Aspect and XML-oriented Semantic Framework Generator: SmartTools

Didier Parigot, Carine Courbis, Pascal Degenne, Alexandre Fau
Joël Fillon, Isabelle Attali

INRIA Sophia-Antipolis - OASIS project
2004, route des Lucioles - BP 93
06902 Sophia-Antipolis cedex, France
First.Last@sophia.inria.fr

Abstract

SmartTools is a semantic framework generator, based on XML and object technologies. Thanks to a process of automatic generation from specifications, SmartTools makes it possible to quickly develop environments dedicated to domain-specific and programming languages. Some of these specifications (XML, DTD, Schemas, XSLT) are issued from the W3C which is an important source of varied emerging domain-specific languages. SmartTools uses object technologies such as visitor patterns and aspect-oriented programming. It provides code generation adapted to the usage of those technologies to support the development of semantic analyses. In this way, we obtain at minimal cost the design and implementation of a modular development platform which is open, interactive, uniform, and most important prone to evolution.

Key words: software generation, development environment, semantic analyses, aspect-oriented programming, visitor pattern, program transformation, XML, XSLT.

1 Introduction

In software applications, quality and ability to evolve, as well as development speed, are of major concern. Well-designed software can be quickly adapted to new requirements and technologies. It must also be able to exchange many varied data with other applications, particularly since the wide use of Internet.

The data structures are commonly defined with a DTD [1] (*Data Type Definition*) or a Schema from the World Wide Web Consortium (W3C), and

[1] No reference to W3C specifications (XML and DTD, Schema, DOM, XSL and XSLT, BML, SOAP) is given in this paper as they are easily available on the W3C web-site (`http://www.w3c.org`)

6 February 2002

exchanged with the XML (*eXtensible Markup Language*) format. These definitions are sort of abstract syntaxes of simple languages, named Domain-Specific Languages (DSL). For this large amount of new specific languages, there are needs for tools to handle treatments. All programming techniques can be applied to DSLs as they often have more simple syntaxes and semantics than the programming ones. As DSL designers and end-users may have no knowledge of these techniques (analysis, compilation, interpretation, etc), these tools should facilitate (hide) their uses. Additionally, these tools related to Internet applications need to be quickly developed, prone to evolution and integration, and easy to use.

The SmartTools platform fits in with these requirements. Its main goal is to help designers of domain-specific or programming languages to create new tools. No more than one specification (e.g. a DTD) is needed to quickly produce (generate) a dedicated development environment that contains a parser, a pretty-printer, a language-specific structure editor and a set of Java source files useful for semantics treatments (transformations, analyses). Both, SmartTools and the target environment are easy to use with a minimal knowledge and based on well-known techniques (e.g. visitor design pattern, aspect-oriented programming) or standard specifications (e.g. XSLT - *XML Stylesheet Language Transformation*). They have a modular and flexible implementation based on re-usable and generic components organized into a distributed architecture.

All the techniques and the generic components are tested on the internal languages of SmartTools. It is bootstrapped : about 40% of its source code is automatically generated. With its open architecture, it is very easy to plug in new components or interconnect other platforms, among which is .NET with the SOAP protocol.

The main innovation of SmartTools is to homogeneously gather many different technologies : XML technologies, component, oriented-object programming, visitor design pattern, and Aspect-Oriented Programming (AOP). This paper does not describe how these technologies are combined but rather why. It is made of two parts : the first one gives the reasons for using DTDs as input to define languages, and the visitor design pattern and aspects to specify semantics analyses ; the second one explains the choices about the architecture.

2 Semantic Tools

Internally, SmartTools uses extended and strongly typed abstract syntax (AST) definitions for all its tools. The important notions of these definitions are: 'operators' and 'types'. The operators are gathered into named sets: types. The sons of operators are typed and named. Figure 1 shows the definition of our

toy language: tiny [2]. For example, the *affect* operator belongs to the *Statement* type and has two sons: the first one is of type *Var* and the second one of type *Exp*.

```
Formalism of tiny is
Root is %Top;
  Top =           program(Decls declarationList, Statements statements);
  Decls =         decls(Decl[] declarationList);
  Decl =          intDecl(Var variable), booleanDecl(Var variable);
  Statements =    statements(Statement[] statementList);
  Statement =     affect(Var variable, Exp value),
                  while(ConditionExp cond, Statements statements),
                  if(ConditionExp cond, Statements statementsThen,
                                        Statements statementsElse);
  ConditionOp =   equal(ArithmeticExp left, ArithmeticExp right),
                  notEqual(ArithmeticExp left, ArithmeticExp right);
  ConditionExp =  %ConditionOp, true(), false(), var;
  ArithmeticOp =  plus(ArithmeticExp left, ArithmeticExp right),
                  minus(ArithmeticExp left, ArithmeticExp right),
                  mult(ArithmeticExp left, ArithmeticExp right),
                  div(ArithmeticExp left, ArithmeticExp right);
  ArithmeticExp = %ArithmeticOp, int as STRING, var as STRING;
  Exp =           %ArithmeticOp, %ConditionOp, var, int, true, false;
  Var =           var;
End
```

Fig. 1. the AST definition of *tiny*

From the AST definition, SmartTools can automatically generate a structured editor specific to the language. To facilitate the editing (to copy-paste nodes), it is useful to make the type inclusion possible.

We want, as much as possible, to use existing software components stemming from the W3C standards, such as the DOM (Document Object Model) API to handle XML documents. But, this latter API does not consider strongly typed structures. To manipulate strongly typed trees, we have extended it with the notions of fixed node, listed node and typed node (c.f. Figure 2). In this way, the tree consistency is guaranteed by the Java typechecker at its construction. For each operator, SmartTools automatically generates one class and the associated interface (Figure 3 shows the interface generated for the *affect* operator), and one interface by type. These classes contain the getters and setters needed to handle the sons (e.g. *getValueNode*, *setValueNode*).

It is important that the language designers can define their languages (abstract syntax) by using standard formats (DTD or Schema) proposed by the W3C and not necessarily with the internal AST definition format of SmartTools. Therefore, we have implemented conversion tools with some restrictions. For example, the notion of type does not explicitly exist within the DTD format i.e. the elements (seen as operators) do not belong to named sets. As this notion was essential, we had to define a type inference mechanism to convert DTDs. Additionally, the right part of element definitions should only contain parameter entity references to indicate the types of the sons (e.g. the line 6 of Figure 4 shows a DTD-equivalent definition of the *affect* operator). Unfortunately, few DTDs are written in this way. To be

[2] used all along this article

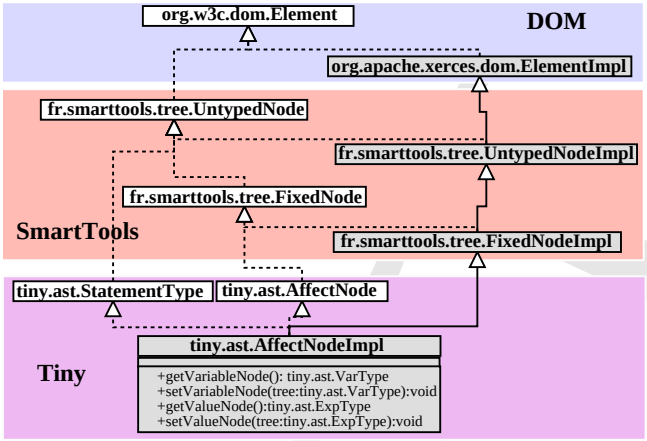

Fig. 2. Class hierarchy for the *affect* operator

```
package tiny.ast;
public interface AffectNode extends StatementType {
   public tiny.ast.VarType getVariableNode();
   public void setVariableNode(tiny.ast.VarType tree);
   public tiny.ast.ExpType getValueNode();
   public void setValueNode(tiny.ast.ExpType tree);
}
```

Fig. 3. Generated *affect* operator interface: *AffectNode*

able to accept as many as possible DTDs, a more complex type analysis (type inference) was carried out.

```
1  <!ENTITY % Top 'program'>
2  <!ENTITY % Statements 'statements'>
3  <!ENTITY % Statement 'if|while|affect'>
4  <!ELEMENT program  ((%Decls;), (%Statements;))>
5  <!ELEMENT statements  (%Statement;)*>
6  <!ELEMENT affect  ((%Var;), (%Exp;))>
```

Fig. 4. Part of the generated DTD of *tiny*

Moreover, we have implemented generators that produce a parser and the associated pretty-printer to manipulate programs with a more readable format than the XML one. For this purpose, the designer has to provide extra attributes information on each element (or operator) definition (see attributes in Figure 5). This possibility is useful for designers that do not have expertise on how to write a parser and makes sense only for small and unambiguous languages.

```
affect(Var variable, Exp value)
   with attributes {fixed String S1 = "=",
                    fixed String styleS1 = "kw",
                    fixed String AO = ";",
                    fixed String styleAO = "kw"}
```

Fig. 5. Extra data of the *affect* operator useful for generating a parser and the associated pretty-printer

Figure 6 shows all the specifications that can be generated from an AST specification:

- the API of the language (i.e. one class and the associated interface by operator, and one interface by type),
- the basic visitors useful for creating semantic analyses,
- a parser for the language (if extra syntactic sugars are provided as operator attributes in the language definition),
- a pretty printer to unparse ASTs according to these extra syntactic sugars,
- a minimal resource file that contains useful information for the structured editor and the parser,
- the DTD or the Schema.

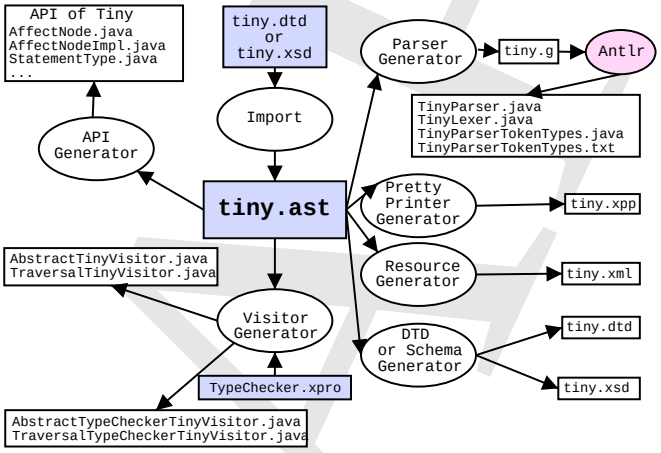

Fig. 6. All the specifications generated from an AST

For example, thanks to these tool generators, the *tiny* environment (Figure ??) was automatically generated only from one AST specification (see Figure 1), one xprofile specification (see Figure 7) , and the type-checker visitor (100 Java lines).

Semantics

This sub-section presents ways to write analyses (e.g. a type-checker, an evaluator or a compiler) on programs by using the visitor design pattern. If the reader wants to have more details and explanations on this well-known methodology, he can refer to [3,8,7]. For instance, we present three extensions of the visitor pattern technique: v1 using reflexivity mechanism with profiled visits and tree traversal possibilities, v2 adding simple aspect-oriented programming, v3 splitting the tree traversal (visit method calls) and the semantic actions by using more complex aspects.

Reflexive visitors (v1)

To make the development of visitors based on the AST definitions easier, SmartTools automatically generates two visitor classes: *AbstractVisitor* and *TraversalVisitor*. The abstract visitor declares all the visit methods (one by operator). The *TraversalVisitor* inherits from the *AbstractVisitor* and implements all the visit methods in order to perform an in-depth tree traversal. This visitor can be extended and its visit methods refined (overridden) to specify an analysis.

Thanks to the *xprofile* specification language of SmartTools, it is possible to specify the visit signatures i.e. to generate visits with different names, return types, and parameters. The granularity of this personalization is at the (AST) type level. Figure 7 presents the *xprofile* specification of a type-checker for *tiny*. From this specification, the system automatically generates the two correctly-typed visitors (*AbstractVisitor* and *TraversalVisitor*). Only useful visit methods have to be overridden to implement the type-checker (see Figure 8 for the *affect* operator). The advantage of using profiled visits is to avoid casts and obtain more readable visitor programs.

```
XProfile TypeChecker;
Formalism tiny;
import tiny.visitors.TinyEnv;

Profiles
 Object check(%Top, TinyEnv env);
 Object check(%Decls, TinyEnv env);
 Object check(%Decl, TinyEnv env);
 Object check(%Statements, TinyEnv env);
 Object check(%Statement, TinyEnv env);
 String check(%Exp, TinyEnv env);
 String check(%ArithmeticOp, TinyEnv env);
 String check(%ConditionOp, TinyEnv env);
 String check(%ArithmeticExp, TinyEnv env);
 String check(%ConditionExp, TinyEnv env);
 String check(%Var, TinyEnv env);

Strategy TOPDOWN;
```

Fig. 7. Visit signatures of a type-checker for *tiny*

```
1  public Object check(AffectNode node, TinyEnv env) throws VisitorException {
2     String varName = node.getVariableNode().getValue();
3     String typeLeft = env.getType(varName);
4     String typeRight = check(node.getValueNode(), env); //visit the value node
5
6     if (typeLeft == null)
7        errors.setError(node, "This variable " + varName + " was not declared");
8     else {
9        if (!typeRight.equals(TinyEnv.ERROR) && (!typeLeft.equals(typeRight)))
10          errors.setError(node, "Incompatible types: " + varName + " is a" +
11                typeLeft.equals(TinyEnv.INT)?"int":"bool") +" variable");
12    }
13    return null;
14 }
```

Fig. 8. *Affect* visit of the type-checker

With the *xprofile* language, it is also possible to specify the tree traversal (from the starting node to the destination node(s)) of a visitor. Thus, only the nodes on the path are visited instead of all the nodes of the tree. It reduces the visitor runtime on sizeable trees and above all the size of the generated

visitors. A dependence graph analysis on the AST definition is performed to generate the corresponding abstract and traversal visitors with the 'right' visits according to the given path. For example with the traversal specified on Figure 9, only the visits of the *while* and *affect* operators and the visits of the operators contained between the root (*TOP*) and these operators (i.e *program*, *statements* and *if* according to the AST definition of Figure 1) will be called.

```
Traversal Essai:
   %Top -> while, affect;
```

Fig. 9. Traversal specification from the root (*TOP*) to *while* and *affect*

In SmartTools, we use the Java reflexivity mechanism to implement the visitor technique and not the classical solution of a specific method, usually denoted *accept*, defined on each operator[3]. Indeed, the introduction of a visitor profile prohibits from using this classical solution (*accept* method). A generic method (named *invokeVisit*) is executed when any visit method is called. The goal of this generic method is to invoke the 'right' visit method (with a strongly-typed node) by using reflexivity.

The use of reflexivity is runtime-expensive. To accelerate the invoke process, an indirection table is statically produced at compilation-time when the abstract visitor is generated. This table contains for each pair (operator, type) the Java reference to the visit *java.lang.reflect.Method* object to call. With this table, it is also possible to change the visit method name and to have different arguments. This solution is a simplification of the multi-method approach that dynamically performs the search of the best method to apply. We have compared these two approaches by using a Java multi-method implementation [2]. The performances are equivalent, but our approach is much easier to realize.

Visitors with Aspect (v2)

The reflexivity mechanism used to implement the visitor pattern technique makes the execution of additional code before or after the visit calls possible. In this way, a concept of aspect-oriented programming [4,6] specific for our visitors can be added without modifying the source code, unlike the first versions of AspectJ [1,5]. An aspect can be defined just by implementing the *Aspect* interface and then recorded (see methods on Figure 10) on any visitor. For example, if the aspect of Figure 11 is recorded on a visitor, it will trace out all the called visits.

Several aspects can be connected on a visitor. They are executed in sequence (according to the registration order). This connection (as well as the

[3] SmartTools can also help designers to develop this kind of efficient visitors. But, their codes are less readable (more casts, no aspect, no tree traversal choice, etc) than the v1 or v2 visitors . Therefore, we do not describe them in this article.

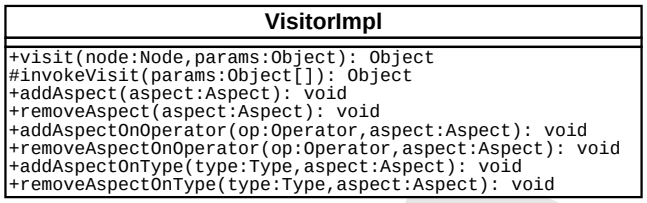

Fig. 10. Visitor with aspect (v2) API

```
package fr.smarttools.debug;
import fr.smarttools.tree.visitorpattern.Aspect;
import fr.smarttools.tree.Type;

public class TraceAspect implements Aspect {
   public void before(Type t, Object[] param) {
      System.out.println ("Start visit on " + param[0].getClass());
   }
   public void after(Type t, Object[] param) {
      System.out.println ("End visit on " + param[0].getClass());
   }
}
```

Fig. 11. Aspect that traces out the visit methods

disconnection) can be done dynamically at runtime. The behavior of a visitor can thus be modified dynamically by addition or withdrawal of these aspects. For example, a graphical debug mode for the visitors with a step-by-step execution was specified as an aspect regardless of any visitor. To add these aspects on the v1 visitors, the generic method (*invokevisit*) was extended.

Visitor with Tree Traversal and complex Aspects (v3)
With the concept of aspect-oriented programming, it is possible to split the tree traversal (visit method calls) and the semantic processing (semantic actions). Let us suppose that the visit code of the *affect(Var, Exp)* operator has this shape:

```
visit(AffectNode node ...) {
   codeBefore
   visit of the first son
   codeBetween1_2
   visit of the second son
   codeAfter
}
```

One can observe that the semantic part (i.e all except the recursive calls) is divided into N sons + 1 pieces of code. These N+1 pieces can be treated like aspects with new points of anchoring i.e before, between and after the visit method calls of the sons. We have defined a new visitor (named v3 visitor) that takes as arguments a tree traversal and one or more semantic actions (i.e. in the form of aspects) as shown on Figure 12. This visitor can call these aspects on these new points of anchoring. Therefore, these aspects must have for each operator, in addition to the traditional *before* and *after* methods, the *betweeni_i+1* methods (code to be executed between the i^{th} and $i+1^{th}$ sons). This new visitor can connect one or more aspects described in the v2 visitors. Figure 13 shows the type-checker semantics associated with the *affect* operator using this new form of aspect. There is no more recursive call unlike

the v1 (see Figure 8 line 4) or v2 visitors but it is necessary to use stacks (see Figure 13 lines 5 and 6) to transmit the visit results of the sons.

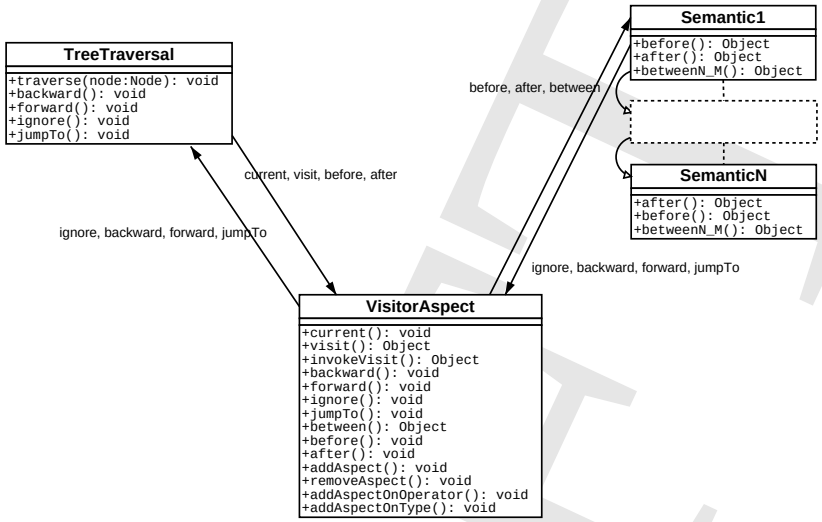

Fig. 12. v3 visitor

```
1  public void before(AffectNode node, Object param) {}
2  public void between1_2(AffectNode node, Object param) {}
3  public void after(AffectNode node, Object param) {
4     String varName = node.getVariableNode().getValue();
5     String typeRight = (String)typeStack.pop();
6     String typeLeft = (String)typeStack.pop();
7
8     same if code than Figure 8 (lines 6 to 12)
9  }
```

Fig. 13. Type-checker of the *affect* operator

The type-checker of *tiny* was extended with a initialization check on variables (see Figure 14) only by composing the two aspects (see Figure 15). The main interest of this programming style is to make the extension of analyses possible without modification only by adding new aspects. In this way, analyses are modular and re-usable. However, these analyses are more complex to program because of the splitting of the semantics and the tree traversal (compare Figures 13 and 8). Currently, we study how to share data between semantics, problems linked to the common tree traversal (e.g. what to do if one semantics wants to loop on a node and not the others?), ; we also study mechanisms to ease the programming of these aspects by hiding the stack management.

```
public void before(AffectNode node, Object param) {unplugVariableCheck = true;}
public void visit1(AffectNode node, Object param) {unplugVariableCheck = false;}
public void after(AffectNode node, Object param) {
   env.setInitialized(node.getVariableNode().getValue());
}
```

Fig. 14. Initialization check for the *affect* operator (v3 visitor)

```
TypeCheckerVisitor typeCheck = new TypeCheckerVisitor();
TinyEnv env = typeCheck.getEnv();
InitVarCheckerVisitor initVarCheck = new InitVarCheckerVisitor(env);
new Visitor(new LeftToRightTreeTraversal(),
            new Semantics[]{typeCheck, initVarCheck}).start(tree, null);
```

Fig. 15. Composition of two aspects

For the v3 visitor (see Figure 12), there is also a generic method that manages the next node to visit according to the current position, the tree traversal and some special traversal instructions. This method also copes with the search of the next method to call and the invocation of the v2 aspects on these visits.

3 Architecture

SmartTools is composed of independent software modules that communicate with each other by exchanging asynchronous messages. These messages are typed and can be considered as events. Each module registers itself on a central software component, the message controller (c.f. Figure 16), to listen to some specific types of messages. It can react to them by possibly posting new messages. The controller is responsible for managing the flow of messages and delivering them to their specific destination(s). The components of SmartTools are thus event-driven. This section presents the different modules of SmartTools and describes the behavior of the message controller.

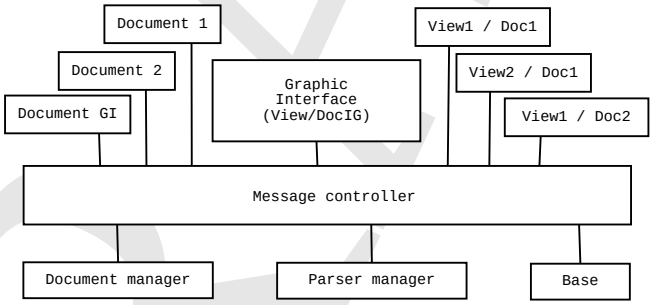

Fig. 16. Architecture of SmartTools

The main software modules of SmartTools are the following:

- Each **document** contains an AST. In Figure 16, *Document 1* and *Document 2* contain the ASTs on which the user is working. *Document GI* is a special one. It contains the AST describing the structure of the GUI (e.g. the AST of the Figure **??**).
- The **user interface** module manages the views, the menus and the toolbar of SmartTools.
- Each **view** is an independent module showing the content of a document in a format depending on the type of the view. For example, some views display the tree in colored-syntax text format, others as a graphical representation.

- The **parser manager** chooses the right parser to use for a file. Then, it runs the parser and builds the corresponding AST. The **document manager** uses this tree to build a document module and connects it to the message controller.
- The **base** is a module that contains definitions of resources used in SmartTools: colors, styles, fonts, menus, toolbars, actions, etc.

Of course, new types of modules can register themselves on the message controller. That is one of the ways to extend the features of SmartTools for a specific purpose or to embed SmartTools in another environment.

When a module needs to communicate with another module, it creates a message and posts it on the message controller. Then, the message controller broadcasts this message to the appropriate listeners (modules) that will react to it. Thus, modules that want to receive special types of messages from the message controller have to become listeners of these types of messages. They have to implement the *MsgListener* interface and provide a *receive(xxxMsg)* method for every type of supported message. Then, they have to register on the message controller (see code just below) and obtain their unique module identifier from it.

```
idDoc= msgControler.register(this);
```

XxxMsg in the *receive* method stands for the class of the expected message. Messages are typed objects i.e there is one specific class for every type of message. Their common behavior is held in one abstract class that is the super class of all the messages. New kinds of messages can be created by extending that common class or any other existing message class.

In the following example, the module expects to receive *SelectMsg*, *CloseDocMsg* and *CutMsg* messages sent to the module identified by *idDoc* and coming from an anonymous sender.

```
msgControler.addMsgListener("SelectMsg", idDoc, Msg.ANONYMOUS);
msgControler.addMsgListener("CloseDocMsg", idDoc,  Msg.ANONYMOUS);
msgControler.addMsgListener("CutMsg", idDoc,  Msg.ANONYMOUS);
```

Documents (i.e ASTs) and views are independently registered on the message controller. A document does not need to know how many views are related to it. When a modification is made, the document posts a modification message. The type of that message indicates which modification has been done and the message body contains the path of the modified node (from the root of the tree). For some kinds of messages, the change is also specified. Such messages will be sent only to the views that are registered to receive these modification messages coming from this document. Other modules will not receive them.

The message controller has a built-in message filtering capability. It is possible to write filters that watch or influence the flow of input and output messages on the controller. That filtering capability has been successfully used for several specific needs: benchmarking, debugging, undoing user actions, and automatically translating messages into another format (SOAP messages).

The architecture of SmartTools is designed to ease connection with other development environments or tools. Some experiments [9] are in progress to provide several features of SmartTools as web services and to use them from a client tool running on a .NET platform.

4 Conclusions

We have presented a software generator which produces programming environments strongly based on XML and object-oriented technologies. The most important contribution of this approach was to propose at the same time and with a uniform way, a set of advanced programming features, integrated into a modular architecture, with extensible graphical viewing engines and open to XML. We have chosen to use non-proprietary APIs to be open and to take advantage of future or external developments around W3C specifications. On the semantic level, we present a dedicated aspect-oriented programming approach associated with the visitor design pattern compliant with the DOM specifications. We expect a large set of domain-specific languages to be based on the W3C specifications. The users (and designers) of such languages are not supposed to be experts of language theories. Therefore, we propose a semantic framework easy to use and requiring a minimal knowledge. Domain-specific languages represent a large potential of applications in various fields and will certainly introduce new open problems.

Acknowledgments

We have much benefited from discussions with Colas Nahaboo, Thierry Kormann and Stéphane Hillion from the ILOG team on the topic of XML technologies. We would also like to thank Gilles Roussel, Etienne Duris and Rémy Forax for their helpful comments of their Java Multi-Methods implementation.

References

[1] Aspectj-oriented programming (aop) for java. http://www.aspectj.org.

[2] R. Forax, E. Duris, and G. Roussel. Java Multi-Method Framework. In *International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'00)*, Nov. 2000.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison Wesley, Reading, MA, 1995.

[4] G. Kiczales. Aspect-oriented programming: A position paper from the xerox PARC aspect-oriented programming project. In M. Muehlhauser, editor, *Special Issues in Object-Oriented Programming.* 1996.

[5] G. Kiczales, J. Hugunin, M. Kersten, J. Lamping, C. Lopes, and W. G. Griswold. Semantics-Based Crosscutting in AspectJ. In *Workshop on Multi-Dimensional Separation of Concerns in Software Engineering (ICSE 2000)*, 2000.

[6] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, New York, NY, June 1997.

[7] J. Palsberg and C. B. Jay. The Essence of the Visitor Pattern. In *COMPSAC'98, 22nd Annual International Computer Software and Applications Conference*, Vienna, Austria, Aug. 1998.

[8] J. Palsberg, B. Patt-Shamir, and K. Lieberherr. A New Approach to Compiling Adaptive Programs. In H. R. Nielson, editor, *European Symposium on Programming*, pages 280–295, Linkoping, Sweden, 1996. Springer Verlag.

[9] J. G. Variamparambil. Getting smarttools and visualstudio.net to talk to each other using soap and web services. Technical report, INRIA, 2001. http://www-sop.inria.fr/oasis/SmartTools/publications/Joseph/report.ps.