can the tool deal with complex grammars? Are there possibilities to debug the transformation from the source DSL to the target language?

IDE assistance: A modern IDE like Eclipse, IntelliJ IDEA or Visual Studio has tons of features like syntax highlighting, error detection, debugging-support, code-navigation, auto completion and refactoring. It also provides means to discover the language and API of libraries by showing the appropriate documentation and improvements-suggestions. In a language workbench this is even more important. These features should be available for the language workbench itself and for the created language. It should assist the developer with implementing a DSL and should also provide means and tools for implementing IDE support for a DSL.

Combination of DSLs: The central idea of LOP is to have languages which fit the problem domain. But usually software has different concerns and it's a good practice to separate these concerns. For LOP this means that we have different DSLs for the different concerns. As soon as we have multiple little languages we want to combine them. So the language workbench should provide means to combine different existing languages.

Changing DSLs: In modern software development refactoring is a regularly used practice to improve the code quality and keep a system ready for future changes. We've also seen that general purpose languages evolve to adopt the changing environment Favre [7]. However, you it is very hard to change the syntax or behavior of existing language constructs in a language, since that almost certainly breaks existing programs. You only can add new features to to the language. In in a LOP environment the development team has tight control over the used languages. This means that we should have two capabilities. The first is to refactor existing code written in the DSL. The second capability is to change the DSL itself. For example to change the DSL syntax and not break existing code and still keep the semantic meaning of the language.

3 The Tools

In this section I present three tools and give an overview of their functionality.

3.1 Meta Programming System MPS

In 2009 Jetbrains, known for their IDEs like IntelliJ IDEA, RubyMine or Resharper, released their approach to LOP called Meta Programming System (MPS) [10]. The idea for MPS was introduced by Sergey Dmitriev [24] and is heavily influenced by the Intentional Programming approach [3].

The goal of MPS is to provide an environment where you develop and evolve your DSL along with your application. It is not built to create a new language which is then shipped to a third party.

3.1.1 Programs as Models

What makes MPS quite unique is that it stores the programs an object model instead of text. In MPS the storage and the visual representation of a program is separated. This is different to regular programming languages where the storage and visual representation is the same, namely text. First of all we need to understand the motivation behind this approach. This different take on programming tries to avoid the inherent limitations of a text based solution.

In a text-based programming language a great effort is required to parse the source-code. A precise grammar needs to be defined

to describe the language. With this the text is parsed to an abstract syntax tree (AST) which then is processed further. While lots of tools assist you to define a grammar and create parser it is still not an easy task. Defining an grammar is difficult and gets harder as the language evolves. You always have to ensure that new language constructs don't make the grammar ambiguous. Also creating a robust parser is challenging. A parser should be able to provide meaningful error messages. And for IDE support a parser should be able to deal with incomplete and invalid programs [18]. This is still not enough. Today the refactoring support of IDEs is getting more and more sophisticated. Often refactorings do their transformation on the AST. However, since the program is written in text this transformation have to be translated back to text and should preserve the layout of the text [22, 14].

Now in MPS programs are persisted as a model and not as text. This avoids all the issues related to parsing and working with text. Let's explore this approach with examples. Let's say we have an object oriented language in MPS. How does such a model look like? Well in an object oriented language there are constructs like classes, methods, fields, method-calls, program-statements etc. In MPS such constructs are called concept. Let's say we've a car-class. Then an instance of the class-concept, which contains instances of method-concept instance, is stored. Now we can start to imagine how programs are stored. In MPS we don't create grammars; instead we create a meta-model of our language model, hence the name Meta Programming System. This means to define a language we create concepts such as a class-concept, method-concepts etc. We also describe the relations between the different concepts like that a class consists of methods and fields. The layering of software in a base level and a meta level is also known as reflection pattern[5] and pursued by model driven approaches [8].

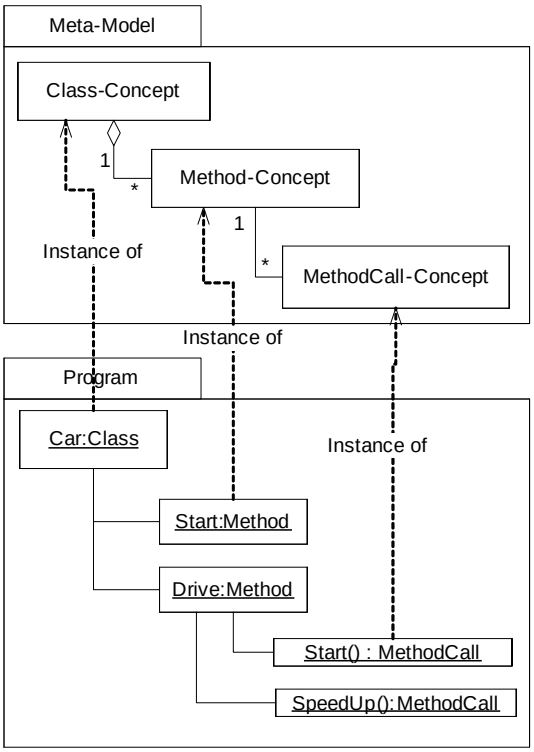

Figure 1: Programs as model

But how can we now write and use the described language? For this we need some kind visualization or editor. That's exactly what you create in MPS. You create one or multiple editors which operate on the model representation of your programs. An editor can be a spread-sheet, a UML-editor, anything which can visualize and manipulate the program-model. Since MPS is focused on programming language the editor usually mimics a text-editor.

All in all we end up with a very sophisticated program modeling system.

3.1.2 MPSs Collection of Languages

MPS brings a whole collection of languages with it, which themselves are developed in MPS [12]. The basic language is called 'Base Language' and is a MPS version of Java. Additionally MPS brings extensions to this Base

Language for different aspects, like DSLs for collections, tuples, closures etc. You can refer to those extensions in your project and then use them. Besides the general purpose Base Language and extensions there are lots of other DSLs. There are languages for defining concepts, a language for describing editors, a language for describing refactorings, a language for describing constrains on concepts and so on. Those languages are used to create DSLs in MPS.

3.1.3 Model to Model Transformations

The compilation process in MPS is a model to model transformation. Your DSL-models are translated into another model, typically into a Base Language model. For this task MPS again provides its own DSL.

As alternative you can transform a model into text, for example, if there's no appropriate MPS language available. And MPS uses the model to text transformation to transform its Base Language to Java, which then is compiled and executed.

3.1.4 Talking To The Outside World

The DSLs you create have to talk to the outside world via other languages and existing libraries. We already have seen that we can transform a model to text if necessary. But we probably want to use libraries in MPS. This means we need to be able to refer to artifacts which live outside of the MPS world. To do this so called 'stubs' are used [13]. Stubs are a representation of artifacts outside of MPS, like existing API's, frameworks etc. For example we in MPS we need stups to use the Java libraries like collections, net-work access etc.

For stubs you first need a MPS representation of your target language / framework. Then you can define a stub-generator in MPS, which transforms the external source-code or API definition into MPS-stubs using the previously declared MPS concepts. Together with the right generators you can then talk to the outside world.

A good example is MPS's Base Language. This language is the door to the Java world. It models a language which is very close to Java. Furthermore there's a model to text transformation for the Base Language to generate Java source code. The final piece is a stub generator which creates stubs for Java-API's. These stubs then allow you to call existing Java libraries.

3.1.5 MPS by Example

Let's take a look at a small example to illustrate the basic concepts of MPS. We create a language to describe a data model. This language consists of simple type definitions, which have a type-name and a list of data fields. A data type which describes a person with first- and surname should look something like this:

```
entity Person {
  firstName : string
  sirname : string
}
```

In MPS we first describe a meta-model of our DSL with MPS-concepts. In our small DSL there are three main concepts. There's the 'entity'-concept which is an abstract data type description with a name and fields. A field has a name and a type. The last concept is a type, which can be a built-in type like string and integer. Additionally a type can refer to an entity. Our model of the language is shown in figure 2. Now these relations are modeled in MPS using the Structure Language. For the entity concept we add a relation the field concept and give it a name property. Because a name is something very common there's a predefined concept 'INamedConcept'. This brings some predefined capabilities with it, like the renaming refactoring. Hence we inherit from that concept to get those features.

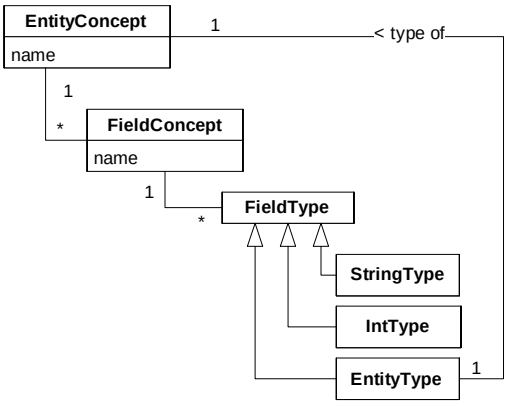

Figure 2: Model of our DSL

```
concept Entity extends     BaseConcept
                implements IResourceContent

  instance can be root: false

  properties:
  << ... >>

  children:
  Field attributes 0..n specializes: <none>

  references:
  << ... >>

  concept properties:
  alias = entity

  concept links:
  << ... >>

  concept property declarations:
  << ... >>

  concept link declarations:
  << ... >>
```

Figure 3: Concepts of our DSL

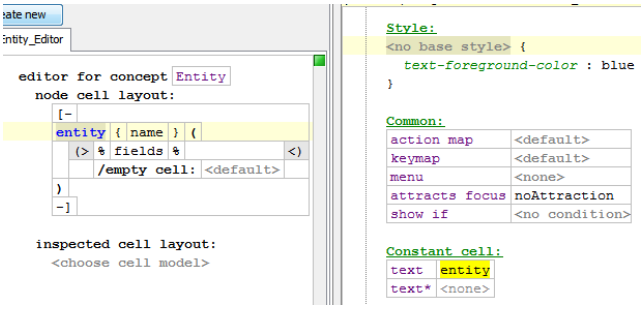

Figure 4: Defining an editor

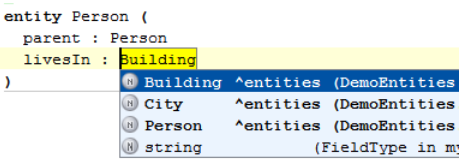

Figure 5: Using the editor

The figure 3 shows the concept declaration for the entity.

After defining the concept we create editors for all concepts, shown in figure 4. In our DSL we need editors for our entity-concept, field-concept etc. These editors define how the DSL is displayed and edited. The editor is described as a collection of cells. In each cell has content, like lables, text-fields, more cells, lay outing components, cells which iterate over data etc. For each cell we can configure additional behavior and style in a separate window with a CSS-like language. For our DSL we create an editor which first has some label cells which contain with the 'entity'-text, followed by a cell which has a text-field for the entity name. After that we add a cell which iterates over all fields of the entity model. The resulting editor looks is shown in figure 5.

The final step is to create the transformation of our DSL to the Base Language. We do this by writing an example of the desired result first. Then we annotate this result with macros. There are macros like loops, conditionals, replacements etc. For our DSL we start by writing a simple Java bean in the Base Language. Then we start to annotate the different parts of this prototype. We re-

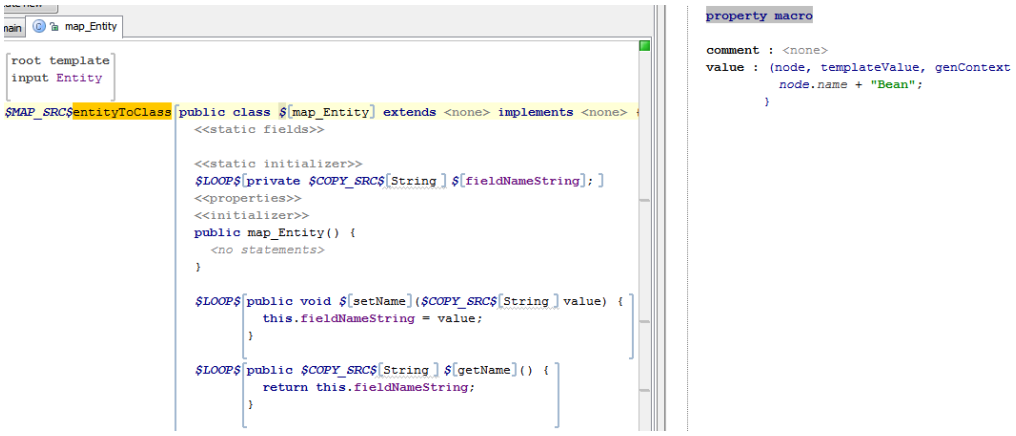

Figure 6: Translate to Base Language

place the class-name with a macro to use the name of the entity. We add a loop-macro to translate fields into getters and setter and so on and so forth. In figure 6 you see the template with the macros on the left. On the right is the behavior which we want to add to the macro.

3.2 Actifsource

Actifsource [1] is a tool to create complex domain models and generate code out of them. It doesn't allow you to create a new language, so it isn't a really language workbench. Therefore it cannot directly be compared with other language workbenches. However, its model based approach shares some similarities MPS. In this paper we only take a brief look at Actifsource.

3.2.1 Similarities to MPS

As in MPS you programs are captured in models, which then are translated to a target language. You build a meta-model of your problem domain. For example, when you build a meta-model of a web-service then you describe the building blocks like parameters, service name, etc. Once the meta-model is built, you use the building blocks to model your problem. This model then is translated to the target language, like Java, C#, Ruby etc.

Activsource is a regular Eclipse plug-in. Therefore, it can be used together with all the other features and plug-ins for the Eclipse platform.

3.2.2 Differences to MPS

Now there is a big difference between MPS and Actifsource. In Actifsource you cannot build editors for your meta-model. Instead you use a UML [21]-like language to describe the meta-model and also the model of your application. This means that you cannot create a complete new language, but rather a specialization of the UML model. The limitations make Actifsource also much easier to learn and use. You don't need to create an editor for your meta-model, which can be quite complex. On the other side it restricts you to problems which can be modeled in UML.

The models in Actifsource are translated to the text of the target language. For this a special template language is used.

3.2.3 Different Purpose

Actifsource has a different target audience than language workbenches. It's a modeling

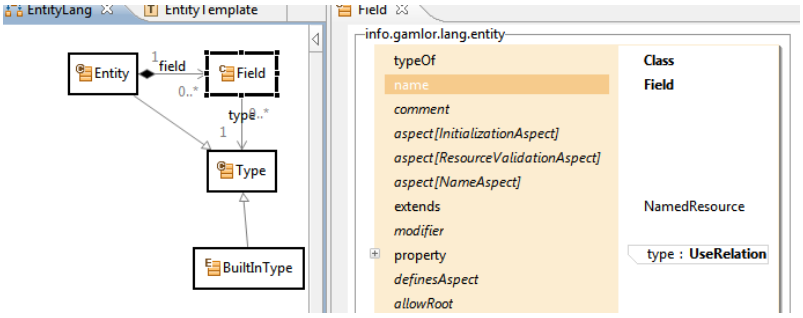

Figure 7: DSL model in Actifsource

```
Entity.nameBean.java
class Entity.nameBean{
   private Field.type:Entity.nameBean Field.name.toFirstLower@BuiltIn;

   public void setField.name.toFirstUpper@BuiltIn(
           Field.type:Entity.nameBean value){
           this.Field.name.toFirstLower@BuiltIn = value;
   }

   public Field.type:Entity.nameBean getField.name.toFirstUpper@BuiltIn(){
        return this.Field.name.toFirstLower@BuiltIn;
        }
```

Figure 8: Translation template

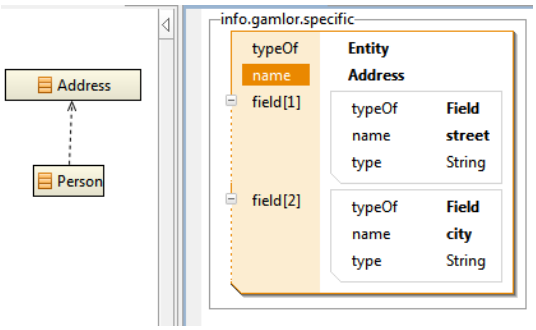

Figure 9: Model in Actifsource

tool, which allows you to model your problems with UML. Such modeling tools can assist you in areas where there's a lot of boiler plate code and where modeling complex relationships is important. However it cannot compete in areas where UML isn't an appropriate language.

3.2.4 Actifsource by Example

We use the small language we used for the MPS example to illustrate the features of Actifsource. We start by implementing the model shown in figure 2. The example implementation is shown in figure 7.

After that we can define the transformation from the model to Java source code. For this the template language is used. It provides access to the model and can generate code accordingly. We build a Java bean line by line and access the model for the information. This way, we can use the entity-name as class name, use the fields for getters and setters and so on. In figure 8 you can see the transformation template for our DSL.

Now everything is ready. We can model our entities with the UML dialect we've created. This model is then translated to regular Java code and ready to be used as shown in figure 9. Then our model is translated to Java source code.

```
class AddressBean{
   private String street;

   public void setStreet( String value){
      this.street = value;
   }

   public String getStreet(){
       return this.street;
   }

   private String city;

   public void setCity( String value){
      this.city = value;
   }

   public String getCity(){
       return this.city;
   }
}
```

3.3 Spoofax

The Spoofax language workbench [15] is based on the popular Eclipse platform. In contrast to MPS it uses a classic text based approach for DSLs. The Spoofax platform provides tooling for defining grammars, transforming the DSL to the target language and providing IDE support for the DSL. You can edit and develop the DSL in the same Eclipse instance, which results in very tight feed-back loop. To define the grammars the Syntax Definition Formalism (SDF) [9, 6] is used. On top of this the Stratego transformation language [19] is used to transform the DSL to the target language. Additionally, there is a special editor language for defining syntax-highlighting, code-outline, code-folding and code-completion.

The Spoofax language workbench takes advantage of the Eclipse IDE Meta-tooling Platform IMP. The final language can be exported as regular Eclipse plug-in. This allows you to distribute the created language to other people.

3.3.1 Syntax Definition Formalism

The SDF-language is a pure and declarative language Heering et al. [6], Kats et al. [16]. Like other grammar definition languages it describes productions rules of the grammar. But it doesn't allow embedding any code or rules with side effects. The SDF-language is backed by a scannerless generalized-LR parser (SGLR) [4]. Unlike other language categories like LL(k), LL(*) or LALR(k) SGLR supports the full range of context free grammars Kats et al. [16], Visser [4]. This has a great advantage. Context free grammars are closed under composition. When you combine two context free grammars the result is still a context free grammar. This isn't the case when you only have a subset of context-free grammars like with LL(k), LL(*) or LALR(k). In practice this means that it is possible to combine different grammars with SDF. You can import and reuse other existing grammars.

Now a context-free grammar may produce an ambiguous parse-tree. In such cases the SGLR parser returns a parse-forest instead of parse-tree. In such cases you can express associativity of productions and prioritize the production rules and try to make the grammar unambiguous. The fact that it SDF doesn't force this with a technical limitation makes the process much easier a cleaner. The rules which make a language unambiguous are not 'hidden', but are clearly stated with annotations.

3.3.2 Transformations as Fundamental Concept

Spoofax uses the Stratego [19] transformation language to describe transformations from the source AST to the final target language. Stratego allows you to write reduction rules that describe transformation steps. Usually such transformations consist of rules which first remove the syntactic sugar from the source language. For example, we transform a high-level foreach loop to a lower level constructs like a while loop. Then other transformation rules describe the transformation from the basic language constructs to the target language.

However, Spoofax uses this transformation process not only for translating the DSL into the target language. Transformations are a fundamental concept of Spoofax. Think about features like error-messages, type lookups or auto-completion. This can be described as special transformation of the source code. For error messages the code is transformed to a list of errors. For type-references the source-code is transformed to list of known types. For auto-completion a list of possible types, fields and methods is generated from the source. This is the fundamental way how you describe and provide complex IDE features in Spoofax.

Only transformations to simple error lists is not enough. For compiler-error messages and IDE features it is extremely vital that the location of the error is known. Spoofax takes care of this. Each node in the AST is associated with the origin position in the source code file. As transformations are applied the position information is preserved. In the error-reporting transformation we just need to return the right node to mark the error location.

A Term Transport Layer: The Stratego transformation system has to consume the output of the parse subsystem. Spoofax uses efficient abstract data types, which are based on annotated terms Van den Brand et al. [17], referred to as ATerms. A textual representation of ATerms is also used to display the AST. However, it is possible to write adapters in Java for other term representations.

3.3.3 Editor Language

Spoofax uses a special editor language do define the features of the generated IDE. In this language you can define IDE features like syntax coloring, points where auto completion is invoked, outline and code-folding properties. Here we can use the lists which are created by transformations as information source. For example, auto-completion uses the list of types which has been generated by a Stratego transformation.

3.3.4 Spoofax by Example

Again we use our small DSL from the previous examples to show the basic concepts of Spoofax. The first thing we do in Spoofax is to write down the grammar. For our grammar we import the 'Common' grammar which contains often used parser rules like whitespace handling, numbers, identifiers etc. In SDF you write first then matching symbols followed by the resulting symbol. In the curly braces you can additional information, in example the name of the AST node.

```
module GamlorEntityLanguage
imports Common
exports
  context-free start-symbols
    Start
  context-free syntax

    EntityDef*  -> Start {cons("Entities")}
    "entity" ID "(" Field* ")"
         -> EntityDef {cons("Entity")}
    ID ":" Type -> Field {cons("Field")}
    ID          -> Type {cons("Type")}
```

After creating the grammar we start to build the compilation process by creating a translation to Java. We create rules which translate each node of the generated AST to a piece of text. The AST-nodes of entities are translated to classes, the fields to getters and setters. In the snippet below we take a 'Field'-AST node and replace it with the given text. Within some parts are replaces with variables.

```
to-java: Field(x, Type(t)) -> $[
  private [t] [x];
  public [t] get_[x] {
    return [x];
  }
  public void set_[x] ([t] [x]) {
    this.[x] = [x];
  }
]
```

The second step is to improve the IDE support for our DSL. Let's take a look at the error detection for our DSL. For example, we should check that a property uses a type which actually exists. For this, we create a list of known entities with the Stratego transformation language. Then we add a rule which checks that the type of a field is either built in or one of the entities.

```
// Create a list of known entities
record-entity:
Entity(x, body) -> Entity(x, body)
with
  // Create a lookup rule
  rules(
    GetEntity :+ x -> x
  )

// Check that a field only uses known types
type-error:
Field(x, Type(type))
   -> (type, $[Type [type] is not defined])
where
  not(!type => "String");
```

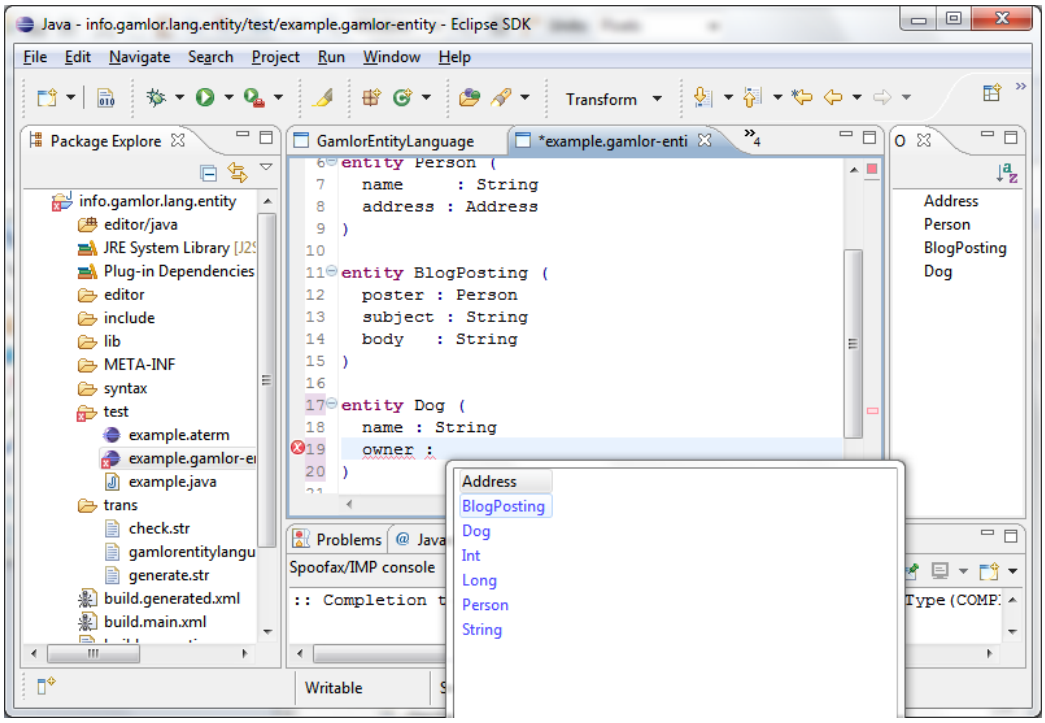

Figure 10: The IDE for our DSL

```
not(!type => "Int");
not(!type => "Long");
not(<GetEntity> type) // lookup type
```

The list of known entities is used for further IDE features, like type-completion:

```
completions
  // Syntax completion:
  completion template
        : "msg " <msg> (blank)
  completion template
       : "entity " <e> " {}" (blank)

  // Semantic (identifier) completion:
  completion proposer
      : editor-complete
  completion trigger    : ":"
```

We can add further features to the IDE support like syntax highlighting, code folding etc. The final result is a nice IDE for our language, as shown in figure 10. As last step we can export our IDE features as Eclipse plug-in and use the DSL for our projects.

4 Comparisons

As discussed there are different approaches for language workbenches. MPS and Spoofax tackle DSL in a very different way. That's why it's interesting to compare these two realizations and find out where their strengths are. Because Actifsource is not a language workbench it is covered more briefly than the other two tools.

4.1 Integration With Other Tools

In this section compare how the three tools are integrated with other tools.

4.1.1 Integration with Text Based Tools

Here MPS has serious short comings due to its model-based nature. Text-based tools don't work for MPS. MPS uses XML as storage format, but the stored XML documents are very technical and not 'human' readable. This

means that the whole text-based tool chain programmers are used to doesn't work with MPS. Diff and merge tools, text-editors and other text-tools only show the raw XML. MPS supports popular version-control systems and brings its own merge/change/diff-view which shows the changes in the right format. As long as you use a version control system supported by MPS the situation acceptable. But even for some more complex changes MPS only shows the underlying XML-storage instead of the regular MPS views. The issues go further. Even regular copy and paste functions don't work. You cannot copy a code-snippet from a text-source like a tutorial into MPS. Because MPS cannot deal with text, it needs a complete model. This situation is a bit improved in the upcoming MPS 2.0 [11].

The situation in Actifsource looks very similar to MPS. It stores its models in XML-files as well and brings its own diff tools for version control systems. Since Actifsource is not a replacement for regular programming language, but rather a helpful addition the situation is not as bad.

In Spoofax all text-based tools work just fine because it is all text based. For version control Spoofax even has a clear naming scheme to distinguish generated files from hand edited files. All files which have '.generated.' in the name shouldn't be checked in to a version contol system.

4.1.2 Integration with Continuous Build Tools

MPS provides a way to build the languages and projects with Apache Ant. So it is possible to build it with most continuous build tools for Java. MPS actually brings its own DSL to describe the build process. This is also based on Ant but provides a nicer syntax. MPS creates a bootstrap Ant build-script which then calls into the scripts written in the DSL. However, the default generated build script contains hard references to the MPS-system, which could be different on a build-server than on a developers machine

ctifsource also brings Apache Ant tasks with it. This allows you to run the Actifsource model to code generator during the build.

Spoofax also uses Apache Ant as build tool and a Spoofax project can be build on most continuous build systems. Unlike MPS it doesn't bring its any DSL to describe the build.

4.1.3 Unit-Testing

Once again MPS brings its own testing DSL with it. This DSL is based on the popular JUnit testing framework. It contains special support for instantiating parts of the model under tests. It allows you to use your DSL in the test-case ad write a small snippet in your DSL. In the test you then get the model-instance where you can check if the model behaves as expected. For example to test a math DSL you write a snippet in the math DSL. Then you get the model instance in the test to perform tests on it.

Actifsource and Spoofax don't have special support for testing. Of course you can use the regular tooling like JUnit and Eclipse plug-ins to create test cases.

4.1.4 Integration with Environment

MPS doesn't have any special integration with other tools. While it is based on the IntelliJ IDEA platform it doesn't provide any export mechanism to that IDE. It can use some IDEA plug-ins, but by no means all IntelliJ IDEA plug-ins. Therefore MPS is pretty isolated.

In contrast Actifsource and Spoofax are regular Eclipse plug-ins and can be used together with other plug-ins. Furthermore, Spoofax allows exporting the created languages as stand-alone Eclipse plug-in.

4.2 Creating DSLs

4.2.1 Defining Grammars and Complex Languages

Both, MPS and Spoofax, allow you to define fairly complex languages. In Spoofax you can create arbitrary context-free languages. In MPS you can create complex meta-models and editors to build languages.

Actifsource is not a language workbench, hence it cannot be compared with the other two tools.

4.2.2 Transformation

MPS, Actifsource and Spoofax use a transformation language to translate the source DSL to the target language. All three tools use a similar approach where you can write a template and then replace the right parts of the template according to the source material. Spoofax and Actifsource translate the output to text. MPS transforms DSL-models to another program-models, although model to text transformations are also supported.

4.2.3 Debugging

MPS brings debugging support for its Base Language and the extensions to that language. It is possible to extend the debugging support for a DSL as long as it is based on the Base Language. However, for DSLs not integrated into the Base Language no debugger is available. You cannot even step through the final resulting Java code. For that you need to use an external IDE like Eclipse or IntelliJ IDEA.

Actifsource doesn't have a model debugger, but you can use the regular Eclipse debugger to debug the generated code.

Spoofax itself doesn't bring any direct debugger support for generated DSLs. However, since it's an Eclipse plug-in you can use the regular Eclipse debugger to debug the generated code. And it is possible to use the underlying Eclipse infrastructure to implement the debugger support separately.

4.3 IDE assistance

Let's take a look at the IDE support of the three tools.

4.3.1 Syntax Highlighting, Code Folding

MPS and Spoofax provide good support to implement syntax highlighting and code folding. In MPS this is done by changing the style of elements in the editor of a model. For this a CSS-like language is used to define the style of the different elements. In Spoofax you can define syntax highlighting and code folding in the editor language. By default, certain elements like terminal symbols are already colored appropriate. You then can give colors of other nodes-types in the AST.

Although Actifsource isn't a language workbench it still brings limited support for syntax highlighting. This is useful when you translate models into a certain programming language. Then the syntax highlighting helps to read the generated code.

4.3.2 Navigation & Discoverability

In MPS code-navigation works extremely well for the MPS languages and for created DSLs. MPS always knows to what you're referring to in your language. You don't need to do any additional work. This way you can navigate along definitions of any symbols, types etc. You can also find all locations where a symbol is used. Additionally you always can open up the concept of the current element to find out how something is defined. This is also useful to learn MPS itself, since you can take a look at the implementation of the MPS languages.

Actifsource brings also excellent support for navigating along the models. You can find

the definitions, references, subtypes etc. for any artifact in the model.

In Spoofax code navigation and discoverability are very different for the Spoofax languages, SDF and Stratego, and DSLs you create yourself. For the Spoofax languages there are limited navigation features available. You can navigate to the definition of transformation-rules and grammar-productions. But advanced navigation features like finding all usage locations of a symbols are not supported. For such cases you are stuck with text-searching for symbols and definitions across the different code artifacts. For the DSL you easily can define look-up tables for symbols and add provide navigation support, although some work is required to add this features for your DSL.

4.3.3 Error-Detection

In MPS typical errors are prohibited by the model approach. You cannot enter DSL code not defined by the model. Additionally, you can define constrains on the model that are validated at runtime. For complex error detection you can add type-system definitions where you can manually program validations of the models.

Actifsource also catches most errors because you cannot write anything which isn't defined by the meta-model. Besides that you can add additional validation rules, written in Java.

In Spoofax there are two types of error detection. All syntactic errors are checked by Spoofax and marked in the IDE. For semantic errors you need to define transformation rules which return the appropriate errors.

4.3.4 Auto-Completion

Again MPS benefits from its model based programming that it automatically can infer a lot of information and bring the appropriate auto-completion suggestions. Also constrains on the model are honored in the suggestions. To polish the editor you can define 'intentions' to your model. Intentions are suggestions, transformations, refactorings, etc, appropriate for current code.

The same applies to Actifsource autocompletion; it can give you useful suggestion due to the model restrictions. You can refine the auto completion with additional rules.

Like before, in Spoofax you use Stratego to transform the AST to look-up tables. These lookup tables are then used to provide appropriate auto completion at the right locations.

4.3.5 Refactoring Support

In MPS basic refactorings like renaming, move and safe deletion are implemented automatically. For MPS languages more advanced refactorings are available. Additional refactorings for your DSL can be defined with a special refactoring language to describe the transformations on the model.

Actifsoure brings some basic refactorings like renaming with it. However you cannot create more complex refactorings yourself.

At the moment Spoofax doesn't provide any direct support for refactoring. The Spoofax team hopes that refactorings can be described declaratively in the future [15]. But it isn't implement right now. Of course, you can fall back to the Eclipse platform to implement refactorings for your DSL in Java with the Eclipse APIs.

4.4 Combination of DSLs

MPS strongly encourages to create many small DSLs and the combine and reuse DSLs. For example, MPS provides a collection of extensions to the Base Language which can be optionally included. Also all MPS specific languages like those for concepts, constrains, tranformations often resue parts of the same language.

To support this kind of combination of

DSLs MPS strongly relies on the fact that everything is described with MPS concepts and that there's no parsing required. It also relies on the fact that transformations are usually model to model transformations which end up in a Base Language model. If the different DSL are translated to different models, let's say to Java and Javascript, then the combinations of DSL is not possible without redefining one of the transformations.

Actifsource also allows you to refer to existing models and reuse parts of it, similar to MPS.

In Spoofax allows you to use multiple DSL side by side. You can create two separates DSLs and the export them as Eclipse plug-in. Then you can install both plug-in and use both languages in your project. Also the SDF language is modular which makes it possible to reuse syntax definition Kats et al. [16]. However, I didn't find a mechanism to import or refer to existing DSLs and reuse their implementation. I also belief that it's hard to combine and reuse the transformation and the editor definitions.

4.5 Changing the DSL

In MPS you can freely change the representation of the DSL, without destroying existing code base. This is possible because in this case you only change the visual representation of the language but not the semantic meaning of it. Some simple refactorings like moving a property of a concept to a parent concept also work without breaking existing code. But more complex changes will break existing programs.

Actifsource doesn't support major metamodel changes. When you change the metamodel you may break DSLs.

In Spoofax there is no refactoring available. Like other text-based DSLs it's hardto change the syntax or semantic meaning of the language without breaking the existing code.

4.6 Overview & Recommendations

The overview of the strengths and weaknesses of the three compared tools are listed in table 1.

Also I want to close with recommendations to pick the right tool for the right job.

MPS is a great tool when you want to create a collection of DSLs, which then are combined to implement a complex system. Additionally the DSL code should live relatively isolated from the rest of the system, because it only can be edited in MPS. In such cases MPSs excellent support for complex languages and IDE support can unleash the full power of MPS. Where MPS is not usable is in scenarios where you develop a DSL and then distribute it as plug-in/system to the developers. MPS is a model based island, which is hard to combine with other text based systems.

Actifsource is great tool when you need to model complex systems and UML is an appropriate model language. In such cases it is a great addition that works together with other programming languages.

Spoofaxs strength is to build text based DSLs for Eclipse. These DSLs then can be used together with general purpose languages in the same project. Spoofax is also optimal when you want to distribute your DSL in form of an Eclipse plug-in.

5 Related Work

There have already been comparisons of language factories by Tony Clark and Laurence Tratt [25]. They focused on a broader spectrum of ways to create DSLs, particular languages which encourage building internal DSLs.

Another comparison of different language workbenches was made by Bernhard Merkle [2], who analyzed a broad spectrum of tools.

It is also worth to mention that currently a language workbench competition [23] is pre-

	MPS	Spoofax	Actifsource
Integration With Text Based Tools	–	+	0
Integration With Continious Build Tools	0	0	0
Unit-Testing	+	0	0
Integration With Environment	-	++	++
Defining Grammars And Complex Languages	++	++	N/A
Transformation	++	++	+
Debugging	+	0	0
Syntax Highlighning, Code Folding	++	++	N/A
Navigation & Discoverability	++	0	+
Error-Detection	++	+	+
Auto-Completion	++	+	++
Refactoring Support	+	0	0
Combination of DSLs	++	+	N/A
Refactoring the DSL	+	0	N/A

N/A	Actifsource is not a language workbench, therefore it can only be compared partially
++	Excellent support for area/features
+	Good support for this area/features
0	No particular support for this area/features
-	Bad support for this area/features
–	Very bad support for this eare/features

Table 1: Strengh and Weaknesses of Spoofax, MPS and Actifsource

pared for the Code Generation 2011 conference. There, different language workbenches provide a implementation of the same task. The goal is to be able to compare strengths and weaknesses.

6 Conclusions

In this paper I've compared three different language workbench implementations, MPS and Actifsource which are model-based and Spoofax which creates regular textual languages. All tree solutions provide a convenient way to define and implement a DSL togehter with IDE support.

MPS gains from its model-based approach superior IDE support, but it suffers heavily that it cannot integrate to the text-based mainstream tooling. If you can live with a relatively isolated system which provides great IDE support for your DSLs you should take a look at MPS.

Actifsource is a tool which can help modeling complex systems with UML models.

Spoofaxs traditional approach integrates well with other text-based tools and benefits from the Eclipse platform. However it currently cannot provide the same level of IDE support for its own DSL and the user defined DSLs. If you're interested in creating DSLs and want to take advantage of existing technologies the more pragmatic text based approach is probably a better solution. Take a look at Spoofax or similar tools.

.

References

[1] actifsource GmbH [2010], 'Actifsource'.
URL: *http://www.actifsource.com/* (accessed 15th December 2010)

[2] Bernhard Merkle [2010], Textual modeling tools: overview and comparison of language workbenches, *in* 'Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion', SPLASH '10, ACM, New York, NY, USA, pp. 139–148.
URL: *http://doi.acm.org/10.1145/1869542.1869564*

[3] Charles Simonyi [1995], 'The death of computer languages,the birth of intentional programming'.
URL: *ftp://ftp.research.microsoft.com/pub/tr/tr-95-52.doc*

[4] E. Visser [1997], Scannerless generalized-lr parsing, Technical report.
URL: *http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.7828*

[5] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal [1996], *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*, 1 edn, Wiley.
URL: *http://www.worldcat.org/isbn/0471958697*

[6] J. Heering, P. R. H. Hendriks, P. Klint and J. Rekers [1989], 'The syntax definition formalism sdf, reference manual', *SIGPLAN Not.* **24**, 43–75.
URL: *http://doi.acm.org/10.1145/71605.71607*

[7] J.-M. Favre [2005], Languages evolve too! changing the software time scale, *in* 'Principles of Software Evolution, Eighth International Workshop on', pp. 33 – 42.
URL: *http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=01572304*

[8] Jean Bézivin [2007], On the Unification Power of Models, *in* 'Software and System Modeling'.
URL: *http://www.sciences.univ-nantes.fr/lina/atl/www/papers/OnTheUnificationPowerOfModels.pdf*

[9] Jeroen Scheerder Joost Visser [2000], 'A quick introduction to sdf'.
URL: *ftp://ftp.stratego-language.org/pub/stratego/docs/sdfintro.pdf*

[10] JetBrains [2010*a*], 'Jetbrains MPS'.
URL: *http://www.jetbrains.com/mps* (accessed 20th November 2010)

[11] JetBrains [2010*b*], 'Jetbrains MPS 2.0 M1 new features'.
URL: *http://confluence.jetbrains.net/display/MPS/What's+new+in+MPS+2.0+M1* (accessed 20th October 2010)

[12] JetBrains [2010*c*], 'MPS 1.5 user guide, section platform languages'.
URL: *http://confluence.jetbrains.net/display/MPSD1/MPS+User's+Guide* (accessed 27th November 2010)

[13] JetBrains [2010*d*], 'MPS user guide, stubs'.
URL: *http://confluence.jetbrains.net/display/MPSD1/Stubs* (accessed 27th November 2010)

[14] Kristoffer Norling Mikael Blom Peter Fritzson Adrian Pop [2008], 'Comment- and indentation preserving refactoring and unparsing for modelica'.
URL: *https://www.modelica.org/events/modelica2008/Proceedings/sessions/session6a3.pdf*

[15] Lennart C. L. Kats and Eelco Visser [2010], The Spoofax language workbench. Rules for declarative specification of languages and IDEs, *in* M. Rinard, ed., 'Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno, NV, USA', pp. 444–463.
URL: *http://researchr.org/publication/KatsVisser2010*

[16] Lennart C. L. Kats, Eelco Visser and Guido Wachsmuth [2010], Pure and declarative syntax definition: Paradise lost and regained, *in* 'Proceedings of Onward! 2010', ACM.
URL: *http://www.lclnet.nl/publications/pure-and-declarative-syntax-definition.pdf*

[17] M. G. T. Van den Brand, H. A. de Jong, P. Klint and P. A. Olivier [2000], 'Efficient annotated terms', *Softw. Pract. Exper.* **30**, 259–291.
URL: *http://portal.acm.org/citation.cfm?id=343460.343468*

[18] Maartje de Jonge, Emma Nilsson-Nyman, Lennart C. L. Kats and Eelco Visser [2009], Natural and flexible error recovery for generated parsers, *in* M. G. J. van den Brand and J. Gray, eds, 'Software Language Engineering (SLE 2009)', Lecture Notes in Computer Science, Springer, Heidelberg.
URL: *http://swerl.tudelft.nl/twiki/pub/Main/TechnicalReports/TUD-SERG-2009-024.pdf*

[19] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas and Eelco Visser [2008], 'Stratego/XT 0.17. A language and toolset for program transformation', *Science of Computer Programming* **72**(1-2), 52–70. Special issue on experimental software and toolkits.
URL: *http://swerl.tudelft.nl/twiki/pub/Main/TechnicalReports/TUD-SERG-2008-011.pdf*

[20] Martin Fowler [2005], 'Language workbenches: The killer-app for domain specific languages?'.
URL: *http://www.martinfowler.com/articles/languageWorkbench.html* (accessed 22th December 2010)

[21] OMG [2010], 'Uml specification'.
URL: *http://www.omg.org/spec/UML/* (accessed 15th December 2010)

[22] Peter Sommerlad, Guido Zgraggen, Thomas Corbat and Lukas Felber [2008], Retaining comments when refactoring code, *in* 'Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications', OOPSLA Companion '08, ACM, New York, NY, USA, pp. 653–662.
URL: *http://doi.acm.org/10.1145/1449814.1449817*

[23] S. Kelly A. Hulshout J. Warmer P. J. Molina B. Merkle K. Thoms M. Völter E.Visser [2010], 'Language workbench competition 2011'.
URL: *http://www.languageworkbenches.net/* (accessed 16th December 2010)

[24] Sergey Dmitriev [2004], 'Language oriented programming: The next programming paradigm'.
URL: *http://www.jetbrains.com/mps/docs/Language_Oriented_Programming.pdf*

[25] Tony Clark and Laurence Tratt [2009], Language factories, *in* 'Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications', OOPSLA '09, ACM, New York, NY, USA, pp. 949–955.
URL: *http://doi.acm.org/10.1145/1639950.1640062*