Aspect and XML-oriented Semantic Framework Generator: *SmartTools*[*]

Isabelle Attali, Carine Courbis, Pascal Degenne, Alexandre Fau, Joel Fillon, Christophe Held, Didier Parigot
INRIA Sophia-Antipolis - Oasis project
2004, route des Lucioles - BP 93
06902 Sophia-Antipolis cedex, France
First.Last@sophia.inria.fr

Claude Pasquier
Schlumberger - Advanced Research
68, Route de Versailles
78430 Louveciennes, France
Claude.Pasquier@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.

Keywords

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

1. INTRODUCTION

With new technologies related to data processing for Internet applications, the concept of language is more and more used to structure information. Therefore, the World Wide Web Consortium (W3C) has introduced new formalisms such as DTDs (Data Type Definitions) or Schemas that popularise the concept of abstract syntax, the basic component to manipulate any program. Additionally, the software quality and the development speed are of major concern in this particular application area. That justifies the creation of a software generator strongly based on XML (eXtensible Markup Language) and object technologies, named SmartTools.

The main goal of this software generator is to help designers of domain-specific or programming languages. No more than one specification (e.g. a DTD) is needed to quickly produce a dedicated development environment. Both, the target environment and the SmartTools framework must fulfil the following requirements:

- easy to use with a minimal knowledge and based on well-known techniques or standard specifications,
- modular and flexible implementation based on re-usable and generic components and a distributed software architecture,
- user-friendly thanks to a Graphic User Interface (GUI) that offers multi-views and an interactive environment,
- open thanks to a standard data exchange format used to communicate with its components and other external applications.

To ease the development of semantic analyses, several techniques have been introduced into SmartTools. First, the solution of "visitor pattern" [10] was largely automated with the generation of Java source code from abstract syntax definitions. Second, an aspect-oriented programming was added to obtain more re-usable semantic components. This new functionality does not require any program transformation. Thus, the addition of aspects on a visitor can be completely dynamic (without recompilation). Section 2 presents these semantic tools.

To meet with the architecture requirements, the modular software architecture of SmartTools was built around a central software component: the message controller. SmartTools is made of several independent software components that communicate with each other by exchanging asynchronous messages. The XML technologies are used to encode these messages. In section 3, the modular architecture of SmartTools is described.

[*]Supported by Schlumberger CP8, Microsoft Research and INRIA

Concerning the interactive requirements, SmartTools has an extensible and modular GUI with a set of pretty-printers or viewers strongly based on XML technologies. For data integration and to be open to new application fields, the XML format is used for all data exchange between components and as an description language for new applications. These interactive functionalities are presented in section 4.

About the re-usability requirement, SmartTools uses and provides several advanced software technologies stemming from various research works [6, 4, 13, 18, 14, 25] but homogeneously gathered together. In fact, web applications and the emergence of XML technologies are assets for a large diffusion and new application fields for this software generator.

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[1]. 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
```

Figure 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 type-checker 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*).

[1]used all along this article

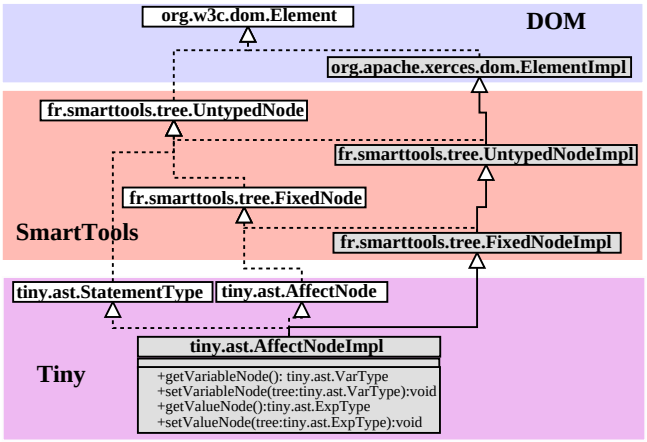

Figure 2: Class hierarchy for the *affect* operator

```
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);
}
```

Figure 3: Generated *affect* operator interface: *AffectNode*

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. Additionnaly, 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 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;))>
```

Figure 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) defintion (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"}
```

Figure 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.

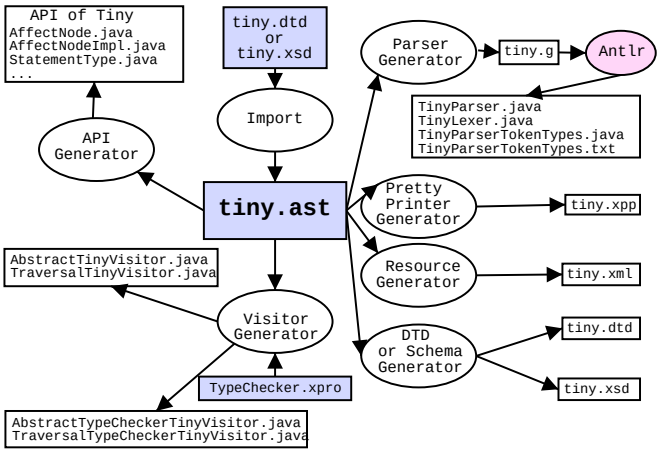

Figure 6: All the specifications generated from an AST

For example, thanks to these tool generators, the *tiny* environment (Figure 18) was automatically generated only from one AST specification (cf Figure 1), one xprofile specification (cf Figure 7) , and the type-checker visitor (XXX 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 [10, 24, 23]. 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.

Re¤exive 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 (cf 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;
```

Figure 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 }
```

Figure 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;
```

Figure 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[2]. 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 pait (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 [9]. 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 [15, 17] specific for our visitors can be added without modifying the source code, unlike the first versions of AspectJ [3, 16]. 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.

VisitorImpl
+visit(node:Node,params:Object): Object #invokeVisit(params:Object[]): Object +addAspect(aspect:Aspect): void +removeAspect(aspect:Aspect): void +addAspectOnOperator(op:Operator,aspect:Aspect): void +removeAspectOnOperator(op:Operator,aspect:Aspect): void +addAspectOnType(type:Type,aspect:Aspect): void +removeAspectOnType(type:Type,aspect:Aspect): void

Figure 10: Visitor with aspect (v2) API

[2]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.

```
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());
   }
}
```

Figure 11: Aspect that traces out the visit methods

Several aspects can be connected on a visitor. They are executed in sequence (according to the registration order). This connection (as well as the 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 (cf Figure 8 line 4) or v2 visitors but it is necessary to use stacks (cf Figure 13 lines 5 and 6) to transmit the visit results of the sons.

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

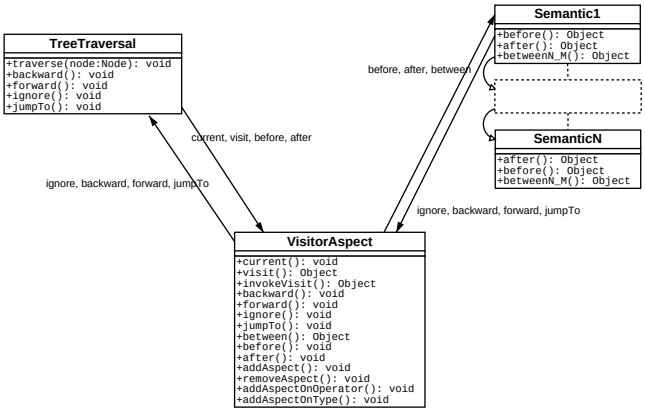

Figure 12: v3 visitor

```
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  }
```

Figure 13: Type-checker of the *affect* operator

```
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());
}
```

Figure 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);
```

Figure 15: Composition of two aspects

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.

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.

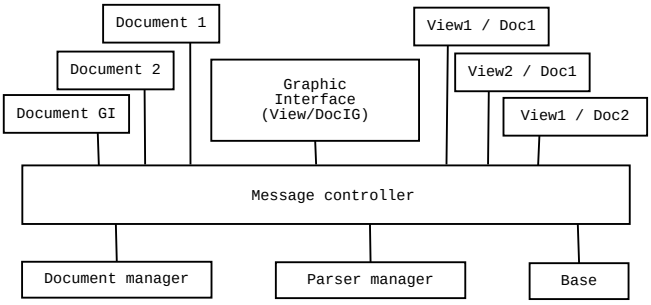

Figure 16: Architecture of SmartTools

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 23).
- 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 kind 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 in another format (SOAP messages).

The architecture of SmartTools is designed to ease connection with other development environments or tools. Some experiments [26] 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. GRAPHICAL USER INTERFACE

SmartTools has a GUI (c.f Figure 18) based on the document/views concept i.e. the user interface is the framework in which views on a document (AST) can be displayed and manipulated. For each open document, it is possible to build and display one or more views showing different aspects of the tree according to different formats. XML technologies are extensively used to build this GUI et the different views.

A view on a document is built by applying a transformation to its AST. We have experimented with two different approaches to perform tree transformations and build graphical views. The first approach was to write a visitor that transforms the tree and directly builds the hierarchy of graphical components. That was fast and efficient but required to recompile every time a change is done in the transformation. The second technique was to specify a tree transformation using XSLT to produce a BML description of graphical components to create. The BML result is then interpreted to build the actual view (see Figure 17). Even though there is a loss of efficiency when using XSLT and BML engines, the technique has proved to be easier to learn, more open to new view designs, and well-adapted for sending views through networks.

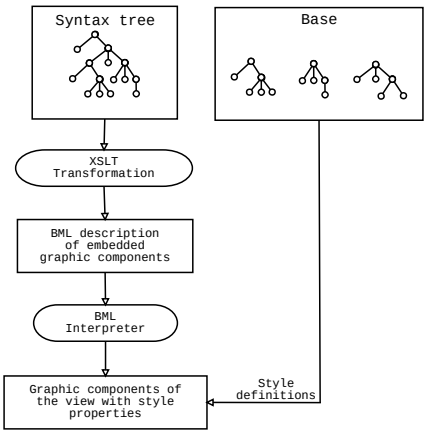

Figure 17: Schema of graphical view construction

Xpp language

A higher-level transformation language, called *Xpp*, has been defined on top of XSLT to specify the pretty-printing of an XML document. Its features are similar to those of XSLT but it is much more concise, more readable and it can perform transformations only on subtrees for incremental purposes. *Xpp* consists of a set of rule definitions (cf Figure 19) which match patterns with explicit variables for subtrees. These variables are used in the right part for recursive calls.

```
Rules
   formalism tiny
...
affect(x, y) -> h(x, label("="), y, label(";"));
plus(x, y) -> h(x, label("+"), y);
...
```

Figure 19: A part of the Xpp specification

We have defined formatting functions (horizontal or vertical alignment, indentation, etc) that designers may use to write their pretty-printers in the right part of the rules. When *Xpp* specifications are translated into XSLT stylesheets (see Figure 20), the designers only need to indicate the expected output format (either BML, HTML or text at the moment) useful for the system to choose the right implementation of the formatting functions (see Figure 21).

The *plus(x,y) -> h(x,label("+"),y); Xpp* rule specifies that the left and right subtrees for each *plus* operator will be horizontally aligned and separated by the + sign. The *h* and *label* formatting functions are defined in all the available output formats. *Xpp* can be extended by adding new formatting functions defined for every available output format.

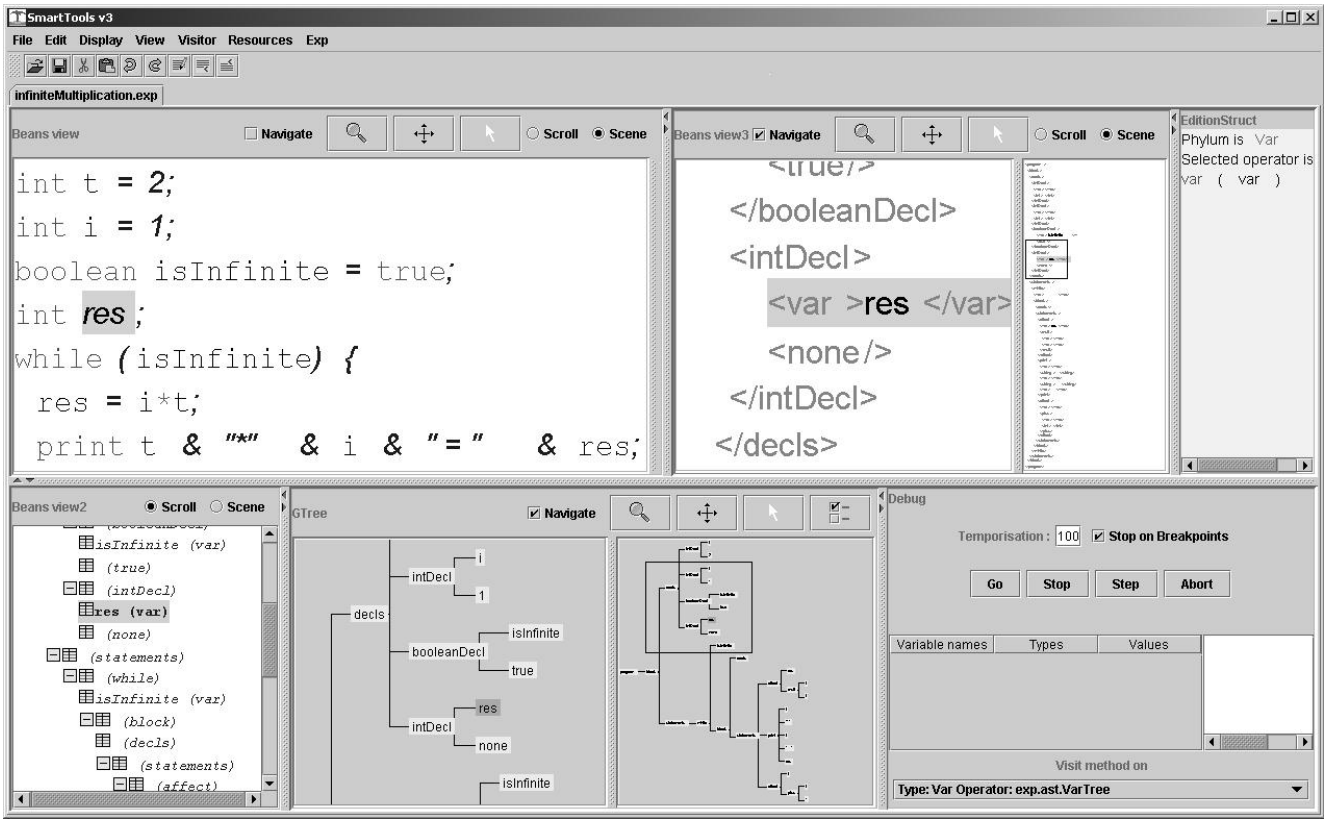

Figure 18: An example of Graphical User Interface

```
<alias:template match="plus[*[1]][*[2]][count(*)=2]">
     <alias:variable name="left" select="./*[1]"/>
     <alias:variable name="right" select="./*[2]"/>
     <bean class="fr.smarttools.view.GNodeContainer">
        <property name="layout">
          <bean class="fr.smarttools.view.HFlowLayout"/>
        </property>
        <add>
           <alias:apply-templates select="$left"/>
        </add>
        <add>
           <bean class="fr.smarttools.view.FJLabel">
              <args>
                 <string>+</string>
              </args>
           </bean>
        </add>
        <add>
           <alias:apply-templates select="$right"/>
        </add>
     </bean>
  </alias:template>
```

Figure 20: XSLT program for *plus* operator

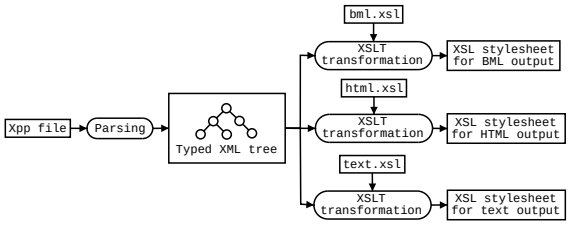

Figure 21: From Xpp to XSLT

Mapping between logical and graphical views

For BML output, every transformation rule specifies how to build a hierarchy of graphical components. Some of these components are associated with nodes of the tree and are marked so. Others are only syntactic sugars and are just ordinary graphical objects (not marked). This marking technique is a convenient way to be able to match any graphical object with its corresponding node in the document tree. When a part of the document tree is modified, an update message is sent to the views of that document. The update message contains the path of the modified subtree and the new subtree. Transformation rules are applied to that new subtree to create a local hierarchy of graphical components: a graphical subtree. The path contained in the update message is interpreted thanks to the marked components and the obsolete graphical subtree is found. It is then replaced by the new one to reflect the document tree modification.

The Base module

Definitions of style (fonts, colors, etc.) are stored in separate XML resource files that are managed by the Base module. When a view (or any other module) needs style information, the Base module uses visitors to find appropriate information in the resources (represented as ASTs). There are three successive search levels: first on a general resource tree, then on the current language-specific resource tree, and finally on the active view-specific resource tree. At every step, the result is overloaded by the newly found information.

GUI description language

A special XML language of SmartTools, called *lmltree*, was designed to describe the structure of the user interface. From such a description, SmartTools builds its user interface by transforming this description with the XSLT engine. The GUI is thus only a view of this description. Figure 22 shows such a description, Figure 23 the schematic graph of its AST, and Figure 18 the resulting GUI.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE layout SYSTEM "lml.dtd" >
<layout>
  <frame title="Smarttools V3">
    <set title="InfiniteMultiplication.exp">
      <split position="55" orientation="0">
        <split position="50" orientation="1">
          <view title="Beans view" Type="BmlView" style="default.xsl" />
          <split position="70" orientation="1">
             <view title="Beans view3" Type="BmlView" style="xml.xsl" />
             <view title="EditionStruct" Type="StructEditionView"
                 style="edstruct.xsl"/>
          </split>
        </split>
      <split position="25" orientation="1">
        <view title="Beans view2" Type="BmlView" style="generic.xsl" />
        <split position="60" orientation="1">
          <view title="GTree" Type="GtreeView" style="" />
          <view title="Debug Type="DebugView" style="" />
        </split>
      </split>
    </set>
  </frame>
</layout>
```

Figure 22: Lmltree specification of the GUI of Figure 18

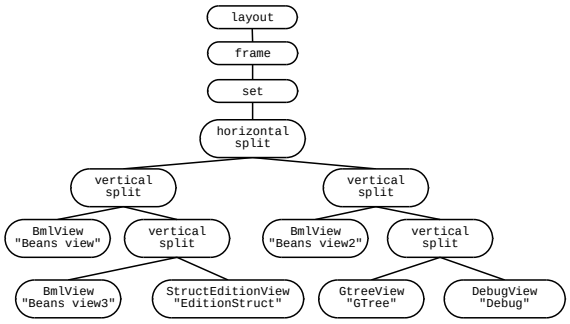

Figure 23: Schematic graph of the AST in Figure 22

5. APPLICATIONS

SmartTools has been used to develop or quickly prototype various environments of several languages. Its first applications were dedicated to the languages used by the system itself; it is bootstrapped. For instance, specific environments were created to edit the resources, to manipulate AST definitions or visit method profiles. Much more complex and powerful environments can be created with additional work.

Java

An integrated environment for Java [7] is currently under development. Figure 24 displays a source file (.java) and its associated class file (.class) on different formats (i.e. using different pretty-printers) as shown on Figure 25. These two documents are linked, thus the selection in one document is communicated to the other. The main tools of this environment are a bytecode type-checker and a bytecode simulator. All these tools use the visitor pattern technique and can be dynamically extended (e.g. with tracing or debugging features) simply by connecting aspects.

Figure 24: GUI of the Java environment

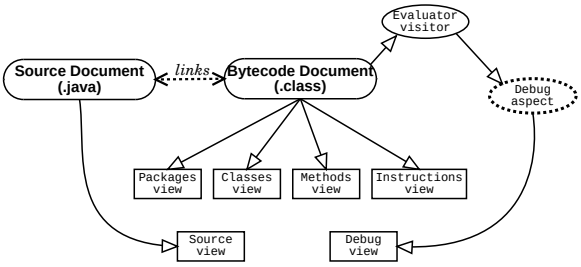

Figure 25: The different views of the Figure 24

Servlets and Web Services

As the SmartTools architecture was designed to easily plug new components, servlets can quickly be registered on the message controller. In this way, we have experimented a distributed version of SmartTools to edit programs on any applet-compatible web browser thanks to a Java applet. This applet was designed to visualize components expressed in BML and to handle user interactions. It uses the HTTP protocol to communicate with SmartTools through a servlet. A generalization of this experiment (Figure 26) was also performed using Web Services (i.e. units providing data and services to other applications). In this manner, applications can access to these web services via standard web protocols and data formats (e.g. XML, SOAP) without worrying about how the service is implemented.

SVG and MathML

We also wanted to display mathematical formulae in SmartTools views to be able to show Coq[3] [12] theories. As Coq can export its data in the MathML 2.0 format (see the HELM project [2] for further information), we needed to find a way to display this XML mathematical language. The solution was to use visitors to transform MathML documents into SVG[4] (Scalable Vector Graphics) format. Then, SmartTools uses the Batik [1] SVG renderer to display these SVG documents.

6. RELATED WORKS

[3]A proof assistant based on a explicite logic framework

[4]Another XML language used to describe two-dimensional vector and mixed vector/raster graphics

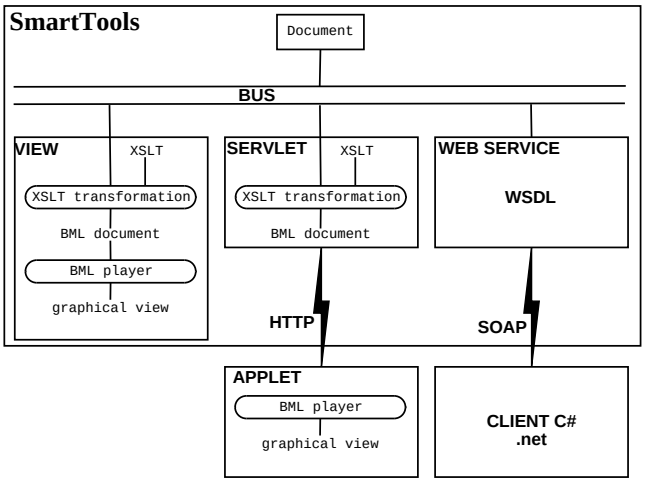

Figure 26: How to access to SmartTools

There are many equivalent or comparable systems [4, 6, 14, 18]. The main difference is that SmartTools strongly uses XML and object-oriented technologies. In this way, our system is open and can take advantage of any further development made around Java and XML technologies. It harmoniously integrates different tools and techniques (e.g. visitor design pattern, aspect) thanks to its modular architecture and has generic visualisation tools.

Our visitor approach is strongly based on this research work [23] and very close to other developments [11, 19, 21]. We essentially use a simplified version of the multi-methods [22, 9] instead of using *accept* methods. In this way, it is possible:

- to obtain much more readable visitor programs (i.e. without cast) thanks to the *xprofile* specifications,
- to get a simple kind of adapative programming [15, 20] dedicated to our applications thanks to the tree traversal specification,
- to introduce an aspect-oriented programming on the top of the visitor design pattern. Our approach is comparable with a more general one [3]. In SmartTools, aspects can be dynamically connected to visitors and no transformation is needed unlike [11].

For the modular architecture, we designed a message controller similar to the Toolbus [5] but it is restricted to our needs. For data integration[5], we use XML and for control integration a multicasting approach. With a minimal development effort, using existing software components (RMI API) or standard protocols (SOAP protocol), we have obtained a system where it is easy to:

- plug in new components,
- build a distributed environment in connection with a web browser or the .NET platform,
- transform it into a distributed version using ProActive [8].

For interactive requirements, our approach is different as we use XML technologies. Moreover, we apply the same transformation model for the document as well as for the GUI; that is quite an original way of building a GUI. This approach makes the export of views possible through the networks (thanks to XML serialisation).

7. 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.

8. REFERENCES

[1] Batik SVG Toolkit. Apache XML Project http://xml.apache.org/batik/.

[2] A.Asperti, L.Padovani, C. Coen, and I.Schena. Helm and the semantic math-web. In *The 14th International Conference on Theorem Proving in Higher Order Logics, TPHOLS 2001,*, volume 2115. Lect. Notes in Comp. Sci., 2001.

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

[4] D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: A Tool Suite for Building GenVoca Generators. In *5th International Conference in Software Reuse*, June 1998.

[5] J. Bergstra and P. Klint. The discrete time ToolBus – a software coordination architecture. *Science of Computer Programming*, 31(2-3):205–229, July 1998.

[6] P. Borras, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: the

[5]The terms *data integration* and *control integration* are explained in [5]

system. In P. Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, volume 24 of *SIGPLAN*, pages 14–24. Feb. 1988.

[7] D. Caromel, L. Henrio, and B. Serpette. Context Inference for Static Analysis of Java Card Sharing. In I. Attali and T. Jensen, editors, *Smart Card Programming and Security*, volume 2140 of *Lect. Notes in Comp. Sci.*, Cannes (France), September 2001. Springer-Verlag.

[8] D. Caromel, W. Klauser, and J. Vayssiere. Towards seamless computing and metacomputing in java. In G. C. Fox, editor, *Concurrency Practice and Experience*, volume 10 of *Wiley and Sons, Ltd*, pages 1043–1061, Sept. 1998.

[9] 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.

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

[11] G. Hedin and E. Magnusson. Jastadd—a java-based system for implementing front ends. In M. van den Brand and D. Parigot, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier Science Publishers, 2001.

[12] INRIA. The Coq proof assistant. http://coq.inria.fr/.

[13] M. Jourdan, D. Parigot, C. Julié, O. Durin, and C. Le Bellec. Design, implementation and evaluation of the FNC-2 attribute grammar system. In *Conf. on Programming Languages Design and Implementation*, pages 209–222, White Plains, NY, June 1990. Published as *ACM SIGPLAN Notices*, 25(6).

[14] U. Kastens, P. Pfahler, and M. Jung. The Eli system. In K. Koskimies, editor, *Compiler Construction CC'98*, volume 1383 of *Lect. Notes in Comp. Sci.*, portugal, Apr. 1998. Springer-Verlag. Tool demonstration.

[15] 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.

[16] 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.

[17] 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.

[18] P. Klint. A Meta-Environment for Generating Programming environments. *ACM Transactions on Software Engineering Methodology*, 2(2):176–201, 1993.

[19] T. Kuipers and J. Visser. Object-oriented tree traversal with jjforester. In M. van den Brand and D. Parigot, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier Science Publishers, 2001.

[20] K. J. Lieberherr and D. Orleans. Preventive program maintenance in Demeter/Java. In *Proceedings of the 19th International Conference on Software Engineering*, pages 604–605. ACM Press, May 1997.

[21] E. V. Merijn de Jonge and J. Visser. Xt: a bundle of program transformation tools. In M. van den Brand and D. Parigot, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier Science Publishers, 2001.

[22] T. Millstein and C. Chambers. Modular statically typed multimethods. In R. Guerraoui, editor, *Proceedings ECOOP'99*, LCNS 1628, pages 279–303, Lisbon, Portugal, June 1999. Springer-Verlag.

[23] 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.

[24] 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.

[25] T. Reps and T. Teitelbaum. The Synthesizer Generator. In *ACM SIGSOFT/SIGPLAN Symp. on Practical Software Development Environments*, pages 42–48. ACM press, Pittsburgh, PA, Apr. 1984. Joint issue with Software Eng. Notes 9, 3.Published as ACM SIGPLAN Notices, volume 19, number 5.

[26] 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.