A Comparison of Tool Support for Textual Domain-Specific Languages

Michael Pfeiffer
Software Competence Center Hagenberg
michael.pfeiffer@scch.at

Josef Pichler
Software Competence Center Hagenberg
josef.pichler@scch.at

Abstract

A domain-specific language is a specialized and problem-oriented language. Successful application of a DSL largely depends on provided tools, so called language workbenches that support end-programmers creating, editing, and maintaining programs written in a DSL. In this paper, we describe four different tools supporting the creating of language workbenches, identify commonalities and differences and compare these tools by means of a set of criteria.

1. Introdcution

Domain-specific languages (DSL) are languages tailored to a specific application domain [1]. They have the potential to reduce complexity of software development by raising the abstraction level towards an application domain. According to the application domain, different notations (textual, graphical, tabular) are used. Textual languages have long tradition in the domain of textual meta-modeling languages like PSL/PSA [2] used for analysis and documentation of requirements and functional specifications for information systems. In this paper, we focus on textual languages following the source-to-source transformation (see [3]), i.e. DSL code is compiled into a general purpose programming language like Java and C#. Such textual DSLs are applied successfully in various areas. For instance, Monaco [4] is a language for end-user programming for automation systems and Tbl [5] is a language for test scripts of mobile devices. Successful application of such languages largely depends on provided tool support. A tool that support end-programmers creating, editing, and maintaining programs in a DSL is called a language workbench [6]. Users expect workbenches including full-featured editors with syntax highlighting, code completion, etc. as available for general programming languages. Such editors are often implemented manually resulting in effort and costs that can be reduced only to a certain degree by using editor frameworks, for example the Eclipse platform. Furthermore, language workbenches require further components like lexical scanner, parser, and data structures for the abstract syntax tree as well as generators to transform the program to a general programming language.

Both industry and research community have recognized this circumstance so that today several tools exists that allows automatically generation of such language workbenches based on meta-models or grammar definitions. The automatically generation of a compiler frontend including scanner and parser out of a context-free grammar is an established area and a lot of such tools like CoCo/R [7] or Antlr [8] are available today. The additional generation of editors towards a full-fledged language workbench is a consequent further development.

In this paper, we compare four different tools supporting the generation of language workbenches for textual DSLs. In particular, we describe and compare approaches for definition of the language, used techniques for parsing and code generation as well as the functional range of the resulting workbenches. For illustration of commonalities and differences, we created language workbenches for a simple language used to define finite state machines. Furthermore, we provide comparison based on a subset of criteria defined by the feature model of DSL tools given in [9].

This paper is organized as follows. Section 2 defines criteria for comparison. Section 3 holds the language definition for our example language for finite state machines. Section 4 gives an overview of tools supporting textual DSL. Section 5 compares these tools based on the defined criteria.

2. Criteria for Comparison DSL Support

The goal of this section is to describe the criteria to be used for the comparison of selected DSL tools. The criteria are based on the DSL feature model defined by Langlois et al. [9]. This feature model covers languages, transformation, tooling, and process aspects. Instead of defining a new criteria catalog, the reuse of an existing catalog facilitates the comparison of our results with other comparisons too, in particular, with the comparison of graphical DSL tools given in [9].

As our comparison will mainly focus on tool support of textual DSL, we omitted some criteria. For example, the process aspects that are defined as optional features only are omitted. Hence, the criteria to be used may be split into three groups:

- Language (LA)
- Transformation (TR)
- Tool (TO)

The criteria for each group and the difference to the feature model are explained at a glance below. The abbreviation of the group combined with a criteria identifier is unique and used later to compactly describe the results. In contrast to the feature model in [9], the criteria will be explained by means of questions; this is a very straightforward and simple approach and should clearly show what the respective criteria are about.

2.1 Language

This group of criteria comprises tool support for both the abstract syntax (AS) and concrete syntax (CS).

- LA-AS1. Which representation is used for the abstract syntax (abstract syntax tree or abstract syntax graph)?
- LA-AS2. Which representation is used for the definition of the abstract syntax (grammar or a meta-model)?

- LA-AS3. Can the abstract syntax be composed of several grammars or meta-models?
- LA-CS1. Which technique is used to map abstract syntax to concrete syntax?
- LA-CS2. Which representation (text, graphic, wizard, or table) can be used for the concrete syntax?
- LA-CS3. Which style (declarative or imperative) can be used for the concrete syntax?

As our focus is on textual tools, the criteria LA-CS2 will be evaluated as text for all tools.

2.2 Transformation

The criteria of this group have to answer questions about specification of transformation, expected target assets, and the realization to produce the expected target assets. The transformation realizes the correspondence from the problem to the solution space. A target asset is a software artifact resulting from the transformation. The criteria of this group cover target assets (TA) and the operational transformation (OT).

- TR-TA1. Which representations of the target asset (model, text, graphic, binary) are possible?
- TR-TA2. Which kind of support of asset update (destructive or incremental) is possible?
- TR-TA3. Which kind of support for integration of target assets is used?
- TR-OT1. Which kinds of transformation technique are used (model-to-model ($M \rightarrow M$), model-to-text ($M \rightarrow T$), $T \rightarrow T$, $T \rightarrow M$)?
- TR-OT2. Which mode (compilation or interpretation) is used for transformation execution?
- TR-OT3. Which environment (internal or external) is used for transformation execution?
- TR-OT4. Which scheduling (implicit or explicit) form is used?
- TR-OT5. Which location (internal or external) is used?
- TR-OT6. Which automation level (manual or automated) is used?

The criteria variability and phasing of the feature model are omitted.

2.3 Tool

This group of criteria stresses on overall tool support.

- TO-RA1. Which respect of abstraction (intrusive or seamless) is used?
- TO-AS1. Which kind of assistance (static or adaptive) is provided?
- TO-AS2. Which kind of process guidance (step or workflow) is provided?
- TO-AS3. Which kind of checking (completeness or consistency) is supported?

Criteria of the feature model covering quality factors like reliability and efficiency are omitted because they are not specific to DSL tools and, hence, do not make a substantial contribution for our comparison.

3. Example

For illustration of commonalities and differences of tools in the next section, we show examples based on a simple language used to define finite state machines (FSM). For our example, a FSM will be described in a text file following a simple syntax. The target asset is Java source code. The lexical and syntactical structure of FSM text files are defined by the following grammar given in the EBNF proposed by Wirth [10]:

```
FSM         = "inputAlphabet" string
                "outputAlphabet" string {State}.
State       = ["start"] "state" id {Transition}.
Transition  = "transition" char ["/" char] "->" id.
```

A FSM is described by the input alphabet, the output alphabet, and a set of states. A state has an identifier and a set of transitions that connect a state to following states. A transition is composed of an input character that triggers the transition, an optional output character, and the identifier of the following state.

Terminal classes *id*, *string*, and *char* of the FSM are specified as regular expressions:

id = [:alpha:][:alnum:]+ *string* = "[~ \t\n\r]*" *char* = [~ \t\n\r]

The following example shows a finite state machine that determines if a binary number has an odd or even number of zero digits.

```
// Digits of a binary number
inputAlphabet "01"
// Char 'e' for even and 'o' for odd
outputAlphabet "eo"
start state Even
transition 0 / o -> Odd
transition 1 / e -> Even
state Odd
transition 0 / e -> Even
transition 1 / o -> Odd
```

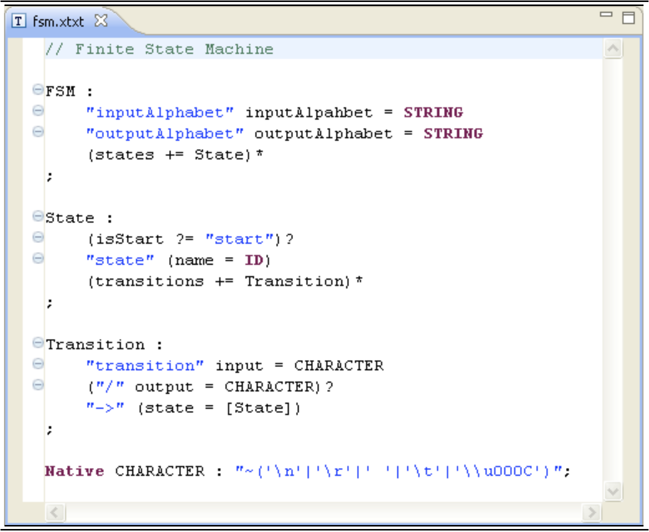
Even this example is very simple it suffices to demonstrate most important aspects of different tools and differences between these tools.

4. Overview of Selected Tools

In this section, we give an overview of tools supporting textual DSLs. According to the focus of this paper, a first investigation resulted in following tools to be considered:

- openArchitectureWare (oAW), version 4.3
- Meta Programming System (MPS), early access version
- MontiCore, version 1.1.5
- IDE Meta-Tooling Platform (IMP), version 0.1.74
- Textual Concrete Syntax (TCS), version 0.0.1
- Textual Editing Framework (TEF), version 1.0.3
- CodeWorker, version 3.5

For the selection of tools to be included, we considered to omit tools that are very similar (at least on technology view) to other ones or system that cannot be considered to be used in real projects (for example, due to immature academic tool). For example, oAW, TCS, and TEF are based on same technology (Eclipse, GMT); hence, the systems TCS and TEF were omitted in favor of the mature oAW. The CodeWorker tool was not considered because it did not provide editor support at the time of the evaluation.



```
// Finite State Machine

FSM :
    "inputAlphabet" inputAlpahbet = STRING
    "outputAlphabet" outputAlphabet = STRING
    (states += State)*
;

State :
    (isStart ?= "start")?
    "state" (name = ID)
    (transitions += Transition)*
;

Transition :
    "transition" input = CHARACTER
    ("/" output = CHARACTER)?
    "->" (state = [State])
;

Native CHARACTER : "~('\n'|'\r'|' '|'\t'|'\\u000C')";
```

Figure 1. xText grammar editor.

4.1 openArchitectureWare

The openArchitectureWare [11] [12] is an open source project for model-driven software development, based on the Eclipse platform. The sub-component xText provides a framework for textual languages, whereas the resulting DSL environment is also based on the Eclipse platform.

The concrete syntax is specified as context-free grammar including production rules and types of terminal symbols. Figure 1 shows the grammar of our FSM example in the corresponding editor provided by oAW. The schema of the grammar supports cross references, type inheritance, and enumerations. The corresponding editor supports syntax highlighting, code completion of keywords and meta-model elements, validation and multi-file templates among other things.

The grammar contains sufficient data to generate the main building blocks of a language workbench supporting the specified language including:

- an abstract syntax graph (ASG)
- scanner and parser for text-to-model transformation by means of Antlr [8]
- a generator for model-to-text transformation by means of xPand part of oAW
- an editor based on the Eclipse

The generation of all these artifacts is configured and controlled by a so-called workflow definition file, as shown in Figure 2.

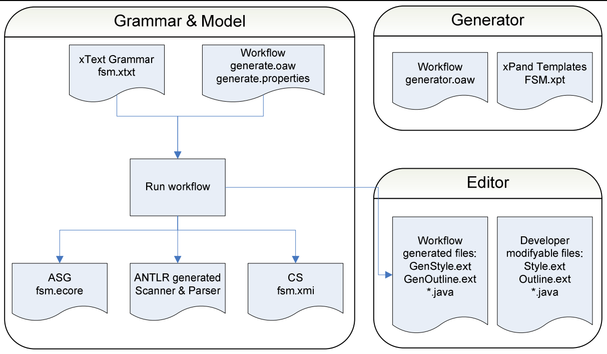
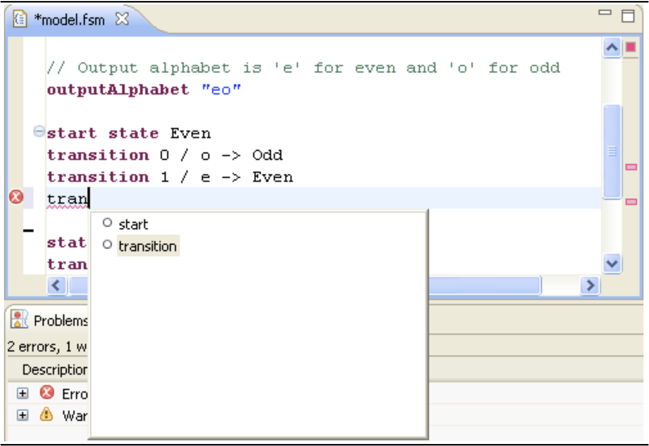


Figure 2. Overview oAW.



```
// Output alphabet is 'e' for even and 'o' for odd
outputAlphabet "eo"

start state Even
transition 0 / o -> Odd
transition 1 / e -> Even
tran

stat
tran
```

Figure 3. oAW FSM editor.

The generated artifacts are separated into files (Java source code or configuration files) that are indented to be reworked and modified by the developer and files that are regenerated each time the workflow is run. This separation avoids a common problem [13] of synchronization of single files that are both generated automatically but also reworked by the developer.

The generated ASG in form of an Ecore model is based on the Eclipse Modeling Framework [14]. oAW uses Antlr to generate scanner and parser for text-to-model transformation whereas for model/text transformation, the template engine xPand, part of oAW, is used.

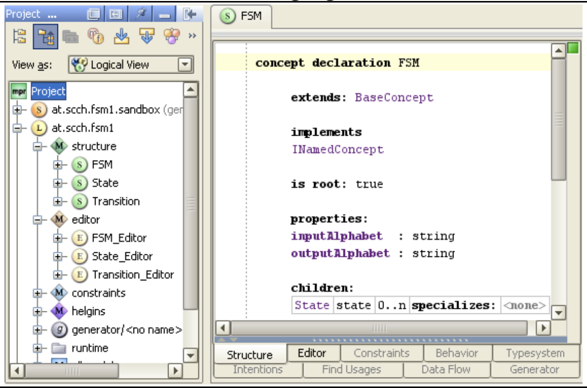
The generated text editor for manipulation of DSL texts supports syntax highlighting, code completion, validation of syntax and model constraints checking. Figure 3 shows an example FSM opened in the editor whereas a syntax error is showed together with provided code completion in order to fix the error.

At the current state, oAW has some limitations of the expressiveness of a DSL grammar. However it provides good tools with early validations to quickly develop a DSL with semantic checks and text generator.

4.2 Meta Programming System

The Meta Programming System (MPS) [15] [16] of JetBrains has not been released yet; however, an evaluation version can be obtained in an early access program. Nonetheless, we have included the MPS to our comparison because of its unique technique to define the syntax of a DSL and the used cell-based editing model.

The abstract syntax tree (AST) is specified by a list of so-called concepts. In MPS, all concepts of one language together are called the structure of the language.



```
concept declaration FSM

    extends: BaseConcept

    implements
    INamedConcept

    is root: true

    properties:
    inputAlphabet  : string
    outputAlphabet : string

    children:
    State state 0..n specializes: <none>
```

Figure 4. MPS concept editor.

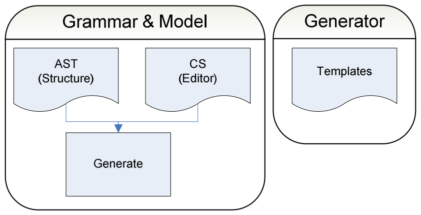


Figure 5. Overview MPS.

Figure 4 shows the concept for the top-level element FSM of our example described in section 3. A concept can extend another concept to inherit its properties. For instance, the concept *FSM* defined Figure 4 extends the built-in concept *BaseConcept*.

Concepts are connected together by means of references or aggregations. An example for an aggregation is the connection between the *FSM* concept and the *State* concepts. Furthermore, a concept has properties that are typically used to contain the values of terminal symbols. One concept of a language must be declared as root element.

The concrete syntax of a concept has to be defined in form of static text that is not editable and editable cells. An example is given later on in this section.

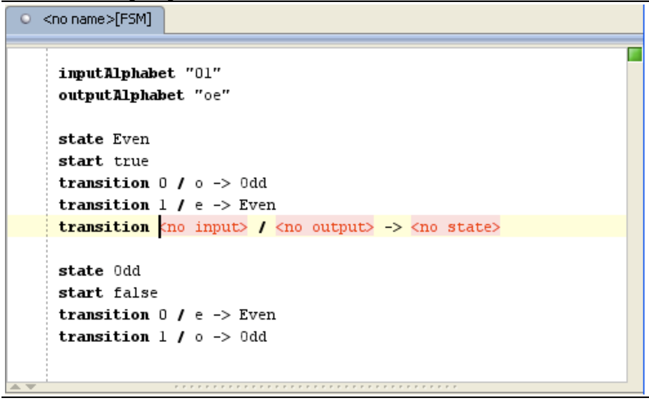
Figure 5 gives an overview of the affected models for the implementation of the example described in section 3. MPS provides a generator following a model-to-model approach that allows the transformation from any model to any other model. The transformation is described by means of templates that are edited with a cell editor.

Figure 6 shows the resulting cell editor for our FSM language containing the example FSM defined in section 3. The editing area is separated into read-only areas and editable cells. For instance, the properties of the third transition in the example are editable, whereas the text outside is read-only. Manipulation of texts follows always the same pattern: first, the kind of element to be added or edited must be selected and, second, the properties for that element can be edited in so called cell editors.

MPS is interesting as it provides a self-contained complete solution that uses its own technology every where, for example editors for manipulation of syntax or templates follow the same approach as the resulting editor for our DSL. For another MPS example we refer to an article by Martin Fowler [16].

4.3 MontiCore

The MontiCore framework [17] [18] [19] is a research project by Software Systems Engineering Institute, TU Braunschweig, Germany. Its core and the generated editor for the DSL are based on the Java/Eclipse platform.

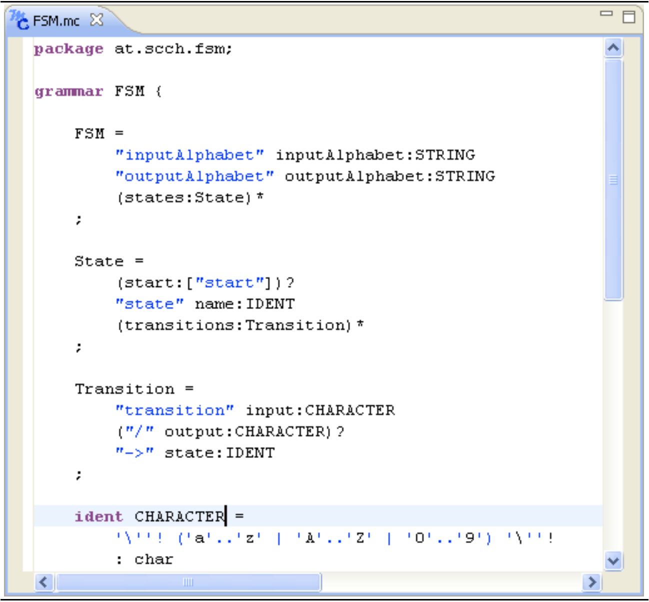


```
inputAlphabet "01"
outputAlphabet "oe"

state Even
start true
transition 0 / o -> Odd
transition 1 / e -> Even
transition <no input> / <no output> -> <no state>

state Odd
start false
transition 0 / e -> Even
transition 1 / o -> Odd
```

Figure 6. MPS FSM editor.



```
package at.scch.fsm;

grammar FSM {

    FSM =
        "inputAlphabet" inputAlphabet:STRING
        "outputAlphabet" outputAlphabet:STRING
        (states:State)*
    ;

    State =
        (start:["start"])?
        "state" name:IDENT
        (transitions:Transition)*
    ;

    Transition =
        "transition" input:CHARACTER
        ("/" output:CHARACTER)?
        "->" state:IDENT
    ;

    ident CHARACTER =
        '\'' ('a'..'z' | 'A'..'Z' | '0'..'9') '\''
        : char
```

Figure 7. MontiCore grammar editor.

MontiCore uses an enriched context-free grammar for the specification of the concrete syntax which is similar to the input format of Antlr [8]. In addition to rules of the grammar, it contains definitions of the data types of terminal symbols, whereas the whole range of simple Java data types is supported.

Figure 7 shows the MontiCore editor with the grammar described in section 3. According to the used input format, the grammar differs slightly from the grammar described in section 3. The input and output character are now enclosed within two apostrophes. Furthermore, MontiCore does not accept a terminal symbol that consists of a single character only. MontiCore uses a single source for defining concrete and abstract syntax of a DSL [18] [20]. The grammar contains sufficient data to generate data structures for the abstract syntax tree as well as scanner and parser, as shown in Figure 8. The later ones are generated by means of the compiler-generator Antlr. However, the parser must be connected with the editor for validation manually by implementing some glue code in the Java programming language.

The model transformation can either be realized in form of Java code using the visitor pattern or using the provided template engine. The model transformation can be triggered by the user inside the editor.

Figure 9 shows the resulting editor for our FSM language with an example FSM. As shown by the example, the syntax is almost as specified in section 3. Working with MontiCore requires labor however it is also more flexible in the expressiveness of the DSL.

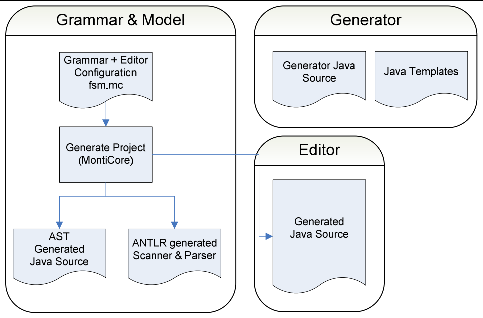
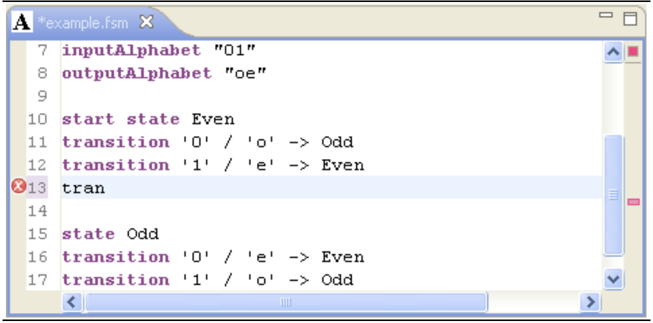

Grammar & Model
Grammar + Editor Configuration fsm.mc
Generate Project (MontiCore)
AST Generated Java Source
ANTLR generated Scanner & Parser
Generator
Generator Java Source
Java Templates
Editor
Generated Java Source

Figure 8. Overview MontiCore.



*example.fsm

```
inputAlphabet "01"
outputAlphabet "oe"

start state Even
transition '0' / 'o' -> Odd
transition '1' / 'e' -> Even
tran

state Odd
transition '0' / 'e' -> Even
transition '1' / 'o' -> Odd
```

Figure 9. MontiCore FSM editor.

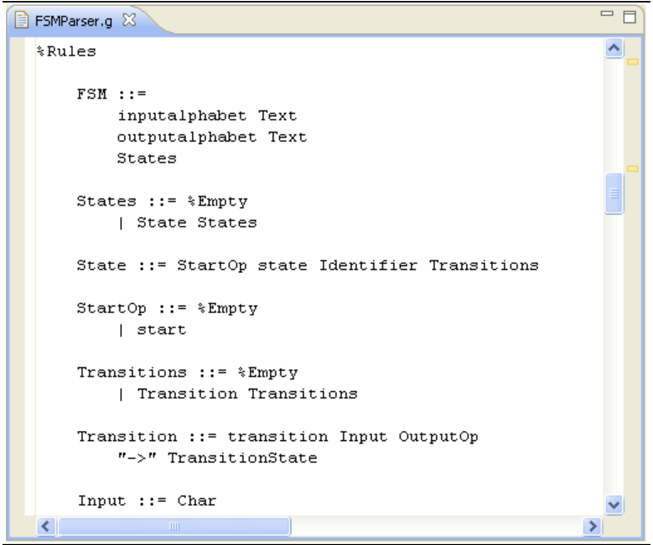
4.4 IDE Meta-Tooling Platform

The IDE Meta-Tooling Platform (IMP) [21] is an open source project begun at IBM Watson Research. Its goal is to ease the development of IDE support for a new programming language. IMP provides wizards to generated code skeletons for a large range of features required for an IDE of a new language.

The concrete syntax of a language is defined by a context-free grammar, whereas the grammar is divided into input used to generate the scanner (lexer) and the parser. Figure 10 shows the input for the parser for our example language edited in a plain text editor. Productions of the grammar can be annotated with Java code that is added to the data structures for the abstract syntax tree.

IMP includes the parser generator LPG [22] to generate the data types for the abstract syntax tree as well as scanner (lexer) and parser, as shown in Figure 11. Furthermore, IMP generates a full-featured editor based on the Eclipse platform including outline and syntax highlighting, whereas the editor includes the generated parser. The resulting editor is registered in the Eclipse platform using extension points. IMP utilizes the visitor pattern for model transformations. For example a source code formatter can be implemented using the visitor pattern.

Figure 12 shows the editor with the example FSM. IMP provides a generic text editor that supports syntax highlighting, folding, formatting and code completion. An outline of the text in the editor is provided too. IMP does not include a template engine so other solutions have to be used instead. For our example, the code generator was implemented by means of JET [23].



FSMParser.g

```
%Rules

    FSM ::=
        inputalphabet Text
        outputalphabet Text
        States

    States ::= %Empty
        | State States

    State ::= StartOp state Identifier Transitions

    StartOp ::= %Empty
        | start

    Transitions ::= %Empty
        | Transition Transitions

    Transition ::= transition Input OutputOp
        "->" TransitionState

    Input ::= Char
```

Figure 10. IMP parser grammar editor.

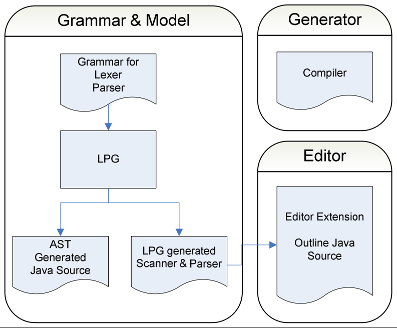


Figure 11. Overview IMP.

As the goal of IMP indicates its usage is not limited to the construction of DSLs. Compared to the other tools it requires more programming effort but on the other hand provides also more flexibility.

5. Comparison

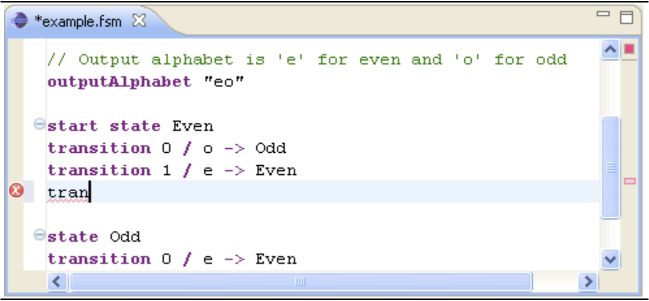
In this section, we will compare the tools, described in the previous section, by means of the criteria defined in section 2. Figure 13 contains the result of the comparison at a glance that is described in more detail below.

5.1 Language

For the representation of the abstract syntax, both trees (AST) and graphs (ASG) are used and realized with different strategies. *MontiCore* and *IMP* generate Java source code implementing the abstract syntax tree, whereas *oAW* and *MPS* use generic meta-models for the abstract syntax graph. Except for *IMP*, the definition of the abstract syntax can be composed of several grammars or meta-models.

All tools differ in the mapping between concrete syntax and abstract syntax. The tools *oAW* and *MontiCore* use a single source for defining both the concrete and the abstract syntax. In *MPS*, a user first defines the abstract syntax in form of concepts and, afterwards, he defines the concrete syntax for every concept. By contrast, *IMP* requires the definition of the concrete syntax only, and the abstract syntax is derived automatically from the concrete syntax.

All tools have in common, that the concrete syntax is represented as text. Of course, this is a consequence of the focus of this paper on tools for textual languages. However, MPS stores the model as XML document and presents it as text in the editor only. All tools support both declarative and imperative style of languages.



*example.fsm

```
// Output alphabet is 'e' for even and 'o' for odd
outputAlphabet "eo"

start state Even
transition 0 / o -> Odd
transition 1 / e -> Even
tran

state Odd
transition 0 / e -> Even
```

Figure 12. IMP FSM editor.

5.2 Transformation

All tools allow the generation of text files as target assets, however, the support level differs. *oAW* provides an outstanding transformation support. The template editor of *oAW* provides code completion and early error detection. *MPS* support model-to-model transformation out-of-the-box, so text generation requires generation of a target model that, in turn, can be transformed into text. *MontiCore* provides a rudimentary template engine only. The invocation of the template engine requires to write Java code for each text file to be generated, though. *IMP* is the only tool that has no transformation support built-in, so transformation must be implemented in the Java programming language following the visitor pattern. All tools provide destructive update of generated assets only.

All tools except *IMP* provide model-to-text mappings. Additionally, *MontiCore* and *MPS* provides model-to-model mappings too. For *MPS*, the model-to-text mapping requires an intermediate model-to-model mapping.

Concerning operational translation covered by criteria TR-OT2 – TR-OT6, all tools are very similar. Three tools follow an interpreted approach whereas a template engine fills templates at runtime; only *IMP* requires compilation of visitor classes implemented in the Java programming language. *MPS* allows starting the transformation process in the same environment whereas all other tools require a new instance of the Eclipse workbench. Only IMP schedules the transformation automatically after changing the DSL text; other tools require manual scheduling of the transformation. Finally, all tools use internal execution of the transformation.

5.3 Tool

Regarding tool assistance we observed a high variance ranging from a plain text editor for CS definition (*IMP*) to an editor with syntax coloring (*MontiCore*), code completion and validation while typing (*oAW* and *MPS*).

A well supported template editor is also very important. Support for editing templates range from using an existing editor without special template support (*MontiCore*), to an editor with comprehensive template support (*oAW* and *MPS*). The latter provides syntax highlighting, code completion and validation.

6. Conclusion

In this paper, we have described four different tools supporting textual DSLs. Furthermore, we compared these tools by a set of criteria based on the feature model of Langlois et al. [9]. The reuse of the criteria facilitates comparison of results presented in this paper with other comparisons, for example the comparison of graphical DSL tools given in [9] and [24].

The feature-set of the resulting language workbenches, mainly the editor, ranges from a plain text editor to a full-featured editor with syntax coloring, code completion and validation while typing.

All tools except *MPS* generate language workbenches based on the Eclipse platform. This explains the commonalities of the resulting workbenches. *MPS* is unique in this comparison as its editor is cell based instead of free text used by the other tools.

References

[1] M. Mernik, J. Heering, A. Sloane. When and How to Develop Domain-Specific Languages, *ACM Computing Surveys*, vol. 37, no. 4, pp. 316-344, December 2005.

[2] D. Teichroew and E.A. Hersehey. PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems. *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 1, 41-48, 1977.

[3] D. Spinellis. Notable design patterns for domain-specific languages. *The Journal of Systems and Software* 56, 91-99, 2001.

[4] H. Prähofer, D. Hurnaus, C. Wirth, H. Mössenböck. The Domain-Specific Language Monaco and its Visual Interactive Programming Environment. *IEEE Symposium on Visual Languages and Human-Centric Computing*, Coeur d'Aléne, Idaho, USA, September 23-27, 2007.

[5] W. Hargassner, T. Hofer, C. Klammer, J. Pichler, G. Reisinger. A script-based testbed for mobile software frameworks. In *Proceedings First International Conference on Software Testing, Verification, and Validation*, pp. 448-457, 2008.

[6] M. Fowler. *Language Workbenches: The Killer-App for Domain Specific Languages*?. 2005. http://martinfowler.com/articles/languageWorkbench.html.

[7] H. Mössenböck. *Coco/R for various languages – Online Documentation*. http://www.ssw.uni-linz.ac.at/Research/Projects/Coco.

[8] T. Parr and R. Quong. ANTLR: A Predicated-LL(k) Parser Generator. In *Journal of Software Practice and Experience*, 25(7):789-810, July, 1994.

[9] B. Langlois, C.E. Jitia, E. Jouenne. DSL Classification. In 7th *OOPSLA Workshop on Domain-Specific Modeling*, 2007.

[10] N. Wirth. What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions? *Communications of the ACM*, November 1977.

[11] *Open Architecture Ware*, http://www.openarchitectureware.org.

[12] P. Friese, S. Efftinge, J. Köhnlein. *Build your own textual DSL with Tools from the Eclipse Modeling Project*. 2008. http://www.eclipse.org/articles/article.php?file=Article-BuildYourOwnDSL/index.html.

[13] L. Angyal, L. Lengyel, H. Charaf. A Synchronizing Technique for Syntactic Model-Code Round-Trip Engineering. *Engineering of Computer Based Systems, 2008. ECBS 2008. 15th Annual IEEE International Conference and Workshop on the* Engineering of Computer Based Systems, pp.463-472, April, 2008.

[14] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, T. Grose. *Eclipse Modeling Framework*. Addison-Wesley, August 2003.

[15] *Meta Programming System*. http://www.jetbrains.com/mps.

[16] M. Fowler. *A Language Workbench in Action – MPS*. 2005. http://martinfowler.com/articles/mpsAgree.html.

[17] *MontiCore* http://www.monticore.de.

[18] H. Grönninger, H.Krahn, B. Rumpe, M. Schindler, S. Völkel. MontiCore: A Framework for the Development of Textual Domain Specific Languages. In *Proceedings of the 30th International Conference of Software Engineering (ICSE)*, Leipzig, Germany, 2008.

[19] H. Krahn, B. Rumpe, S. Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In 7th *OOPSLA Workshop on Domain-Specific Modeling*, 2007.

[20] H. Krahn, B. Rumpe, S. Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In *Proceedings of the ACM/IEEE 10th International Conference of Model Driven Engineering Languages and Systems* (*MODELS*), Nashville, TN, USA, 2007.

[21] *IDE Meta-Tooling Platform*. http://www.eclipse.org/imp.

[22] *The LALR parser generator* (*LPG*). http://sourceforge.net/projects/lpg.

[23] JET, http://www.eclipse.org/modeling/m2t/?project=jet.

[24] T. Özgür. Comparison of Microsoft DSL Tools and Eclipse Modeling Frameworks for Domain-Specifc Modeling. *Master Thesis, Software Engineering, MSE-2007:07, Bleking Institute of Technology*, Sweden, 2007.

	oAW	MontiCore	MPS	IMP
Language				
LA-AS1	ASG	AST	ASG	AST
LA-AS2	ECore meta-model	Java Classes	Proprietary meta-model.	Java Classes
LA-AS3	Composition.	Composition.	Composition.	No built-in support.
LA-CS1	Explicit. CS definition mixed with AS definition.	Explicit. CS definition mixed with AS definition.	Explicit. For each element of AS editor layout of CS is defined.	Implicit. CS defines AS.
LA-CS2	Text.	Text.	Text. Model instance stored in XML file.	Text.
LA-CS3	Declarative or imperative depends on decision of DSL developer.			
Transformation				
TR-TA1	Text. Template engine creates text files.	Text. Template engine creates text files.	Model or Text. Generator creates model instance. That in turn can generate text.	No built-in support.
TR-TA2	Destructive. Overwrites TA.	Destructive. Overwrites TA.	Destructive. Overwrites TA.	No built-in support.
TR-TA3	No integration support available.			
TR-OT1	M2T using template engine (xPand).	M2M. Using visitor pattern. M2T. Using visitor pattern or template engine.	M2M. Model to model generator. M2T. Using a target language that generates text (e.g. BaseLanguage generates Java Code).	No built-in support.
TR-OT2	Interpretation. Templates filled at runtime.	Interpretation. Templates filled at runtime.	Interpretation. Templates filled at runtime.	Compilation if using visitor pattern.
TR-OT3	External. Runtime workbench must be launched.	External. Runtime workbench must be launched.	Internal. Editor and Transformation in MPS.	External. Runtime workbench must be launched.
TR-OT4	Explicit. Workflow triggered by user.	Explicit. Triggered by user.	Explicit. Triggered by user.	Implicit. Eclipse builder runs after change.
TR-OT5	Internal. Runs in runtime workbench.	Internal. Runs in runtime workbench.	Internal. Runs in MPS.	Internal. Runs in runtime workbench.
TR-OT6	Manual. Triggered by user.	Manual. Triggered by user.	Manual. Triggered by user.	Automatically after change.
Tool				
TO-RA1	Depends on DSL. Our example DSL is seamless.			
TO-AS1	Adaptive. Code completion. Validation.	Adaptive. Validation.	Adaptive. Code completion. Validation.	Adaptive. Validation.
TO-AS2	Neither step nor workflow process guidance is supported.			
TO-AS3	Completeness. Using constraint language. Consistency. Ensured by grammar validation.	Completeness. Needs implementation. Consistency. Ensured by grammar validation.	Completeness. Using constraint language. Consistency. Grammar validation.	Completeness. Needs implementation. Consistency. Ensured by grammar validation.

Figure 13. DSL tool criteria comparison.