A MDE APPROACH FOR LANGUAGE ENGINEERING

Francisco Gortázar, Abraham Duarte, Micael Gallego
Department of Computer Science, Universidad Rey Juan Carlos, Tulipán, Móstoles (Madrid), Spain
francisco.gortazar@urjc.es, abraham.duarte@urjc.es, micael.gallego@urjc.es

Keywords: Model Driven Engineering, Language Engineering, abstract syntax, concrete syntax, DSLs

Abstract: Many development tools of modern Integrated Development Environments (IDEs) make an intensive use of abstract syntax tree (AST) representations of the software. This is the case of refactors, code formatters, or content assistants, among others. Such AST is usually an instance of an object oriented abstract syntax model. We propose to center the attention of Language Engineering (LE) on this model. We propose to use UML as the abstract syntax metamodel because UML tools provide code generators for different programming languages for model implementation. As well as an abstract syntax, a concrete syntax of the language it is also necessary. We are concerned about textual languages, whose concrete syntax is usually given as a BNF grammar. Instead, we propose to stereotype the abstract syntax model by means of a profile, aimed at concrete syntax definition. Applying Model Driven Engineering (MDE) practices several development artifacts can be automatically generated.

1 INTRODUCTION

Development tools in modern IDEs rely on the ASTs of the program (Boshernitsan, 2001; Clark et al., 2004; Herranz and Nogueira, 2005). An AST is a representation of the program that is being edited. Some development tools that are based on ASTs to perform their tasks are refactorings, design patterns extractors, call graph visualizers, type hierarchy visualizers, or content assistants, among others. Some IDEs with AST-based tools are *Eclipse*[1], *NetBeans*[2], or *IntelliJ IDEA*[3].

ASTs are generally object oriented, and conform to the abstract syntax model (Vainsencher and Black, 2006; Jones, 2003). The abstract syntax model represents the abstract syntax of the language. We propose to make the abstract syntax model the central piece of LE. We then follow a MDE approach to generate different development tools. MDE is a Software Engineering methodology focused on the integration of bodies of knowledge by different research communities (Favre, 2004). The MDE approach is strongly based on *models*, and model-to-model *transformations* that drive the application generation.

A well-suited modeling language for abstract syntax model definition is UML. UML is a standard modeling language for object oriented modeling, and there are several tools which support it. Furthermore, UML tools provide code generators which can generate code implementing the models for different programming languages, such as Java, C#, and C++.

Languages consist on an abstract syntax and a concrete syntax. There are two kind of concrete syntaxes: graphical and textual. We are concerned about textual syntaxes. Thus, our proposal is aimed at generating development tools for textual languages. The concrete syntax is usually represented by means of some form of a BNF context-free grammar. These grammars are used in reference manuals as the *de-facto* standard for defining the concrete syntax of languages (Paakki, 1995).

Some advantages of using BNF grammars in language definitions follows:

- Context-free grammars, and concretely BNF

[1]http://www.eclipse.org
[2]http://www.netbeans.org
[3]http://www.jetbrains.com/idea/

grammars, are well-known by language engineers.

- Context-free grammars have a long tradition on compiler construction, they also have solid underlying theories, and there is a large amount of documentation on the topic (Blasband, 2001).
- There is a large amount of automatic tools aimed at compiler construction, such as lexer and parser generators.

However, grammars as a concrete syntax definition formalism also present some drawbacks:

- Duplication of information. The structure of the language is represented in both the grammar and the abstract syntax model. It is error-prone to maintain both specifications synchronized.
- Structures of the grammar need to be transformed into their counterparts in the abstract syntax model. For instance, lists and boolean properties in the abstract syntax have different representations in the grammar.

In order to take advantage of MDE methodology for Language Engineering, some authors propose a metamodel for context-free grammar definition (Wimmer and Kramler, 2005; Muller et al., 2006; Fondement et al., 2006). Grammars are then defined by means of models which conforms with such metamodel. However, this approach does not solve the problem of keeping in-sync abstract and concrete models.

Instead, we propose to stereotype the abstract syntax model by means of a UML profile (Gortázar et al., 2007). This profile is aimed at concrete syntax definition. Our approach is compatible with grammar metamodels, because models based on that metamodels can be automatically generated from the annotated abstract syntax model. Although defining in the same model abstract and concrete syntaxes limits the language to a unique concrete syntax, having more than one concrete syntax for a language is unusual. In contrast, this unification of both definitions into one model allows to keep in-sync both syntaxes. It is possible to use one single model because some constructions of both domains are equivalent, like lists, or inheritance, among others (Wimmer and Kramler, 2005; Alanen and Porres, 2003; Antoniol et al., 2003; Hedin and Magnusson, 2003; Lieberherr, 2005; Wile, 1997).

In this paper we propose to apply MDE for automatic generation of development tools from a specification given in UML, thus bridging Model Driven Development and Language Engineering. For this purpose, we propose a new concrete syntax specification to be used within a model-driven approach.

2 ABSTRACT SYNTAX TREES

Abstract syntax represents the structure of the language, present in every computer language (Boshernitsan, 2001; Clark et al., 2004; Herranz and Nogueira, 2005). An AST represents the structure of a program as a tree. ASTs hide syntactic details like reserved words or punctuation symbols. The most common ASTs are object oriented (Jones, 2003). In modern IDEs ASTs are a central repository, and development tools rely heavily on them (Figure 1).

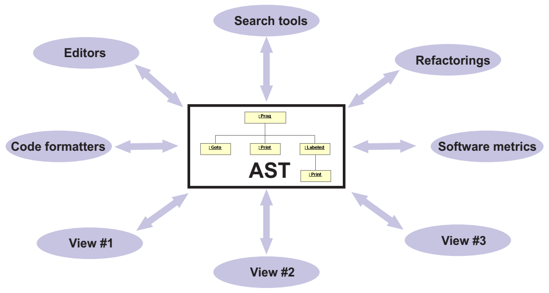


Figure 1: AST dependencies in modern IDEs.

The importance of ASTs in modern IDEs causes in some situations to start the language specification modeling the abstract syntax, and defining separately the concrete and abstract syntaxes. Some examples are SableCC[4], MPS, XMF-Mosaic, among others.

AST quality is gaining importance as a result of the open architecture of IDEs, as long as third parties can contribute development tools to the IDE by means of plug-ins. These plug-ins interact with the AST to perform their tasks. Thus, the abstract syntax model has to be comprehensible and easy to use (Bloch, 2006).

Figure 2 shows package dependencies of three Eclipse plug-ins for development support in three different programming paradigms (imperative, object-oriented, and functional). Dependencies shown in model a) correspond to the EclipseFP project, which is a framework that provides Haskell support in Eclipse. The `Halamo` subpackage, in the `Core` package, contains the Haskell abstract syntax model.

Dependencies shown in model b) correspond to the Java Development Tools (JDT)[5], which is a set of plug-ins that support Java programming in the Eclipse environment. The `DOM` subpackage, in the `Core` package, contains the Java abstract syntax model.

[4]SableCC 3.0: http://sablecc.org
[5]http://www.eclipse.org/jdt

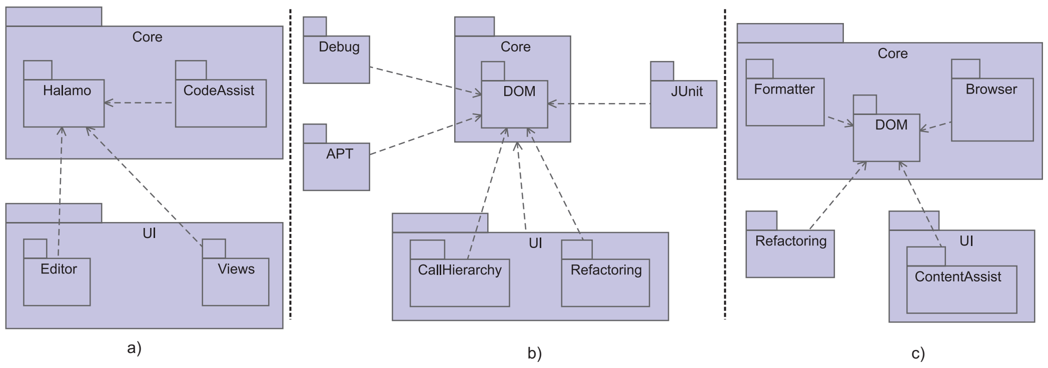


Figure 2: Some packages dependencies from different development tools.

Dependencies shown in model c) correspond to the C/C++ Development Tooling, which provides C/C++ support in the Eclipse environment. The `DOM` subpackage, in the `Core` package, contains the C/C++ abstract syntax model.

As it can be seen in Figure 2 most development tools included in the three plug-ins rely on the abstract syntax model. This is the case of editors, refactors, content assistants, or code formatters, among others.

3 MODEL DRIVEN LANGUAGE ENGINEERING: METACET'S APPROACH

Our proposal is based on the abstract syntax model of the language, as it is an essential part of development tools. We propose a methodology, called MetaCET, for textual language design and automatic generation of development tools. This methodology applies MDE principles to language development. MetaCET is based on modeling the abstract syntax of the target language in UML. The concrete syntax is provided by means of a UML profile aimed at this task, called *Concrete Syntax* profile. This profile is described elsewhere (Gortázar et al., 2007).

We call the resulting model the *language model*. From this model several tools can be automatically derived aimed at development in the target language. Figure 3 shows graphically the general approach of MetaCET.

During the rest of this section, we will use as an example the Statechart language, as defined in (Fondement et al., 2006). Our intention is to obtain a parser for such language, by means of applying MDE principles to language engineering. It follows an explanation of each step.

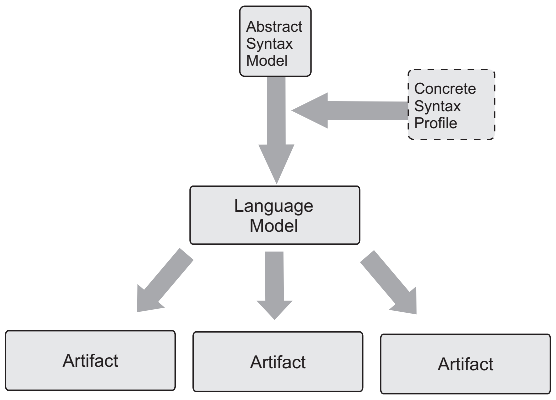


Figure 3: An overview of the approach.

3.1 Modeling the Abstract Syntax of the Target Language

Given the importance of ASTs in modern IDEs, we propose a Language Engineering approach which is focused on the abstract syntax model. We use UML as the abstract syntax metamodel. Our intention in doing so is to take advantage of code generation usually available within UML tools. Furthermore, UML is the *de facto* standard modeling language for object oriented modeling.

Basically, any abstract syntax model contains language concepts, represented as UML classes or interfaces. Relations between concepts are represented as UML associations. Finally, basic properties of concepts are represented as attributes with basic data types. Figure 4 presents an example, taken from (Fondement et al., 2006), of the abstract syntax of a statechart language. This language will be used to illustrate our proposal.

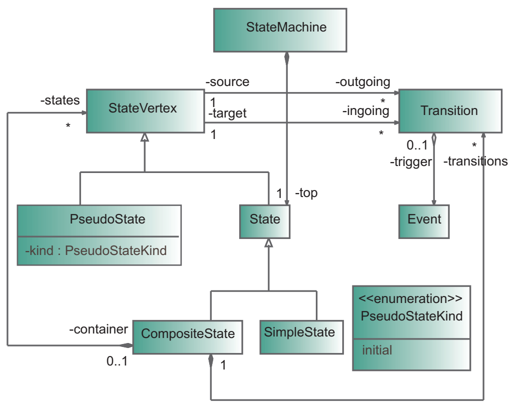


Figure 4: Fragment of an abstract syntax model.

3.2 Modeling the Concrete Syntax of the Target Language

The concrete syntax of a language defines the notation used to write or build documents in such language. There are two kind of notations: textual notations and graphical notations. In this paper we are concerned with textual notations.

We propose to stereotype the abstract syntax model with the concrete syntax. This stereotyping is performed by means of stereotypes defined in a UML profile we have defined: the *Concrete Syntax* profile.

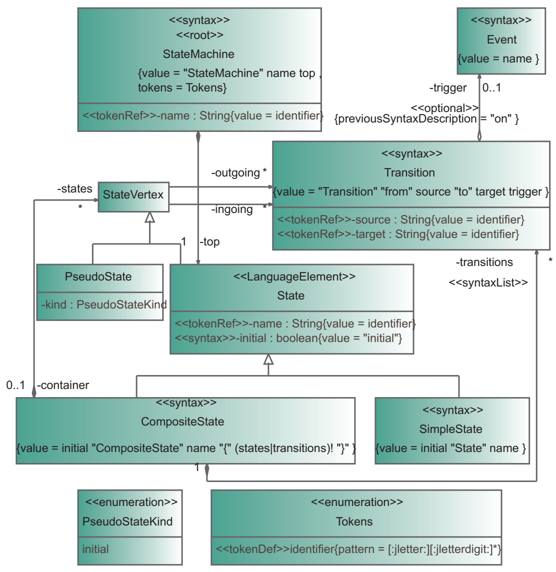
A UML profile is an extension mechanism provided by UML. It may contain information from a domain which is not supported directly by UML. A profile contains stereotype definitions which, when applied to elements of a UML model, provide the semantics of the domain.

In MetaCET, we provide a profile which contains the necessary stereotypes to augment an abstract syntax model with the concrete syntax. This stereotyped abstract syntax model is called the *language model* (see Figure 5).

Keeping abstract and concrete syntax separated might allow to define different concrete syntaxes for the same abstract syntax. However, this is not very usual. On the other hand, defining the language in a single model presents some advantages. Furthermore, there are several similarities between EBNF and object oriented modeling (Wimmer and Kramler, 2005). For instance, inheritance ↔ alternation (Wimmer and Kramler, 2005; Alanen and Porres, 2003), associations with a n upper bound multiplicity ↔ EBNF repetitions (Alanen and Porres, 2003), enumerations ↔ choice between static strings (Alanen and Porres, 2003), among others.

The EBNF grammar corresponding to the *language model* shown in Figure 5 is shown in Figure 6. The inheritance relationship between `State` and its subclasses `CompositeState` and `SimpleState`, is represented in the EBNF grammar as a choice rule (the `State` rule). Symbols on the left represent non-terminals, and symbols in bold represent literals.

Grammar details that cannot be represented directly in UML can be specified by means of the *Concrete Syntax* profile. This is the case of grammar terminals, and the arrangement of UML properties within the syntax definition of the class.



<<syntax>>
<<root>>
StateMachine
{value = "StateMachine" name top ,
tokens = Tokens}
<<tokenRef>>-name : String{value = identifier}
<<syntax>>
Event
{value = name }
-trigger
0..1
<<optional>>
{previousSyntaxDescription = "on" }
-states
*
StateVertex
-outgoing *
-ingoing *
<<syntax>>
Transition
{value = "Transition" "from" source "to" target trigger }
<<tokenRef>>-source : String{value = identifier}
<<tokenRef>>-target : String{value = identifier}
1
-top
-transitions
*
<<syntaxList>>
PseudoState
-kind : PseudoStateKind
<<LanguageElement>>
State
<<tokenRef>>-name : String{value = identifier}
<<syntax>>-initial : boolean{value = "initial"}
0..1
-container
<<syntax>>
CompositeState
{value = initial "CompositeState" name "{" (states|transitions)! "}" }
1
<<syntax>>
SimpleState
{value = initial "State" name }
<<enumeration>>
PseudoStateKind
initial
<<enumeration>>
Tokens
<<tokenDef>>identifier{pattern = [:jletter:][:jletterdigit:]*}

Figure 5: An annotated abstract syntax model for the StateChart language.

```
StateMachine   ::= **StateMachine** <ID> State
State          ::= CompositeState
                 | SimpleState
CompositeState ::= [**initial**] **CompositeState**
                   (State | Transition)
SimpleState    ::= [**initial**] **State** <ID>
Transition     ::= **Transition from** <ID>
                   **to** <ID> [**on** Event]
Event          ::= <ID>
```

Figure 6: EBNF for the StateChart language.

3.3 Tool generation

Several tools can be generated automatically from the *language model*. Usually such specifications are used to generate a parser. However, this is just one of a set of other possible tools. For instance, a call graph visualizer could be generated from this model. The visualizer obtains the information querying the AST.

Models for different tasks can be derived from the *language model*. These derived models can be tailored to specific domains such as design pattern detection, parser construction or software metrics. At the end of the process, it is necessary to generate code from the lowest levels. Code generators must be defined for this purpose. In this sense, low level models must be specific enough in terms of the technology to perform code generation.

Our proposal does not exclude other approaches. For instance, from the *language model*, a grammar models such as those defined in (Wimmer and Kramler, 2005) and (Alanen and Porres, 2003) can be automatically generated.

4 VALIDATION: A PARSER FOR AST CONSTRUCTION

In this section, a particularization of the methodology is presented. We have stated that ASTs are key in modern development tools such as those present in common IDEs. Therefore, we have chosen to validate our proposal by means of automatically generating a parser for AST construction from the *language model*. The parser takes a document in the target language and builds its corresponding AST.

Our proposal for parser generation is based on three different levels (Figure 7). Each level requires different knowledge. This separation in levels allows experts in different domains focus on different parts of the language engineering process. Finally, we build the parser using a parser generator or compiler-compiler. Parser generators provide their own language for parser specification, thus we need to transform the language model into the appropriate specification. We use model-to-model transformations to derive such specification.

4.1 First Level: Language Model

The first level is the specification of the language by means of the abstract syntax model annotated with the concrete syntax (the language model) as was described in Section 3. This model is independent on any kind of grammar. The model could even be hardly tractable by most common analysis methods. However, this is not a key aspect at this level.

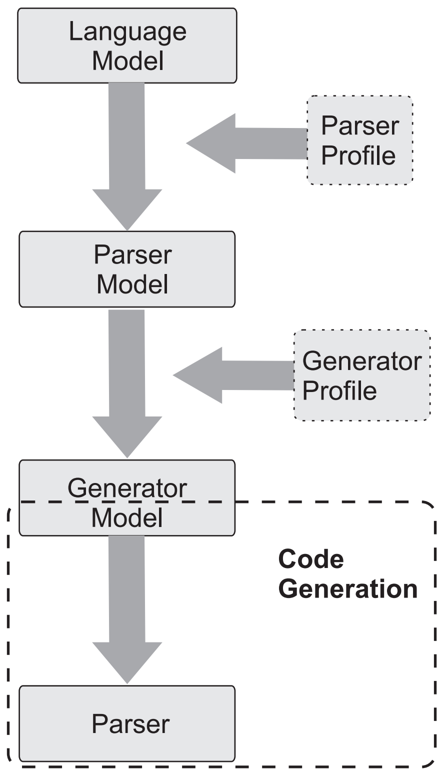


Figure 7: Parser development for a target language with the MetaCET's approach.

4.2 Second Level: Parser Model

The second level is the specification of a context-free grammar for the language. This grammar is represented by means of an object oriented model. This model is called *parser model*. It is not just a grammar model, because elements in the model are related with elements in the abstract syntax model.

The parser model is independent of the analysis method used. Common problems to various of these methods can be solved in this level. Thus, when models in the lower level are generated, they are free of such problems.

The parser model is obtained automatically from the language model by means of a model-to-model transformation. The transformation produces the parser model and annotates it with stereotypes from a profile, called *Parser*, we have defined. Our profile have the same intention as other approaches such as those of (Wimmer and Kramler, 2005; Muller et al., 2006; Fondement et al., 2006). In fact, those approaches could also be used as a result of our transformation.

4.3 Third Level: Generator Model

The third level is the specification of the *generator model*. This model is tailored for a concrete parser generator. The *generator model* is obtained automatically from the *parser model*. Two different model-to-model transformations have been defined. The first one produces a model for a JavaCC parser generator. The second one produces a model for a Cup[6] parser generator.

The generator model is based upon a concrete kind of context-free grammar: LL, LALR, etc. For instance, JavaCC requires a LL grammar. Cup, on the other side, requires a LALR grammar.

4.4 Code Generation: Parser Construction

Code generation is performed in two steps. First, from the generator model, a textual specification for the parser generator is produced. Second, the parser generator takes this textual specification and generates the parser. The aim of the parser is to parse documents written in the target language and to build their corresponding AST. This AST is an instance of the abstract syntax model.

5 CONCLUSION

Many development tools make an intensive use of abstract syntax trees. This is the case of refactors, code formatters, or content assistants, among others. Such AST is usually an instance of an object oriented model which represents the language's abstract syntax. In this paper we have proposed a Language Engineering methodology which is focused on this abstract syntax model. We choose UML as the abstract syntax metamodel because UML tools provide code generators for different programming languages for model implementation. Furthermore, UML is the *de facto* standard for object oriented modeling.

We have proposed to stereotype the abstract syntax model by means of a UML profile, called *Concrete Syntax*, aimed at concrete syntax specification. This stereotyped abstract syntax model avoids the synchronization between abstract and concrete syntax. From this stereotyped abstract syntax model several development artifacts can be automatically generated by means of applying MDE practices. We have given an overview of the generation of a common development component: a parser for AST construction.

[6]Cup: LALR parser generator for Java (http://www2.cs.tum.edu/projects/cup/)

ACKNOWLEDGEMENTS

This work has been partially supported by MCyT TIN2005-08943-C02-02 and URJC-CM-2006-CET-0603.

REFERENCES

Alanen, M. and Porres, I. (2003). A relation between context-free grammars and meta object facility metamodels.

Antoniol, G., Penta, M. D., and Merlo, E. (2003). Yaab (yet another ast browser): Using ocl to navigate asts. In *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*, page 13, Washington, DC, USA. IEEE Computer Society.

Blasband, D. (2001). Parsing in a hostile world. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, page 291. IEEE Computer Society.

Bloch, J. (2006). How to design a good api and why it matters. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming languages, systems, and applications*, pages 506–507. ACM Press.

Boshernitsan, M. (2001). Harmonia: A flexible framework for constructing interactive. Technical report.

Clark, T., Evans, A., Sammut, P., and Willians, J. (2004). *Applied Metamodelling: A Foundation for Language Driven Development*. Xactium.

Favre, J.-M. (2004). Towards a basic theory to model model driven engineering. In *3rd Workshop in Software Model Engineering (WiSME 2004)*.

Fondement, F., Schnekenburger, R., Gérard, S., and Muller, P.-A. (2006). Metamodel-Aware Textual Concrete Syntax Specification. Technical report.

Gortázar, F., Duarte, A., and Gallego, M. (2007). Representing languages in UML. In *Proceedings of the 2nd Conference on Evaluation of Novel Approaches to Software Engineering*.

Hedin, G. and Magnusson, E. (2003). Jastadd: an aspect-oriented compiler construction system. *Sci. Comput. Program.*, 47(1):37–58.

Herranz, A. and Nogueira, P. (2005). More than parsing. In Lpez Fraguas, F. J., editor, *Spanish V Conference on Programming and Languages (PROLE 2005)*, pages 193–202. Thomson Paraninfo.

Jones, J. (2003). Abstract syntax tree implementation idioms. In *Proceedings of the 10th Conference on Pattern Languages of Programs (PLoP'03)*.

Lieberherr, K. J. (2005). Object-oriented programming with class dictionaries. *LISP and Symbolic Computation*, 1:185–212.

Muller, P.-A., Fleurey, F., Fondement, F., Hassenforder, M., Schneckenburger, R., Gérard, S., and Jézéquel, J.-M. (2006). Model-driven analysis and synthesis of concrete syntax. In *MoDELS*, pages 98–110.

Paakki, J. (1995). Attribute grammar paradigmsa high-level methodology in language implementation. *ACM Comput. Surv.*, 27(2):196–255.

Vainsencher, D. and Black, A. P. (2006). A pattern language for extensible program representation. In *Proceedings of the Pattern Languages of Programming Conference*.

Wile, D. S. (1997). Abstract syntax from concrete syntax. In *ICSE '97: Proceedings of the 19th international conference on Software engineering*, pages 472–480, New York, NY, USA. ACM Press.

Wimmer, M. and Kramler, G. (2005). Bridging grammarware and modelware. In *MoDELS Satellite Events*, pages 159–168.