CYBERNETICA
Institute of Information Security

Simplicitas: Software Architecture Document

Margus Freudenthal

T-4-9 / 2010

Copyright ©2010
Margus Freudenthal.
AS Cybernetica, Institute of Information Security

The research reported here was supported by:

1. Estonian Science foundation, grant(s) No. 6944,
2. the target funded theme SF0012708s06 “Theoretical and Practical Security of Heterogenous Information Systems”,
3. the European Regional Development Fund through the Estonian Center of Excellence in Computer Science, EXCS, and the Software Technology and Applications Competence Centre, STACC,

All rights reserved. The reproduction of all or part of this work is permitted for educational or research use on condition that this copyright notice is included in any copy.

Cybernetica research reports are available online at
`http://research.cyber.ee/`

Mailing address:
AS Cybernetica
Akadeemia tee 21
12618 Tallinn
Estonia

Simplicitas: Software Architecture Document

Margus Freudenthal

Version 1.0

Contents

1 Introduction **5**

1.1 Background 5

1.2 Some Terminology 5

2 Requirements **5**

2.1 Environment 5

2.2 Output of the Project 6

2.3 Architecturally Significant Requirements 9

3 Candidate Architecture **10**

3.1 Description of Candidates 10

3.1.1 Xtext-Based Solution 11

3.1.2 Stratego-Based Solution 12

3.1.3 ANTLR-Based Solution 14

3.2 Work to Be Done 16

3.2.1 Xtext 17

3.2.2 Stratego 18

3.2.3 ANTLR 18

3.3 Evaluation 19

4 Technical Issues 20
4.1 Abstract Representation of DSL Program 20
4.2 API for Language Services 20
4.3 Linking 21
4.4 AST Post-Processing 21
4.5 Validation Framework 23

5 Third-Party Software 24

6 Conclusion 24

1 Introduction

1.1 Background

Domain-specific languages (DSL) are proven to be useful in raising the level of abstraction when developing complex software systems. Language-oriented programming (LOP) is an approach to software development, which aims at separating the problem-specific and technology-specific parts of the system and providing a set of DSLs for describing the functionality of the system.

Although there exist several tools for supporting various aspects of DSL construction, there are no turn-key solutions for applying LOP in enterprise information system development. This creates quite a high entry barrier for organization wishing to make extensive use of DSLs.

Simplicitas is a sub-project carried out by Software Technology and Applications Competence Centre[1]. The goal of the Simplicitas project is to develop a resonably comprehensive toolset for creating DSLs in an enterprise context.

This project will leverage existing open source tools to minimize the amount of work that needs to be done. The work will mainly consist of selecting the existing tools, integrating them and adding additional features.

1.2 Some Terminology

DSL tool – a software component that is targeted at creating DSLs. Examples include parser generators, templating engines, IDE generators etc.

DSL toolkit (toolset) – set of DSL tools that covers most (preferably all) aspects of creating a DSL.

System under development (SUD) – enterprise information system (EIS) which is partially written using DSLs.

2 Requirements

2.1 Environment

This project targets organizations that develop Java-based information systems. To make matters more concrete, we can assume web-based user inter-

[1] See http://www.stacc.ee/

faces and use of J2EE standards, but these are not necessary. The information systems implement complex processing logic and there is need to make it more compact/readable with possibility of letting non-technical people express the logic directly in formal language.

The organization also uses elements of product line architecture – it creates systems that have similar functionality or that are based on common architecture. This makes the case for DSLs more compelling because any investments in DSL technologies can be paid back in reduced costs over creating several systems.

In Cybernetica, examples for these kinds of systems/projects are Customs Engine (a set of systems for processing various customs documents) and cyberDiagnostics (medical laboratory and radiology information system).

Published reports on using DSLs for enterprise software development generally emphasise building complex DSLs for very complex systems. The systems under development often are based on product line approach. It seems that these DSLs are generally implemented via complex hand-written code generators.

2.2 Output of the Project

The output of this project is a toolset for creating implementations of textual DSLs[2].

Must-have requirements:

1. The toolset must be open source and free (as in free beer).
   This is becoming industry standard and it is very hard to attract some users to toolkit that does not conform to this requirement.

2. The toolset must support compiling DSL programs to Java.
   This follows directly from section 2.1.

3. The toolset must support compiling DSL programs to various output formats.
   This means that the DSL can be used to generate XML, build files, etc.

[2]The focus is on textual DSLs as opposed to graphical DSLs. I believe that textual DSLs have lower entry barrier because they are more familiar to programmers who are used to textual programming languages and can be managed and version controlled much like ordinary programs (as opposed to potentially complicated repository management usually involved when using graphical DSLs).

4. The toolset must support calling the DSL compiler from the build system.
   Note: the compiler must be fast to support re-generating the code on each build.

5. The toolset must support calling the DSL compiler from the SUD.
   In this case, the DSL programs are compiled by loading them into the SUD.

6. The code generated from the DSL compiler must not have restrictions that prevent it from being integral part of the SUD.
   Just in case...

7. The toolset must support creating IDE for the DSL.
   Nowadays, IDE support is considered to be default feature of a programming language. Having a reasonably friendly IDE is necessary for acceptance of a DSL (both by non-technical users and developers).

   (a) Preferably the IDE should be based on Eclipse that is becoming the industry standard Java IDE.

   (b) The IDE generator should support the followig features (in decreasing order of importance):

      i. syntax colouring;
      ii. real-time error reporting;
      iii. content assistance (autocomplete);
      iv. reference navigation[3];
      v. documentation tooltip;
      vi. locating usages of declared name;
      vii. support for simple refactorings (e.g. renaming a variable).

8. The toolset must support distributing the DSL program over several files.
   Although the program is spread over files, the DSL compiler must be able to process the whole program in one piece. Also, features such as reference resolving must work between source files.

9. The toolset must include documentation generator, similar to JavaDoc. The DSL developer can include the documentation comments in the grammar file and the documentation generator outputs annotated grammar, basically the language reference.

[3] User clicks on a name and navigates to definition of this name

10. The toolset must make it easy to create simple DSLs.
By simple, I mean DSLs that do not have complicated internal structure and that have one-to-one mapping from DSL constructs to generated Java constructs. An example of simple DSL is state machine example in [1]. Creating a full implementation of this example DSL should not take more than half a day by a skilled developer.

11. The toolset must make it possible to create complex DSLs.
Complex DSL is a DSL that has complicated grammar/internal structure. DSL constructs do not have direct mapping to Java constructs. Compilation of a complex DSL is usually done in several steps which transform abstract representation of a program into another abstract representation that is more closer to the target language. This final abstract representation is then transformed into target language (e.g. Java).

Should-have requirements:

1. The toolset should provide support for unix-based editors (e.g. vim). The support can include:
   (a) generating vim syntax files from grammar definitions;
   (b) generating tag files from DSL programs.

2. The toolset should provide support for debugging DSL programs.
However, it is a bit unclear how precisely this support should be manifested. This can mean several things.
   - *Include source information in the generated JVM bytecode so that the compiled DSL program can be debugged using standard source-level debugger. (TODO: do the standard IDEs like Eclipse support debugging of non-Java programs?)*
   - *Establish some sort of traceability between DSL program and the generated Java code. Use this information when debugging the generated code (e.g. running the code in debugger or looking at exception stack traces). Another possibility is viewing DSL program and generated program side by side and jumping between the two views. This could help understanding what exactly is generated from the DSL program and what parts of the DSL program contribute to given generated code construct.*

3. The toolset should support separate compilation of the DSL programs.
This mostly means that if DSL program consists of several files and some files are compiled separately, then some features, such as reference resolving must still work.

4. The toolset should support generating a grammar documentation.
Based on the annotated grammar description, the toolset should generate documentation that contains e.g. annotated BNF of the DSL's grammar.

Nice-to-have requirements:

1. The toolset can support creating interpreter for the DSL. The interpreter must be native Java component.
However, it is a bit unclear what the support for interpreter should look like. Should it include support for bytecode generation and interpretation? Or should it just be some library that simplifies walking the AST?

2. If the DSL is implemented as interpreter, then it must be possible to call the interpreter both from the build system and from the SUD.

3. The toolset can support creating a family of DSLs that share similar syntax or language constructs (e.g. use common expression sublanguage).
The idea is that if the software architect designs several DSLs for describing various aspects of the SUD, then it is desirable that these DSLs resemble each other both syntactically and semantically. Also, this resemblance should be achieved by code reuse where parts of the syntax definition are reused from one DSL to another.

2.3 Architecturally Significant Requirements

The main requirements driving the architecture of the DSL toolkit are the following.

- The DSL toolkit should be based on existing tools, as much as possible.
- It must be possible to integrate the DSL implementation into Java-based software.

- It must be easy to create implementation for a simple DSL. For example, parser, IDE and code generator for state machine example in [1] should take no more than half a day, ideally couple of hours.
- It must be possible to create implementation for a complex DSL[4].
- The toolkit must be easy to learn and use for "architect level" developers. That means, developers who have previous experience with several frameworks and programming languages.

3 Candidate Architecture

3.1 Description of Candidates

This section presents candidate architectures. Each candidate architecture is based on one or several off-the-shelf components. The amount of additional work needed differs between candidates. Because the base components will determine the general characteristics of the composite DSL toolkit, the selection of base technologies is very important.

Each candidate architecture contains components with the following functions:

- representation of the AST (or model, in MDD terms);
- parser generator (generates component for T2M transformation);
- template engine (component for M2T transformation);
- IDE generator (generates text editor with the usual IDE features);
- M2M engine (used for multi-step compilation of complex DSLs);
- validator (checks structural and type correctness of DSL programs);
- builder (invokes various transformations in the correct order).

[4]Complex DSL is a DSL that has complicated grammar/internal structure. DSL constructs do not have direct mapping to Java constructs. Compilation of a complex DSL is usually done in several steps which transform abstract representation of a program into another abstract representation that is more closer to the target language. This final abstract representation is then transformed into target language (e.g. Java).

3.1.1 Xtext-Based Solution

Toolkit will be based on Eclipse Xtext[5], licenced under Eclipse Public Licence. Xtext is based on the Eclipse modelling tools and offers good usability for creating simple DSLs.

The solution would consist of the following parts.

- AST representation: Ecore (format used in Eclipse modelling tools) model. Can be derived from the grammar description.
- Parser generator: Xtext uses ANTLR-based parser generator. Grammar description is based on ANTLR syntax that is annotated with information needed to generate the Ecore metamodel.
- Templating engine: Xpand. It is an interpreted templating engine that is used mostly with Xtext. It offers quite typical set of features and can call functions written in Xtend language (see next bullet point).
- M2M engine: the "orthodox" solutions would be ATL and QVT which are model-to-model transformation languages with implementations on the Eclipse platform. However, they are both quite inconvenient to use in practice and at least QVT implementation cannot be used outside the Eclipse IDE. The Xtext package also includes Xtend language that can be used to process models. It is interpreted, Java-like language with some additional features (multiple dispatch and first-class functions).
- IDE generator: Xtext includes IDE generator that seems to work quite well and offers good functionality.
- Validator: Xtext offers API for writing validation functions in Java.

Overall

Xtext offers good functionality and ease of use out of the box. It is easy to create a simple language containing parser, IDE and code generator. However, it is integrated with the Eclipse platform and it may not be the best choice for creating complex DSLs.

Pros

- Provides nice, usable IDE.

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

- Easy to create simple languages.
- Official part of Eclipse platform.
- Actively developed (at least, actively promoted).

Cons

- Ecore metamodel is a bit overweight, if one wants to use and modify it from the Java code.
- It seems that Xtext is not a very good solution for implementing complex DSLs. The general mode of operation is processing each DSL file in isolation. There seems to be no support for processing set of DSL files together[6]. Implementing complex DSLs will result in programming Java against quite heavyweight APIs. The built-in link resolving only works inside one DSL file and changing this behaviour seems to be very difficult.
- Xtext seems to rely on Modelling Workflow Engine (MWE) which is essentially an XML-based scripting language for invoking the components in the correct order.
- Xtext is based on interpreted languages MWE, Xpand and Xtend. The errors in programs are only reported at run time. Also, this combination is also not very fast. I have not done extensive performance testing, but I suspect that recompiling DSL programs at each build can get time-consuming.
- Xtext is quite dependent on the Eclipse IDE. There is maven plugin for compiling Xtext project without Eclipse, but this requires downloading 40MB of dependencies. The generated language implementation will probably have less dependencies, but this can still make embedding DSL implementation into application runtime difficult.

3.1.2 Stratego-Based Solution

Toolkit will be based on the Stratego/Xt[7] program transformation system and associated tools (licenced under LGPL). Stratego is a language that is

[6]There is a way to import elements from other files, but this means that one must explicitly create some "master" file and including import statements for all the elements in all the DSL files.

[7]http://strategoxt.org/

mainly targeted at rewriting programs, e.g. for adding additional language constructs, optimising and refactoring.

The solution would consist of the following parts.

- AST representation: ATerm (Stratego native representation for syntax trees).
- Parser generator: SDF (parser generator used by Stratego).
- Templating engine: Stratego does not contain a traditional templating engine. However, it does contain mechanism for turning AST back to text ("unparsing") and pretty-printing the result. The unparsing support seems to assume 1:1 relationship between the AST and the output (it really is the reverse of parsing). Therefore, the default approach used by Stratego assumes that the DSL program is transformed to AST that closely resembles output and then transformed to text.

  Stratego comes with *java-front* that contains AST definitions and pretty-printer for the Java language. *java-front* also allows embedding Java code in Stratego programs (the embedded code is syntax-checked at the compile time). In practice, however, the use of embedded Java code is quite cumbersome if the code is composed of many distinct pieces and other data (such as class and variable names). Using this method requires knowing exactly how the AST of various Java code snippets will look like and ensure that these AST pieces will combine into well-formed Java AST.

  Another option is to use string quotation syntax that will be part of the Stratego/Xt 0.18 release. The syntax is similar to e.g. Ruby strings. This will lose the compile-time syntax checking, but is more convenient because various issues with composing AST parts will disappear.
- Validator: Stratego code.
- IDE builder: Spoofax/IMP. It is based on IMP[8] toolkit that is interfaced with Stratego code so that it is possible to program the IMP services in Stratego.

Overall

Stratego toolkit is created by smart people and is specifically targeted for implementing language processors. The IDE part is not very refined though

[8] http://www.eclipse.org/imp/

and implementing simple DSLs in Stratego seems to be quite a bit of work (however, this can probably be reduced by creating suitable wizards or libraries).

The Stratego compiler is a native executable and by default produces native executables. This is a bit problematic because it requires DSL developers to have full Stratego installation. Having DSL implementation in executable form makes it more difficult to include DSL compiler into running application. The solution is to use Stratego/J that is a Java implementation of the Stratego virtual machine and can run Stratego programs that are compiled to bytecode.

Pros

- The language implementation part does not depend on user interface components.
- Powerful language for program transformations.

Cons

- Stratego language is quite different from usual programming languages. Complicated Stratego programs can get messy, although Stratego-Tiger and WebDSL demonstrate that creating complex Stratego programs is possible.
- Large parts of the standard library are undocumented.
- Windows port of the Stratego exists, but it seems that it does not have high priority.
- The Java implementation of Stratego is not very mature and does not offer good performance (order of magnitude slower than the native implementation).
- The IDE builder is not as polished as e.g. Xtext.

3.1.3 ANTLR-Based Solution

The toolkit will be based on the ANTLR[9] parser generator (licenced under BSD licence) and IMP IDE toolkit (EPL licence). Program validation and

[9] http://www.antlr.org/

transformation is done using some high-level programming language (Java does not qualify as a high-level language). Reasonable candidates are Yeti and Scala.

Yeti[10] is a language very similar to OCaml and other ML-style languages. In addition to standard ML features, Yeti has polymorphic structure and variant types that are useful when processing ASTs. Yeti is quite easy to learn for a person with background in languages with Hindley-Milner type system (ML family, Haskell).

Scala[11] is a language that combines functional and object-oriented programming while trying to retain compatibility with the Java language. It seems to be complex language with many features and complicated typesystem, somewhat resembling C++ and C#.

IMP is highly configurable IDE generation toolkit. Unlike Xtext that provides default implementations for most IDE features, the IMP requires writing some code to enable these services. IMP comes with parser generator LPG, but does not depend on a specific parser and AST implementation. The LPG itself seems to be inferior to ANTLR, at least the lexer specifications are very verbose and inconvenient.

The solution would consist of the following components:

- AST representation: Yeti structures or Scala case classes.
- Parser generator: ANTLR, using the Xtext approach. The grammar description is annotated with information that is used to generate both the raw ANTLR grammar that parses the DSL program into AST representation (structures or case classes).
- Templating engine: one option is to use Java-based templating engine, such as StringTemplate[12]. When using Yeti, it is also possible to use escaped expressions in string literals.
- IDE generator: IMP that can is interfaced with ANTLR and can use IDE services written in HLL.
- Validation: HLL programs that navigate the AST and report errors.

Overall

[10]http://mth.github.com/yeti/
[11]http://www.scala-lang.org/
[12]http://www.stringtemplate.org/

This option comes with the least amount of dependencies and therefore can easily be integrated into build systems, applications etc. It is also very flexible because the DSL implementation can use all the possibilities offered on the Java platform (in a way, this is also true for Xtext, but writing complex transformations in Java can quicky get tedious).

This option is also the one requiring the most work to implement. It is somewhat unclear how many features we can implement during the first part of the project (ending in summer 2010).

Pros

- Very little dependencies and restrictions.
- DSL implementation is very easy to integrate into various components.
- Very flexible.
- Should have very good performance (does not use interpreted languages, expensive frameworks. Running speed of Scala should be similar to speed of Java).

Cons

- Potentially more costly to implement than the other options.
- With barebones implementation of the toolkit, implementing a DSL can require writing lot of code.

3.2 Work to Be Done

This section tries to list work that must be done for each of the candidate architectures so that the result would conform to requirements listed in section 2. The main requirements are:

- toolkit is suitable for developing simple DSLs (from zero to sixty in 3-4 hours);
- tookit is suitable for developing complex DSLs;
- DSL implementation can be embedded to build system and application.

An important thing to consider with respect to additional development work is the nature of changes or enhancements to the base tools. The types of development, in the decreasing order of desirability, are:

1. creating additional components that integrate with base components, packaging several base components with additional components[13] to form a complete DSL toolkit;
2. modifying the base components to fix bugs;
3. modifying the base components to add new functionality;
4. modifying the base components to change e.g. how they work.

The problem with options 3 and 4 is that modifications to the base component must be accepted by developers of the component (unless one wants to create fork of the component). Modifications to architecture of the component are very unlikely to be accepted by the developers.

3.2.1 Xtext

Xtext is a bit difficult case because it is currently complete and self-contained system that is developed based on some assumptions and it is almost impossible to change these assumptions.

Xtext's support for simple DSLs is very good, the problematic parts are complex DSLs (support for multi-step compilation and for inter-file linking) and integration options. Support for complex DSLs can in principle developed as addition to the existing code (although seamless integration will still require changes to the Xtext itself). However, integration options are mostly determined by basic design decision to integrate Xtext with other Eclipse components. The Fornax platform[14] offers Maven plugin and repository that can be used to build Xtext projects from command line, but this is still needlessly complex task.

The following list contains improvements that can be made to achieve toolkit that satisfies the requirements of the project.

- Develop a scheme for providing links between DSL source files. In theory, developing the correct linking or scoping service should suffice, but the finer points of linking are undocumented and this may also interfere with how the Ecore model is stored and processed.

[13] If the licence conditions of the base components allow doing so.
[14] http://www.fornax-platform.org/

- Develop some framework for complex compilation. A possible solution would be to move the compilation from Xpand-Xtend combination to Java code and make it easy to use the Java-based abstract representation from Xpand templates.
- Decouple from Eclipse services, e.g. Ecore metamodel. Instead, introduce dependency to interfaces that can then be implemented using Ecore or some more lightweight mechanism. This is a rather invasive change and quite unlikely to be accepted by the Xtext project.

3.2.2 Stratego

Stratego and Spoofax/IMP toolkit contain all the major parts of DSL toolkit. The main development activities are concerned with improving various aspects of the existing kit.

- Develop framework or wizard for creating simple DSLs. This should take care of boilerplate code that is necessary to get a complete DSL implementation.
- Contribute to Spoofax/IMP to make it more reliable.
- Add auto-completion feature to Spoofax/IMP.
- Contribute to Stratego/J (Java implementation of Stratego) to make it more reliable and faster.
- Contribute to Stratego to make the Windows port usable.
- Contribute to Stratego to add documentation to important parts of the standard library.

All the contributions to Stratego are additions or bug fixes/enhancements and getting them accepted should pose no problems.

3.2.3 ANTLR

The task list described in this section assumes that Scala is fairly mature language and that this project will not need to spend additional effort in developing and documenting it.

- Integrate IMP with Scala so that it is possible to build editor services in Scala. Include sensible default implementations so that simple DSLs can be created without much coding effort.
- Develop method for specifying annotated ANTLR grammars (similar to the approach used in Xtext). From this grammar description, DSL toolkit generates code that parses the DSL program to abstract representation (Scala objects).
- Develop framework or wizard for creating simple DSLs.
- Integrate the DSL toolkit with a templating engine that is used for generating code.
- Develop a Scala library for automating tasks related compiling and code generation.
- Develop method for packaging visual (IDE) and non-visual (parser and code generator) parts of the DSL implementation.
- If using Yeti instead of Scala, contribute to Yeti by making it more stable and by documenting the Yeti library.

This approach does not require modification of existing components (except maybe minor bug fixes or enhancements to IMP).

3.3 Evaluation

The main points from discussion from previous section are the following.

- Xtext offers good usability for simple DSLs, but is very difficult to integrate with other tools and to extend.
- Stratego uses language that can be difficult to learn, reliance on native executables can make integration into Java-based enterprise systems difficult.
- ANTLR-based custom implementation offers very good integration possibilities with the downside of potentially requiring the most effort to implement (and becoming "yet another research prototype", if the project fails to get traction in the user community). This option is also a good platform for future developments and allows integration of various research ideas.

Based on these considerations, the DSL toolkit will use **ANTLR-based architecture**. The following sections describe the proposed architecture in more detail.

4 Technical Issues

This section contains mostly exploratory overview of various technical issues. The detailed solutions to these issues will be developed during the course of the project.

4.1 Abstract Representation of DSL Program

Abstract representation of a DSL program should follow the examples of Xtext and IMP. The program is represented as object graph (as opposed to syntax tree). If two objects in the DSL program reference each other, this should also be reflected by direct reference in the object graph (as opposed to textual references).

Following the Scala ideology, the abstract representation should be implemented as case classes. One additional requirement is that abstract representation object should contain references to its position in source file (see also the discussion in section 4.2). However, the abstract representation should not depend too much on the parser and the concrete representation.

4.2 API for Language Services

Language services are pieces of code that are called by the IDE or the compiler to add behaviour that is specific for a given DSL. Examples of the language services are program validation, reference resolution, folding helper and content assist helper.

API for the language services should not depend on low-level details of the parser tokens and pointers to positions in source files. Instead, the service must be able to operate using abstract representation and the framework must automatically handle the correspondence between concrete and abstract syntax.

4.3 Linking

The linking problem is best explained with a simple example. Let's assume that we are creating a DSL that supports defining classes using a syntax in figure 1. The metamodel for this part of the language is shown in figure 2.

```
class Foo extends Bar {
    ...
}
```

Figure 1: Class definition

The abstract representation (model) corresponding to this metamodel is convenient to process. However, if the model only contains objects that are defined in the DSL program, then this metamodel can be restrictive, because it is often desirable to extend classes that are defined outside the DSL program, using other means than the DSL. If we add the possibility to extend externally defined classes, then the metamodel will look something like figure 3. Here the *ExternalClass* stands for external resource defined outside the DSL program.

The important point is that the decision whether *Bar* in the example code (figure 1) points to *Class* or *ExternalClass* generally requires information that is not available at parsing time[15]. Therefore, there should be some facility to post-process the abstract representation of the DSL program and replace the references with referenced objects. This post-processing step should have access to contextual information (e.g. lists of external classes) and the ability to introduce synthetical elements (e.g. for representing external objects).

4.4 AST Post-Processing

AST post-processing is generalization of the linking issue described in the previous subsection. In DSLs it is often useful to use syntactic shortcuts where the meaning of some element can only be determined from context[16].

[15] In Xtext this decision is done at parsing time using only local information. This approach mandates using different syntax for different types of extended objects. For example, one can use "*extends*" and "*extends external*" keywords to differentiate between the two options.

[16] For example, in C language "*(foo)(1 + 2)*" can mean either function call or cast, depending of whether *foo* is a type or a pointer to function.

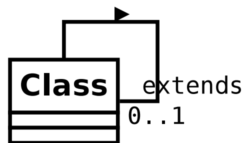

Figure 2: Metamodel for the class definition

The DSL toolkit should include processing step where the abstract representation of the DSL program is analyzed. During this analysis, the ambiguous parts of the program can be resolved, possibly using contextual information (e.g. to determine if some identifier is defined in external context). These parts are then resolved to reflect the intent of the DSL program.

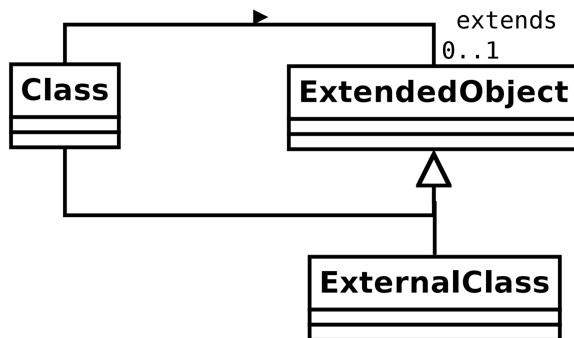

Figure 3: Metamodel with external classes

4.5 Validation Framework

The validation framework should support defining constraints for individual items of the DSL program without the need to traverse the entire object graph. In other words, the traversal code should reside in the framework, not in the DSL implementation.

Additionally, the DSL toolkit should include framework for implementing type system for the DSL. The abstract representation should allow annotating the nodes with type information and the validation framework should make it easy to implement rules for type propagation. One possible source

of inspiration can be MPS[17] that includes a language specifically targeted for describing type systems.

5 Third-Party Software

This section lists important pieces of third-party software that the DSL toolkit depends on:

- Eclipse 3.5 Galileo (latest stable version);
- IMP (exact version will be specified later);
- ANTLR 3.2 (latest stable version);
- Scala 2.7.7 (latest stable).

6 Conclusion

This document analyzed combinations of off-the-shelf components that can be used as base for the Simplicitas project. Based on evaluation of these components, the ANTLR parser generator and the IMP IDE generator were chosen.

This document additionally describes some technical issues that must be solved during detailed design of the DSL toolkit.

References

[1] Martin Fowler. Domain Specific Languages. An Introductory Example. http://martinfowler.com/dslwip/Intro.html, 6 August 2007.

[17] See http://www.jetbrains.com/mps/