REPRESENTING LANGUAGES IN UML
A UML Profile 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: UML profile, UML metamodel, Language Engineering, abstract syntax, concrete syntax, DSLs.

Abstract: In this paper a UML profile for textual concrete syntax specification is described. The profile provides the necessary elements to associate the concrete syntax of a language L to an abstract syntax model of L. Such augmented abstract syntax model is called the language model of L. This language model avoids keeping the abstract and concrete syntaxes synchronized. We take advantage of the similarities between object oriented modeling and BNF-based language specification, and use a profile to specify the dissimilarities.

1 INTRODUCTION

UML is a general purpose modeling language, commonly used in object oriented development. Currently UML provides different notations for different parts of the development process, from use cases to deploy diagrams. One of the most known notations is the one centered in object oriented modeling.

However, UML has been used for other tasks different from software systems specification. For these tasks UML is too general, and UML profiles are used to tailor the model to a specific domain. Some UML profile examples are: real-time applications (Aldawud et al., 2003; Backus and Vallecillo, 2004; Apvrille et al., 2004), aspect oriented modeling (Aldawud et al., 2003), QoS (Cortellessa and Pompei, 2004; Asensio et al., 2001), agent systems modeling (Huget, 2004; Marcos and Pryor, 2003), requirements engineering (Heaven and Finkelstein, 2004), or XML-Schema (Provost, 2002; Carlson, 2001; Bernauer et al., 2004; Routledge et al., 2002), among others.

In this paper we present a UML profile for concrete syntax specification of textual languages. The choice of a UML profile is motivated by the fact that the abstract syntax is modeled in UML. Our aim is to annotate this abstract syntax UML model with the textual representation (concrete syntax) of elements in the model. In our proposal, language structure (abstract syntax) is provided by means of standard UML elements, and concrete syntax is provided by means of stereotypes of a UML profile that are applied to elements of the abstract syntax model.

2 ABSTRACT SYNTAX

We are concerned with automatic IDE generation. When aiding developers, development tools rely heavily on the abstract syntax tree (AST) of the program that is being developed. The AST is an instance of an abstract syntax model which in our case is defined with UML. As long as the abstract syntax needs to be implemented by means of a programming language, it is natural to express it with UML. For instance, from the UML abstract syntax model Java classes can be generated which represent the abstract syntax model in Java. This is the case of IDEs such as Eclipse[1], or NetBeans[2], among others.

In language definition, the same elements are used systematically (Wimmer and Kramler, 2005; Alanen and Porres, 2003; Antoniol et al., 2003; Hedin and Magnusson, 2003; Lieberherr, 2005; Wile, 1997). These elements, and their representation in the abstract syntax model are described in the rest of this

[1] http://www.eclipse.org
[2] http://www.netbeans.org

section. Although the modeling decisions presented here are general, we show such decisions using an sample language extracted from the literature (Fondement et al., 2006): the statechart language. Thus, we will show during the rest of the paper how to define the statechart language using the profile we have defined.

In Section 3 we will show how to apply the concrete syntax profile to these elements to specify the textual projection of the abstract syntax concepts.

2.1 Language Concepts

Classes and interfaces represent statechart language concepts like *state*, *state machine*, or *transition*. Language concepts expose an inner structure which is represented in the abstract syntax as associations and attributes of classes. These concepts usually pertain to some classification, which is expressed by means of inheritance relationships.

It follows a textual fragment of a `StateMachine` as defined in (Fondement et al., 2006). The fragment corresponds to a `CompositeState` definition:

```
CompositeState closed {
    initial State locked
    State unlocked
}
```

Figure 1 shows the fragment of the corresponding abstract syntax. Three different concepts appear in the abstract syntax model: `State`, which corresponds to an abstract state; `CompositeState`, which corresponds to a composite state (`closed` in the example); and `SimpleState` which corresponds to a simple state (`locked`).

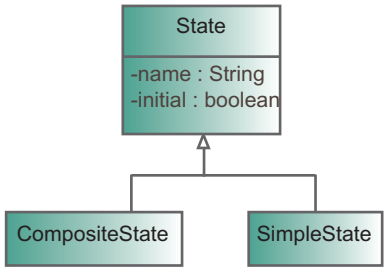

Figure 1: Language concepts.

2.2 Sequences

In most languages there are elements that are arranged into sequences. In a statechart there are several examples of elements arranged into sequences, like states and transitions. The language might impose restrictions on the bounds of a list. Commonly, the upper bound is undefined, but the lower bound is usually zero or one. For instance, a composite state in the statechart language might contain no inner states. Lists of elements are represented in UML as associations with a multiplicity of 0..*n*, or 1..*n*.

Given the previous statechart example, Figure 2 shows the abstract syntax fragment corresponding to the containment relationship between `CompositeState` and its inner states. A composition association between `CompositeState` and `StateVertex` is used to represent the containment relationship. The name of this association is `states`. As long as a composite state might not have inner states, the multiplicity of the association is 0..*n*.

Figure 2: Lists of elements.

2.3 Optional elements

Optional elements are those that might appear in a concrete position. These elements can be further divided into two groups:

- Elements associated with a true/false value. When they are present, they represent a true value, otherwise, they represent a false value. These elements are represented in the abstract syntax with a boolean property. In a state declaration such as `initial State unlocked`, a state is declared as the initial state. Figure 3 shows the corresponding abstract syntax representation of such modifier.

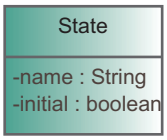

Figure 3: Optional elements of kind true/false.

- Elements associated with some structure. Consider, for instance, the event that triggers a transition. When a transition is triggered by an event, this is specified with the reserved word `on` followed by the name of the event that causes it. In the abstract syntax model these kind of elements are represented as an association with multiplicity 0..1. Figure 4 shows this example.

Figure 4: Optional elements of kind 0..1.

2.4 Tokens

Some elements such as identifiers, constants, etc, are represented as string properties in the abstract syntax model. Figure 3 shows a class that represents an abstract state. The name of the state is represented as the string property `name` , such as `locked` .

3 CONCRETE SYNTAX PROFILE

The concrete syntax is specified by means of annotating the abstract syntax. This annotation is performed by means of stereotypes defined in a UML profile (called *Concrete Syntax*) we have defined for this purpose. The stereotypes of the *Concrete Syntax* profile provide information of the textual projection of abstract syntax elements. The profile is presented by means of the application of the stereotypes to an abstract syntax model of the statechart language.

3.1 Language model

We propose to represent a language model as a model with stereotype ≪Options≫. The name tag of this stereotype is used to represent the language name (Table 1 and Figure 5).

Stereotype	Base classes	Tags
Options	Model	languageName

Tag	Type	Mult	Constraint
languageName	String	0..1	

Table 1: ≪Options≫ stereotype.

3.2 Classes

Classes in the abstract syntax model represent concepts of the language. These classes contain properties of one of the types defined in the previous section.

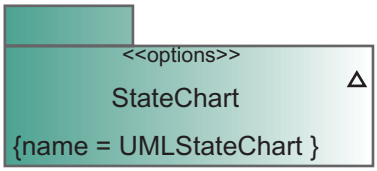

Figure 5: The StateChart UML model.

UML does not impose an order between these properties. However, the textual representation of the class requires an order in which the syntactic definitions of each property must be disposed. This is done by means of applying the ≪Syntax≫ stereotype to the class. The value tag of this stereotype allows defining tokens, property references, and their arrangement:

- *Tokens*: are specified inside apostrophes. A reference to a token which has been defined and has been named can be used enclosed in angular parentheses "<" and ">".
- *Property references*: are specified using the name of the property. The concrete syntax of the properties is obtained from their respective stereotypes.
- *Arrangement*: elements are considered in order from left to right.

A special arrangement can be specified when a set of elements can appear in any order. Inner states and transitions within a composite state are such a case. In the statechart language the following declarations are valid:

```
State state1
State state2
Transition from state1 to state2
State state3
```

The ≪Syntax≫ stereotype allows the specification of this kind of situations, with the ()! operator, used in conjunction with the | operator with the usual BNF semantics:

```
(elem1 | elem2 | ... | elemN )!
```

The semantics of the ()! operator consists of recognizing as many elements of the set `elem1...elemN` as possible, without taking care of their order (Figure 6).

3.3 Abstract classes and interfaces

It is common that elements of the abstract syntax model are related to each other by means of inheritance relationships. For instance, an abstract class might represent any state, and classes derived from it might

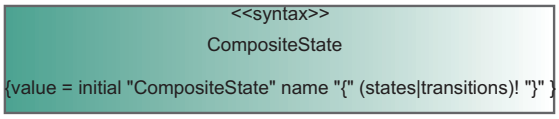

Figure 6: Unordered groups.

represent concrete state declarations (such as a simple state and a composite state). The abstract class is part of the conceptual model. However, it does not have a textual representation. We propose to represent abstract concepts with the ≪LanguageElement≫ stereotype.

Properties defined in these classes can still be used in the ≪Syntax≫ stereotype of subclasses. In Figure 7 the `name` property, although defined in the `State` class, appears in the ≪Syntax≫ stereotype of `CompositeState` .

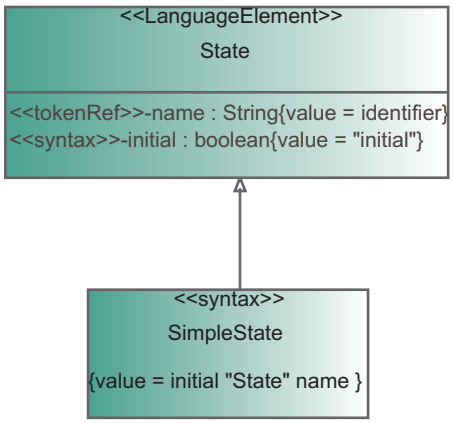

Figure 7: Referencing inherited properties.

3.4 The Root Class

There is a class in the abstract syntax model which represents the root of the AST. This is called the root element. We propose to annotate the root element with the ≪Root≫ stereotype (Table 2). Tags of this stereotype are aimed at token definition.

3.5 Token Definition

Tokens are the vocabulary of the language. There are different kinds of tokens, like reserved words, identifiers, constants, among others.

We propose to define tokens as enumeration literals of a special enumeration type. There is an enumeration literal for each token with a corresponding name. The ≪TokenDef≫ stereotype (Table 3) is applied to each enumeration literal. Tags of this stereotype are used to represent the token pattern, the scope information, the skip information, the token precedence and the action to be executed each time the token is recognized.

Stereotype	Base classes	Tags
Root	Class	tokens, macros, scopes, initialScope

Tag	Type	Mult	Constraint
tokens	Enumeration	0..1	
macros	Enumeration	0..1	
scopes	Enumeration	0..1	
initialScope	Enumeration Literal	0..1	!scopes.isEmpty() and scopes.contains(self)

Table 2: ≪Root≫ stereotype.

Stereotype	Base classes	Tags
TokenDef	EnumerationLiteral	pattern, skip, action, scopes

Tag	Type	Mult	Constraint
pattern	String	1..1	JFlex regexp
skip	Bolean	1..1	
action	String	0..1	Java code
scopes	Enumeration Literal	0..n	

Table 3: ≪TokenDef≫ stereotype.

Figure 8 shows the definition of a token for identifiers, represented as the enumeration literal `identifier` . The `pattern` tag contains the regular expression. This regular expression is given in JFlex[3] format.

[3]The Fast Scanner Generator for Java. http://jflex.de

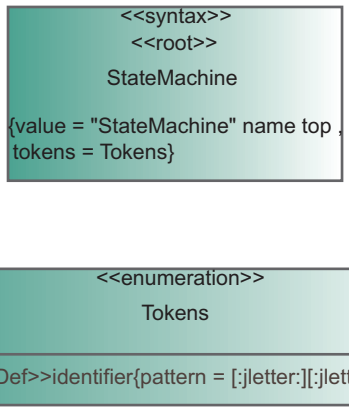

Figure 8: Token definition.

3.6 String Properties

String properties store identifiers, constants, and similar elements. To specify which kind of token is allowed for a given string property, we propose to stereotype the property with stereotype ≪TokenRef≫ (Table 4). The tag of this stereotype references the enumeration literal which corresponds to the kind of token accepted.

Stereotype	Base classes	Tags
TokenRef	EnumerationLiteral	token, scope

Tag	Type	Mult	Constraint
token	Enumeration Literal	1..1	self.isStereotypedWith(TokenDef)
scope	Enumeration Literal	1..1	self.isStereotypedWith(Scope)

Table 4: ≪TokenRef≫ stereotype.

Figure 9 shows the class `State` . This class contains a string property aimed at containing the state's name. The stereotype ≪TokenRef≫ is applied to this property, and its *value* tag references the enumeration literal corresponding to identifiers.

3.7 Boolean Properties

Boolean properties usually refer to some feature which might or might not be present. Their value usually depends on the presence of a token, or a

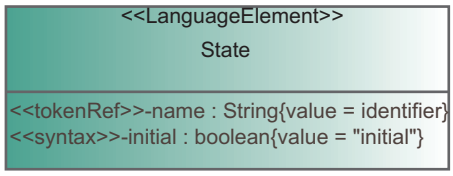

Figure 9: String properties.

set of tokens, in the document. Figure 9 shows the case of an initial state. An state is the initial state if the string `initial` appears just before the `State` reserved word. We propose to apply the stereotype ≪Syntax≫ to these boolean properties. (Table 5). The `value` tag of this stereotype contains the set of tokens which sets the value of the property. If these tokens are present, the property is set to true, otherwise, the property is set to false.

Stereotype	Base classes	Tags
Syntax	Class, Interface, Property, EnumerationLiteral	value

Tag	Type	Mult	Constraint
value	String	1..1	()! Operator allowed just when applied to classes

Table 5: ≪Syntax≫ stereotype.

3.8 Lists

Lists usually contain elements of the same type, sometimes with a separator. This separator might appear between each pair of elements, or at the and of each one. In Figure 10, the `transitions` property is stereotyped with the ≪SyntaxList≫ stereotype (Table 6). This stereotype contains a `separator` tag used to specify the separator (if any), which might be a token or sequence of tokens. The `endSeparator` tag holds a boolean value representing whether the separator must appear between each pair of elements or at the end of each element.

3.9 Optional elements

Optional elements are usually specified by means of an association with a zero lower bound. However, more information is sometimes needed. First, some

Stereotype	Base classes	Tags
SyntaxList	Property	separator, endSeparator

Tag	Type	Mult	Constraint
separator	String	1..1	Token sequence
endSeparator	Boolean	1..1	Token sequence

Table 6: ≪SyntaxList≫ stereotype.

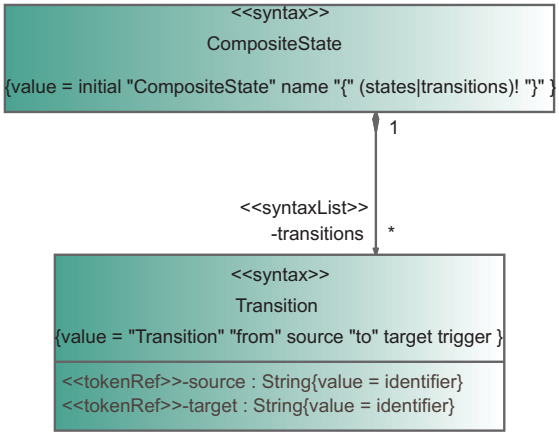

Figure 10: ≪SyntaxList≫ usage.

elements, when they are present, appear together with other elements. Second, some elements, when they are not present, are substituted by other elements. The following is an example of the first situation:

```
Transition from state1 to state2 on anEvent
```

In a transition, it is possible to indicate the event that causes it. In the statechart language, when this event is specified, it must be preceded by the `on` reserved word. It is necessary to provide a mechanism for specifying this information.

Figure 11 shows our proposal for the previous example. The `trigger` property is stereotyped with the stereotype ≪Optional≫ (Table 7). The `previousSyntaxDescription` tag contains the tokens that must appear before the element when it is present. There is also a `laterSyntaxDescription` tag which refers to the tokens that must appear just after the element.

It follows an example of the second situation (taken from the Java language):

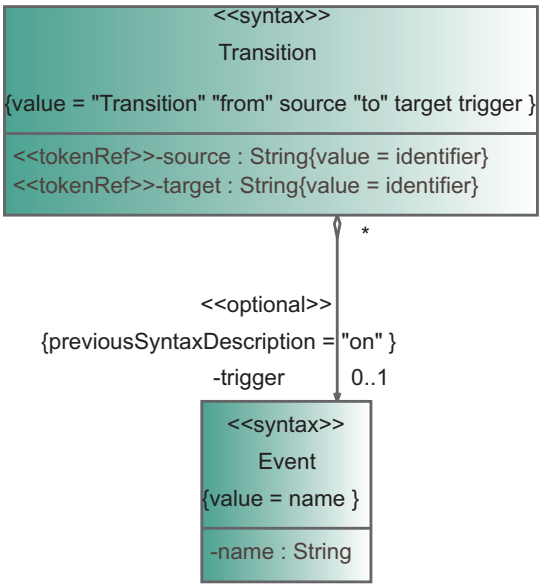

Figure 11: Optional elements.

Stereotype	Base classes	Tags
Optional	Property	previousSyntaxDescription, laterSyntaxDescription, alternativeSyntaxDescription

Tag	Type	Mult	Constraint
previousSD	String	0..1	Token sequence
laterSD	String	0..1	Token sequence
alternativeSD	String	0..1	Token sequence

Table 7: ≪Optional≫ stereotype.

```
public abstract void methodOne();
public void methodTwo() { ... }
```

A method declaration might have a body. If so, the body is enclosed in brackets. When there is no body, it is substituted by a semicolon. The `alternativeSyntaxDescription` tag could be used to define which tokens must appear instead of the body when it is not present.

4 CONCLUSION

We have shown how to represent languages in UML, such that representation expresses abstract and concrete syntaxes. The representation is based on a UML profile to convey the concrete syntax of languages to the object oriented model of the abstract syntax.

Evaluation of the profile has been done applying it to a statechart language as defined in (Fondement et al., 2006). This stereotyped model has been used to generate useful language support tools such as parsers and editors.

By means of unifying abstract and concrete syntax definition into a single model, we avoid the synchronization needed between the abstract syntax model and its corresponding concrete syntax specification. We exploit the similarities between both kind of artifacts, and express the dissimilarities by means of an UML profile. This profile conveys the concrete syntax information that cannot be directly expressed by means of UML.

Further work includes mapping known language specification formalisms equivalent to this one from and to this UML representation of a language. In particular we are interested in recovering grammar information for re-engineering and reverse engineering of grammar software.

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.

Aldawud, O., Elrad, T., and Bader, A. (2003). Uml profile for aspect-oriented software development.

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.

Apvrille, L., Courtiat, J.-P., Lohr, C., and de Saqui-Sannes, P. (2004). Turtle: A real-time uml profile supported by a formal validation toolkit. *IEEE Trans. Softw. Eng.*, 30(7):473–487.

Asensio, J. I., Villagr, V. A., de Vergara, J. E. L., and Berrocal, J. (2001). Uml profiles for the specification and instrumentation of qos management information in distributed object-based applications.

Backus, L. and Vallecillo, A. (2004). An introduction to uml profiles. *UPGRADE, The European Journal for the Informatics Professional*, 5(2):5–13.

Bernauer, M., Kappel, G., and Kramler, G. (2004). Representing xml schema in uml - a comparison of approaches. In *ICWE*, pages 440–444.

Carlson, D. (2001). Modeling xml vocabularies with uml.

Cortellessa, V. and Pompei, A. (2004). Towards a uml profile for qos: a contribution in the reliability domain. In *WOSP '04: Proceedings of the 4th international workshop on Software and performance*, pages 197–206, New York, NY, USA. ACM Press.

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

Heaven, W. and Finkelstein, A. (2004). Uml profile to support requirements engineering with kaos. *IEE Proceedings - Software*, 151(1):10–27.

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

Huget, M.-P. (2004). Agent uml notation for multiagent system design. *IEEE Internet Computing*, 8(4):63–71.

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

Marcos, C. A. and Pryor, J. (2003). Una extensin de uml para sistemas de agentes.

Provost, W. (2002). Uml for w3c xml schema design.

Routledge, N., Bird, L., and Goodchild, A. (2002). Uml and xml schema. In *ADC '02: Proceedings of the 13th Australasian database conference*, pages 157–166, Darlinghurst, Australia, Australia. Australian Computer Society, Inc.

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.