

Automated DSL Construction Based on Software Product Lines

Changyun Huang, Ataru Osaka, Yasutaka Kamei and Naoyasu Ubayashi

Kyushu University, Fukuoka, Japan

Keywords: Domain-specific Language, Software Product Line, Language Workbench.

Abstract: DSL (Domain-Specific Language) is one of the important approaches for software abstraction. In the past decades, DSLs have been provided by expert engineers familiar with domain knowledge and programming language processors. It is not easy for ordinary programmers to construct DSLs for their own purposes. To deal with this problem, we propose a language workbench called *Argyle* that can automatically generate a DSL by only specifying a set of functions needed to the DSL and an execution platform supported by the DSL. *Argyle* is based on software product lines and consists of the following two steps: 1) development of the core assets for constructing a family of DSLs and 2) DSL configuration using these core assets. To demonstrate the effectiveness of our approach, we developed a prototype DSL for supporting MSR (Mining Software Repositories), the most active research field in software engineering.

1 INTRODUCTION

DSL (Domain-Specific Language) is one of the promising approaches to dealing with software abstraction. An application can be developed at a high abstraction level by using a DSL that encapsulates the details of domain knowledge and hides the usage of a specific software platform. DSL can improve the efficiency of software development and maintenance (Fowler, 2010). However, it is not easy to develop DSLs. In most cases, DSLs are constructed by expert engineers who are familiar with not only application domains but also programming language processors.

In this paper, we propose *Argyle*, a language workbench based on SPL (Software Product Line) (Clements and Northrop, 2001). Using *Argyle*, a programmer can obtain a DSL for a specific purpose by only specifying a set of functions needed to the DSL and an execution platform supported by the DSL. *Argyle* automatically generates a DSL grammar and its compiler by finding a set of language assets satisfying the specified requirements from a software product line targeted to the language development.

The remainder of this paper is structured as follows. In Section 2, we illustrate the overview of *Argyle*. In Section 3, we explain the design and implementation of *Argyle*. In Section 4, we show a prototype DSL for MSR (Mining Software Repositories) to demonstrate the usefulness of our approach. Concluding remarks are provided in Section 5.

2 Argyle: LANGUAGE WORKBENCH

The key concept of *Argyle* is to define language assets and combine them to construct a DSL according to user requirements. Though the SPL-based DSL construction has already been proved to be effective (Mernik et al., 2005), there is still a lack of the systematic methodology to achieve it. SPL engineering consists of domain engineering and application engineering. In our case, an application is a DSL.

Figure 1 shows the overview of *Argyle* supporting domain analysis, language design, and compiler construction. In domain analysis, we create a feature model based on SPL by analyzing what language assets are needed to construct a family of DSLs in the target application domain such as MSR. In language design, a DSL grammar in the form of BNF (Backus-Naur Form) is derived by user requirements (capabilities selection for anticipated DSL functionality). In DSL construction, a language processor for the derived grammar is constructed using a framework for DSL construction or an extensible programming language. In current implementation of *Argyle*, we use Xtext as a framework. Domain analysis, language design, and compiler construction correspond to domain engineering in SPL. Application development using the generated DSL corresponds to application engineering.

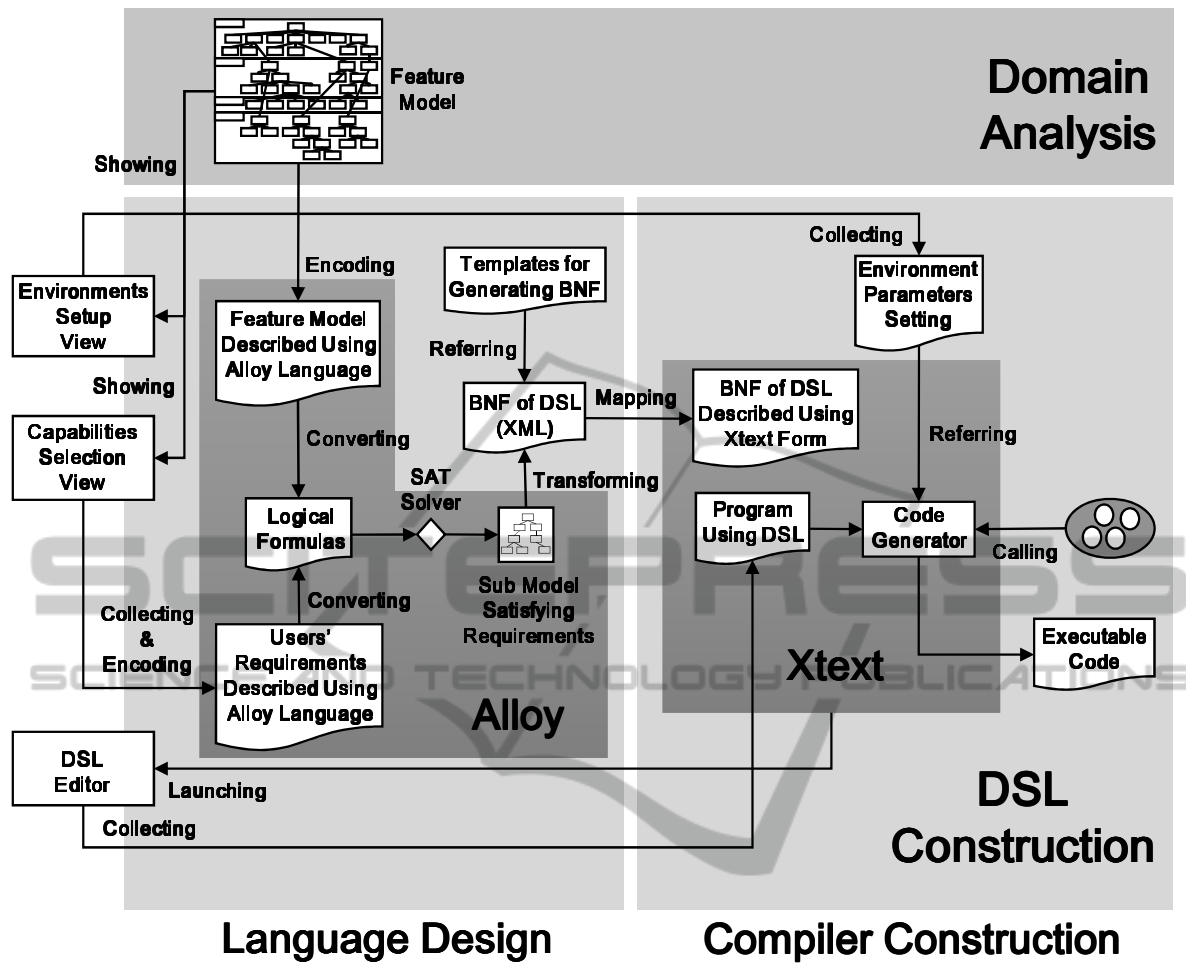


Figure 1: Argyle: Language Workbench for DSL Construction.

3 DESIGN AND IMPLEMENTATION OF Argyle

We show the details of domain analysis, language design, and compiler construction supported in the language workbench Argyle.

3.1 Domain Analysis Support

In Argyle, Feature-Oriented Domain Analysis (FODA) (Kang et al., 2002) (Kang et al., 1990) is used to define language assets and the constraints among them (Huang et al., 2013). A feature model in FODA consists of four layers: capability layer; operating environment layer; domain technology layer; and implementation technique layer.

Language assets in Argyle are defined according to FODA layers as follows:

- **Capability Assets.** A set of DSL functions needed in an application domain is defined as the assets in the capability layer.
- **Syntax Assets.** A variety of domain-specific operators and reserved words is defined as the assets in the operating environment layer. By configuring these operators and reserved words, Argyle generates the syntax of a DSL. In our prototype DSL for MSR, `select` is defined as a domain-specific operator for retrieving a set of data from a repository.
- **Data Type Assets.** A variety of domain-specific data types is defined as the assets in the domain technology layer. These data types are used in the operators defined as syntax assets. In the MSR application domain, domain-specific data types such as `Project` and `Author` are useful to write a program for analyzing software engineering data.

- **Implementation Assets.** Code generation templates and software components for executing a program written in a generated DSL are defined as the assets in the implementation technique layer.

The constraints such as the appearance order of the domain-specific operators, the relation between the argument types of a domain-specific operator and domain-specific data types, and the dependency among assets have to be defined in a feature model.

Argyle provides a diagram editor for modeling a feature diagram consisting of the above four layers. Domain analysis is performed by expert engineers familiar with domain knowledge and the theory of programming languages. However, in Argyle, language design shown in the next can be performed by ordinary programmers. Compiler construction is fully automated.

3.2 Language Design Support

3.2.1 Overview

In Argyle, a DSL is constructed by configuring language assets as follows:

1. Argyle acquires DSL requirements by letting a DSL user (programmer) select the necessary functions of the anticipated DSL from capability assets and an execution platform from implementation assets.
2. According to the above selection, Argyle extracts the related syntax assets and data type assets. After that, Argyle configures these assets to generate the DSL grammar in the form of BNF consisting of domain-specific operators, reserved words, and data types. Using this DSL, we can write a program that realizes the functions specified in step 1.
3. Argyle extracts a set of implementation assets for the DSL compiler to generate the executable code from a program written in the DSL generated in step 2.

DSL users have only to execute a few simple operations in step 1. Even the ordinary programmers without the deep knowledge of programming language processors can develop a DSL, because step 2 and 3 can be automatically performed by Argyle. In Argyle, a feature model, constraints among language assets, and user requirements are encoded into logical formulas. A generated DSL corresponds to the subset of these assets satisfying constraints and user requirements. We use Alloy (Jackson, 2006) (Gheyi et al., 2006) to obtain the subset (Nakajima and Ubayashi, 2007). Alloy, a name of both a formal specification

language used for describing structures (e.g., feature model) and an analyzer tool for exploring them, has been proved to be effective for describing and finding structures. In this paper, we call the specification language “Alloy language” and the analyzer tool “Alloy Analyzer” separately. Argyle can easily encode a feature model, constraints, and user requirements into Alloy code. We introduce the Alloy encoding patterns.

3.2.2 Translation from Feature Model into Alloy Language

Two elements (i.e., *signature* and *predicate*) of the Alloy language are mainly used in our encoding. Reserved words, **sig** and **pred**, are used to declare these two elements. A *signature* is used to define a set of atoms and may have some *fields* in its body that can be used to establish a relation to the other *signature*. The concept of a *signature* is as similar as a class in Java and a *field* in a *signature* looks like an attribute or property in a Java class. On the other hand, a *predicate* is used to define the interaction among *signatures*. Using *predicate* in Alloy language is as similar as using a method in Java.

The main encoding patterns consist of the encoding of features (Figure 2), relations (Figure 3), and composition rules (Figure 4).

An *abstract signature* “Feature” is defined at first by using the reserved word **abstract**. “Feature” has two *fields*: “up” and “select”. The “up” field points to the super feature. If the current feature is selected, the “select” field is set to “Yes” otherwise to “No”. A concrete feature is defined by extending *abstract signature* using the reserved word **extends**. All concrete features also have the above two *fields*.

The relations among features are defined by using *predicates*. In a *predicate*, “... =>... else ...” signifies the expression if ... then ... else ... and a reserved word **and** means a logical conjunction. Each *predicate* defines a relation among features by indicating, for example, whether these features can be selected at the same time or not.

A feature related to a syntax asset in the operating environment layer is distinctive from other features, because it is considered as an “Operator” in our encoding. An “Operator” is a special “Feature” that has some new *fields* such as “next”. These *fields* are used to define some additional information which are necessary on constructing BNF. For example, “next” is used to define the order between two “Operator”s when they appear in a BNF. The details are described inside the *predicate* “GeneratingRule”.

```

abstract sig Feature
{
  up : Feature
  select : Selected
}

abstract sig Selected {}
sig Yes, No extends Selected {}
sig F0, F1, F2 extends Feature {}
    
```

Figure 2: Encoding Features.

```

abstract sig Operator extends Feature
{
  ...
  next : Feature
  ...
}
pred GeneratingRule
[pre : Operator, post : Operator]
{
  ...
  pre.next = post
  ...
}
    
```

Figure 4: Encoding Composition Rules.

```

run {
  ...
  F0.select = Yes
  F1.select = Yes
  ...
}
    
```

Figure 5: Encoding User Requirements.

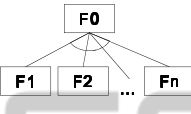
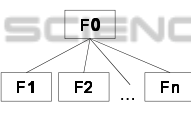
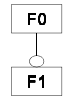
Feature Model (Relation)	Alloy Encoding
 <p>(Alternative)</p>	<pre> pred Alternative [sup : Feature, sub : some Feature] { sup.isSelected = Yes => one f : Feature (f in sub and f.isSelected = Yes and f.up = sup and (sub-f).isSelected = No and (sub-f).up = none) sup.isSelected = No => sub.isSelected = No and sub.up = none } </pre>
 <p>(Mandatory)</p>	<pre> pred Mandatory [sup : Feature, sub : Feature] { sup.isSelected = Yes => sub.isSelected = Yes and sub.up = sup sup.isSelected = No => sub.isSelected = No and sub.up = none } </pre>
 <p>(Optional)</p>	<pre> pred Optional [sup : Feature, sub : Feature] { sup.isSelected = No => sub.isSelected = No and sub.up = none sub.isSelected = Yes => sup.isSelected = Yes and sub.up = sup } </pre>

Figure 3: Encoding Relations.

3.2.3 BNF Generation using Alloy Analyzer

Argyle generates BNF with the help of Alloy Analyzer as follows:

1. Set user requirements;
2. Find submodels (a subset of features and their constraints) satisfying the requirements; and
3. Map the submodel to BNF.

First, user requirements (feature selections or asset selection) are encoded by setting related features' "select" fields to "Yes". The code is written inside the **run** command as shown in Figure 5. Next, this code is inputted to Alloy Analyzer that uses a SAT solver. The analysis results include the submodels satisfying the user requirements. Alloy Analyzer has two output types, diagram and XML file. For auto-process, Argyle uses the XML file which includes the information for generating the syntax of a DSL. Finally, XML

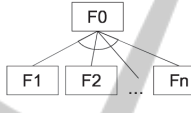
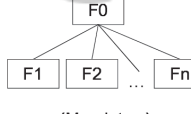

Feature Model (Relation)	Composition Rule	BNF
 <p>(Alternative)</p>	--	<pre> <F0> ::= <F1> <F2> ... <Fn> </pre>
 <p>(Mandatory)</p>	<pre> F0 next F1 F1 next F2 ... next Fn </pre>	<pre> <F0> ::= <F1> <F2> ... <Fn> </pre>
 <p>(Optional)</p>	--	<pre> <F0> ::= <F1> "" </pre>

Figure 6: Templates for generating BNF.

files are mapped to BNFs. Argyle prepares templates for this mapping process. Figure 6 shows the templates.

3.3 Automated Compiler Construction

Argyle translates the BNF of a generated DSL to Xtext, a Java-based extensible programming language framework. The process is as follows:

- Encode the generated BNF using Xtext; and
- Make a code generator to establish correspondences between the BNF and the target language (executable source code).

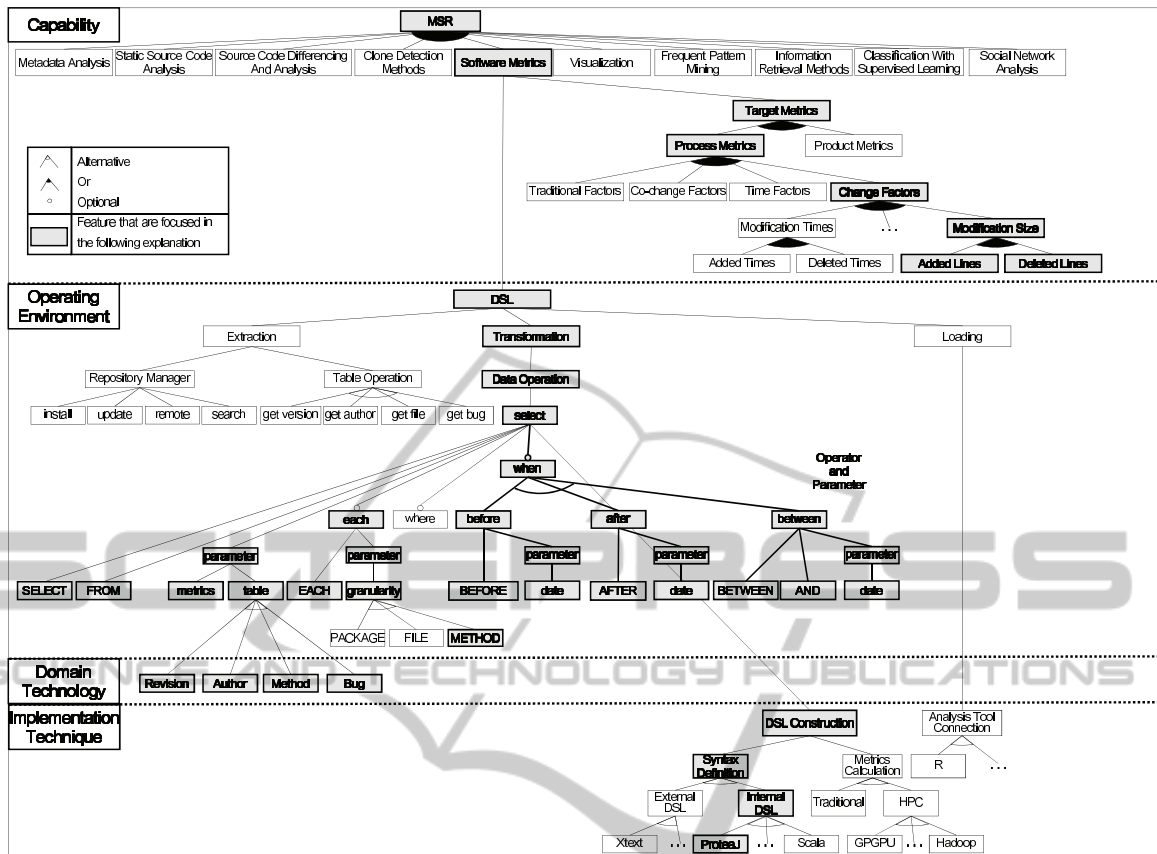


Figure 7: A Part of Feature Model for MSR.

Argyle provides an interpreter to translate a BNF to the corresponding Xtext code. Argyle also automatically generates the source code of a code generator which is used to transform a DSL program into the target language (e.g., Java) program that calls implementation assets. A DSL user (programmer) can use a DSL editor to write their own DSL programs.

4 CASE STUDY

In this section, we show a case study of constructing DSL for Mining Software Repositories (MSR) using Argyle.

4.1 Domain Analysis on MSR

MSR (Shang et al., 2011) is a research field where practitioners analyze the data stored in software repositories, such as version control systems, bug tracking systems, and mailing list etc., to discover meaningful information to support the software development. While there are many work needed to be

done in MSR, deriving necessary data from repositories is one of the most important work (Dyer et al., 2013) (Gu et al., 2012). In our case study, Argyle provides an approach to constructing a simple SQL-like DSL for retrieving related data from repositories (Yamashita, 2013). Figure 7 shows a part of domain analysis on MSR. This feature model includes language assets used to construct SQL-like DSL statements, such as “select ... from ...” and some optional operators to set period (e.g., “between ... and ...”) and granularity (e.g., “each ...”).

4.2 Specification of User Requirements

To give user-friendly interfaces to specify user requirements, Argyle provides the following views as shown in Figure 8: 1) capabilities selection view and 2) environments setup view.

4.2.1 Capabilities Selection View (Area 1)

Argyle provides capability assets in this view that lists what DSLs can be used to do. Each capability asset shown in this view means that Argyle provides

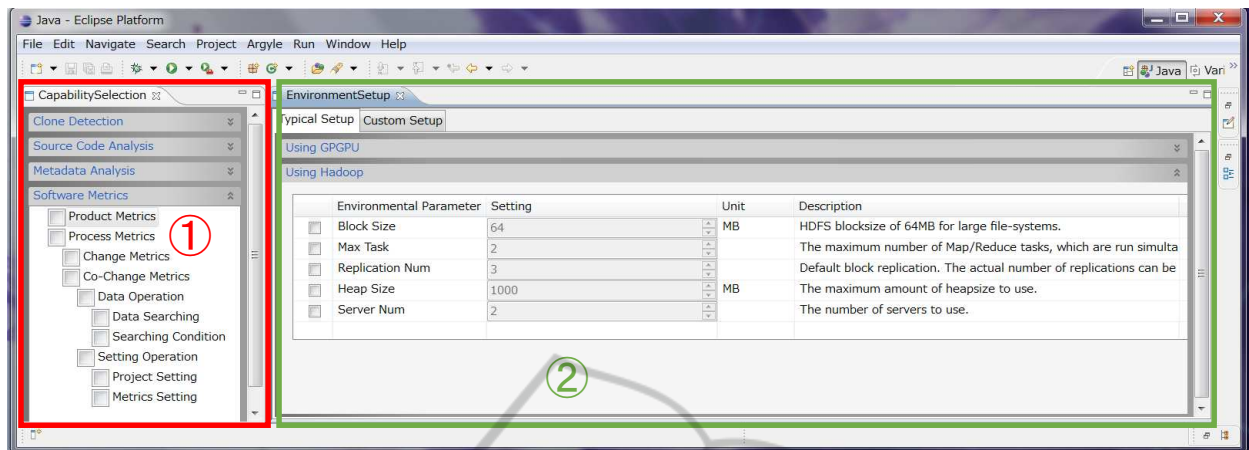


Figure 8: Setting Interfaces.

syntax assets and data type assets needed to realize the corresponding capability. Argyle tries to provide the easy-to-understand description of each capability, such as “Data Searching” and “Granularity Setting”, because we assume that users of Argyle have not enough knowledge about the details of DSL construction. Users can select capabilities they like by selecting related check boxes.

4.2.2 Environments Setup View (Area 2)

Argyle requires users to decide which platform their DSLs will be executed in. This view is used to deal with the execution environment problems. It allows users not only to decide the environments but also set the related parameters of those environments (if necessary). As an example, in Figure 8 there are two environments Hadoop and GPGPU. To take care of users who have little knowledge about these environments, Argyle prepares default parameter settings for using Hadoop or GPGPU.

4.3 DSL Generation using Argyle

After user requirements are specified, Argyle generates a DSL.

The feature model shown in Figure 7 consisting of about 30 features takes about 100 lines of Alloy code. Table 1 shows the details of Alloy code. A part of Alloy code can be reused for other DSL generations.

Figure 9 shows a submodel derived by Alloy Analyzer. This submodel is stored in an XML file that is mapped to the BNF. This BNF is also stored in another XML file with a form shown in Figure 10. Figure 11 shows a part of description of BNF in Xtext. Argyle can provide a DSL editor and a code generator.

Table 1: Composition of Alloy Code.

Number of Features	30
Total Alloy Code (lines)	86
Reusable Code	28
Relations Definitions	16
Abstract Signatures Definitions	9
Composition Rule Definition	3
None Reusable	47
Features Definitions	6
Relations Settings	27
Composition Rule Settings	14
Others	11

4.4 DSL Editor and Code Generation

Argyle launches a DSL editor for users as shown in Figure 12. This editor allows users to make their program using the DSL whose BNF is generated above. In this case, a statement such like “select ... from ...” can be used in a program. A program written in the DSL editor is translated to a programming language which can be used in the environment that was indicated by users through environments setup view. For example, users can use the Hadoop platform to calculate the software engineering metrics.

5 CONCLUSIONS

In this paper, we provided a language workbench called Argyle which is used to support DSL constructions. The key idea of Argyle is to define language elements as assets and combine them according to user requirements. Argyle gives developers, who want to use DSLs, an approach to make DSL by themselves even though they have not enough knowledge about

```

<alloy builddate="2014-XX-XX XX:XX EST">
<instance bitwidth="0" maxseq="0" command="XX" filename="XX">
...
<sig label="this/SELECT" ID="7" parentID="8" one="yes">
<atom label="SELECT"/>
</sig>
<sig label="this/FROM" ID="9" parentID="8" one="yes">
<atom label="FROM"/>
</sig>
<sig label="this/metrics" ID="10" parentID="8" one="yes">
<atom label="metrics"/>
</sig>
<sig label="this/table" ID="11" parentID="8" one="yes">
<atom label="table"/>
</sig>
...
<sig label="this/EACH" ID="12" parentID="8" one="yes">
<atom label="EACH"/>
</sig>
<sig label="this/granularity" ID="13" parentID="8" one="yes">
<atom label="granularity"/>
</sig>
...
<field label="next" ID="19" parentID="8">
<tuple> <atom label="SELECT"/> <atom label="metrics"/> </tuple>
<tuple> <atom label="metrics"/> <atom label="FROM"/> </tuple>
<tuple> <atom label="FROM"/> <atom label="table"/> </tuple>
...
<tuple> <atom label="EACH"/> <atom label="granularity"/> </tuple>
<types> <type ID="8"/> <type ID="9"/> <type ID="10"/> <type ID="11"/> <type ID="12"/> <type ID="13"/> </types>
...
</instance>
...
</alloy>

```

Figure 9: Results of Alloy Analyzer stored in a XML File.

```

<syntax>
...
<statement label="select" type="RepositoryTable">
<keyword label="SELECT"/>
<parameter label="metrics" type="String"/>
<keyword label="FROM"/>
<parameter label="table" type="Table"/>
<keyword label="EACH"/>
<parameter label="granularity" type="Granularity"/>
</statement>
...
</syntax>

```

Figure 10: A part of BNF stored in XML file.

programming language processor.

Currently, Argyle has only a small set of defined language assets. This set needs to be expanded to support more user requirements. However, it is difficult to complete this work only by ourselves. The better way is to encourage many programmers to add new language assets to Argyle. This is our future work.

ACKNOWLEDGEMENTS

This research was conducted as part of the Core Research for Evolutional Science and Technology (CREST) Program, “Software development for post petascale supercomputing — Modularity for super-computing”, by the Japan Science and Technology Agency

grammar org.xtext.example.mydsl.MyDsl with org.eclipse.xtext.common.Terminals

generate myDsl "http://www.xtext.org/example/mydsl/MyDsl"

Model:
(types+=Type)*
(statements+=Statement)*;
Type:
Projects | Metrics;
Statement:
SelectStatement | SetStatement | OptionalStatement;
OptionalStatement:
EachStatement | AfterStatement | BeforeStatement |
BetweenStatement;
BetweenStatement:
'between' date1=INT 'and' date2=INT;
BeforeStatement:
'before' date=INT;
AfterStatement:
'after' date=INT;
EachStatement:
'each' granularity=ID;
Metrics:
'Metrics' metricsname=ID;
Projects:
'Project' projectname=ID;
SetStatement:
'set' object=ID 'name' objectname=ID;
SelectStatement:
'select' metrics=ID 'from' project=ID (optional ?=
[OptionalStatement])?;

Figure 11: BNF Representation in Xtext.

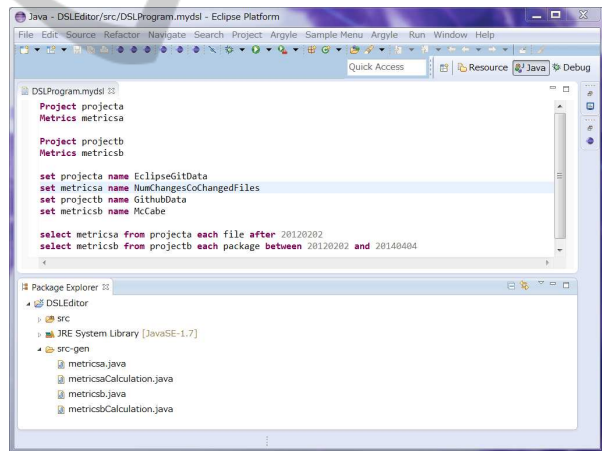


Figure 12: DSL Editor.

REFERENCES

Clements, P. and Northrop, L. (2001). Software product lines. Addison-Wesley.

Dyer, R., Nguyen, H. A., Rajan, H., and Nguyen, T. N. (2013). Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *35th International Conference on Software Engineering*.

Fowler, M. (2010). *Domain-Specific Languages*. Addison-Wesley Professional.

- Gheyi, R., Massoni, T., and Borba, P. (2006). A theory for feature models in alloy. In *First Alloy Workshop*.
- Gu, Z., Barr, E. T., Schleck, D., and Su, Z. (2012). Reusing debugging knowledge via trace-based bug search. In *12th ACM International Conference on Object Oriented Programming Systems Languages and Applications*.
- Huang, C., Yamashita, K., Kamei, Y., Hisazumi, K., and Ubayashi, N. (2013). Domain analysis for mining software repositories -towards feature-based dsl construction-. In *4th International Workshop on Product Line Approaches in Software Engineering*.
- Jackson, D. (2006). *Software Abstractions*. The MIT Press.
- Kang, K., Cohen, S., Hess, J., Nowak, W., and Peterson, S. (1990). Feature-oriented domain analysis feasibility study. Technical report, Carnegie Mellon University/Software Engineering Institute.
- Kang, K. C., Lee, J., and Donohoe, P. (2002). Feature-oriented product line engineering. *IEEE Software*.
- Mernik, M., Heering, J., and Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM Computing Surveys*.
- Nakajima, S. and Ubayashi, N. (2007). Lightweight formal analysis of foda feature diagrams. In *4th International Workshop on Rapid Integration of Software Engineering techniques*.
- Shang, W., Adams, B., and Hassan, A. E. (2011). Using pig as a data preparation language for large-scale mining software repositories studies: An experience report. *Journal of Systems and Software*.
- Yamashita, K. (2013). Modular construction of an analysis tool for mining software repositories. In *ACM Student Research Competition at the 13th Aspect-Oriented Software Development Conference*.