An Eclipse Plug-in for Aspect-Oriented Bidirectional Engineering

Oscar Pulido-Prieto, Ulises Juárez-Martínez
Division of Postgraduate and Research Studies
Instituto Tecnológico de Orizaba
Orizaba, Veracruz, México
Email: opp026@gmail.com, ujuarez@ito-depi.edu.mx

Abstract—**This paper describes a plug-in for the IDE Eclipse, which enables the generation of AspectJ and CaesarJ code from a UML class diagram, as well the capability to perform reverse engineering from these languages to obtain a system model in a UML class diagram representation. In addition, the capability of generating an XML representation for visualization and understanding of AspectJ and CaesarJ code is provided. This plug-in also provides a graphical interface for system design through UML.**

Keywords-AspectJ; CaesarJ; UML; Software Engineering; Reverse Engineering.

I. INTRODUCTION

Aspect-oriented programming (AOP) [1] involves the development of modular components, which simplifies their reuse and maintenance. AOP solves encapsulation problems in object-oriented programming (OOP) [1] and structured programming [1]; such problems consist of an inability to encapsulate elements whose functionality covers more than a single object when these elements are not related through inheritance, composition, or aggregation.

Nowadays, there are several commercial tools for code generation; these cover the object-oriented paradigm and the aspect-oriented paradigm, but only partially for the aspect-oriented paradigm. This necessitates the use of extensions that are difficult for novice developers to understand. The inability to make equivalences between different aspect-oriented languages is another problem; it decreases interoperability and delays paradigm consolidation.

In this paper, a plug-in for the Integrated Development Environment (IDE) Eclipse [2] is presented. This plug-in adds the capability of generating code in AspectJ and CaesarJ from a model generated with a Unified Modeling Language (UML) extension through stereotypes, and generates a model from both languages, which comprises direct engineering. In addition, the plug-in generates an eXtensible Markup Language (XML) representation of both languages to improve the application design, which comprises reverse engineering. The combination of both direct and reverse engineering constitutes what is known as bidirectional engineering.

This paper is organized as follows: In Section 2, previous studies related to this work are presented. In Section 3, the plug-in architecture is described. In Section 4, the plug-in capabilities are analyzed. In Section 5, a discussion of the scope of this work is presented. In Section 6, a code generation example is provided. In Section 7, the conclusions are discussed. Finally, in Section 8, future work is described.

II. RELATED WORK

In [3], a plugin for an Eclipse Integrated Development Environment was presented. This plug-in generates AspectJ and CaesarJ code from an XML Metadata Interchange (XMI) document generated with a modeling tool. This plug-in was developed using the Eclipse Modeling Framework (EMF), a Model-View-Controller (MVC)-based architecture and a meta-model for AspectJ and another for CaesarJ. Both meta-models were defined through Java annotations, which belong to the core of EMF (Ecore) and transform Java interfaces into EClasses. This transformation occurs through the XQuery query language to search for items that are required in the XMI document. This plug-in is only focused on code generation from UML class diagrams; the system modeling is carried out with an external tool and is exported to the XML format, and the plug-in just generates code.

In [4], a tool that allows the representation of CaesarJ source code in XML format, simplifying the design of code generators and code analyzers, was presented. This tool generates XML documents based on a Java file containing CaesarJ code. The generated XML document is validated using an XML Schema (XSD). As an XML management Application Programming Interface (API), Java API for XML Processing (JAXP) was also used because this API supports several processing standards, allows transformations of XML documents, and XML Schema support is a standard part of JAXP. An analysis of the different approaches for creating XML documents was made, and one representing the elements as an abstract syntax tree was selected. As a result, a minimal tool for the generation of an XML document from CaesarJ source code and for obtaining source code for an XML document was developed.

In [5], BON-CASE, a Computer-Aided Software Engineering (CASE) tool for generating Java code, was presented. This was accomplished from a modeling diagram Business Object Notation (BON) language. This tool was developed to solve the need for a system that allows the generation of source code efficiently from a system abstract model, and also to generate robust and accurate implementations. The authors highlighted the existing CASE tool features, which allow tests to be carried out and code to

 ISBN: 978-1-61208-283-7

be generated, but with limitations in terms of robustness and analysis of the correctness of a model. BON-CASE is an extensible CASE tool for the BON modeling language focused on contracts and frame specifications; it generates Java source code. BON-CASE generates low coupling components to integrate these components in other platforms. BON-CASE offers a formal specification backup, code generation extensible templates, and a partially validated meta-model. The authors mentioned that UML is a modeling language that generates system-independent views. UML allows the use of formal techniques through Object Constraint Language (OCL). BON, on the other hand, is based on formal techniques that allow an inherent design by contracts. BON-CASE generates code in the Java Modeling Language (JML).

In [6], meta-programming was defined as the act of writing programs that generate other programs; this approach is essential for automated software development. On this basis, the authors developed Meta-AspectJ, a meta-language that extends Java code to generate AspectJ code. Meta-AspectJ is based on generative programming and aspect-oriented programming to design a specific domain aspect-oriented code generator, which generates efficient code to solve AspectJ limitations in general purpose code generators. The authors mentioned the disadvantages of domain-specific APIs: compatibility regressive problems when libraries are updated and bad interaction when these are independently modified. Meta-AspectJ solves these problems through the use of annotation that does not interfere with code execution and syntax.

In [7], an analysis of aspect-oriented framework architecture was performed. The authors proposed that there is a problem that results from the inability to encapsulate nonfunctional requirements as these are scattered throughout the system; they mentioned that the main problems of aspect-oriented architecture are the language type used and source transformations.

In [8], Aspectra, a framework designed to carry out test entry in previously generated aspects to measure reliability, was developed. The authors mentioned that the development of aspect-oriented software improves the software quality, but does not provide the desired accuracy owing to programmer mistakes, difficulty in verifying the appearance of an error during unit tests, or these aspects not being implemented directly.

In [9], the specifications of a meta-model for aspect-oriented modeling based on extension mechanisms using UML 2.0 and XMI, software implementation to facilitate use in other software tools, were presented.

In [10], methods of automatic code generation of aspect-oriented models by Theme/UML were defined. Additional requirements to develop a code generator were cited as follows: a meta-build system model and generator specification, which in turn is made up of snippets and production rules.

In [11], problems resulting from the use of frameworks in the design of an application due to the difficulty of demonstrating the design using UML were highlighted. To solve this problem; the authors proposed the creation of a framework called Aspect-Oriented Crosscutting Framework.

In [12], a method of aspect-oriented modeling, using the model-driven process that focuses on business applications, was described. The authors mentioned that there are no means of modeling crosscutting concerns in UML; this generates system designs with scattered artifacts; against this background, they proposed modeling crosscutting concerns in UML diagrams representing aspects, advice, and crosscuts as first-class models to produce an aspect-oriented model.

Table 1 shows a comparison of the contributions of each study described above. The target language and modeling language are shown, as well as the restrictions in terms of both design and programming. Among these studies, only the BON-CASE tool provides mechanisms for both direct and reverse engineering, but is limited to the Java language and the BON modeling language; the other works shown in Table 1 are design specifications. In the last row, our work is shown in order to highlight the current contribution with respect to previous studies.

III. PLUG-IN ARCHITECTURE

Eclipse is an IDE based on plug-ins that encourages the use of the Model-View-Controller architectural pattern.

In the Model layer, a meta-model through EMF is defined, as well as an XSD template and an eXtensible Stylesheet Language Transformations (XSLT) template for transformations between aspect source code and its XML representation. With EMF, a set of interfaces was generated to obtain a general meta-model and two specific meta-models for AspectJ and CaesarJ, as is shown in Figure 1(1).

In the Controller layer, the behavioral mechanism for the plug-in is defined. The controller has two parsers, which transform source code into an XML representation: one for AspectJ and the other for CaesarJ. This is possible by using an Abstract Syntax Tree, which decomposes the source code into nodes and then generates the XML representation, as is shown in Figure 1(2). On the other hand, an XSLT is used to parse the XML representation into source code. The controller also has an XMI analyzer to generate a Java model from the XMI document, as is shown in Figure 1(3); every class is tailored with one meta-model interface. Later, a template of Java Emitter Templates (JET) is used in order to generate source code from meta-model instances, either AspectJ or CaesarJ, as is shown in Figure 1(4).

In the View layer, a graphical interface to allow system modeling is defined, and then this interface is exported into source code or an XML representation through the XMI standard. The use of XMI furthermore provides an export and import mechanism to facilitate work with other tools. For generating a model representation, the plug-in takes source code in AspectJ and CaesarJ, and parses it for generating the XML representation. Then, an XMI document is generated with an XSLT and, finally, the model is loaded in the main window.

 ISBN: 978-1-61208-283-7

TABLE I. COMPARISON OF RELATED WORK

Author	Output Language	Modeling Language	Implementation	Independent Tool	Reverse and Direct Engineering
Rosas-Sánchez	Java, AspectJ, CaesarJ	UML	Yes	No	Direct
Salinas-Mendoza	Java, CaesarJ, XML	Not apply	Yes	Yes	Not applicable
Paige	Java	BON	Yes	Yes	Both
Huang	AspectJ	Java Annotations	No	No	Direct
Constantinides	Not apply	Not apply	No	No	Not applicable
Xie	Not apply	Not apply	Yes	Yes	Not applicable
Evermann	AspectJ	UML	No	Not apply	Not applicable
Hetch	Not apply	Theme/UML	No	Not apply	Not applicable
Júnior	AspectJ	UML	No	Not apply	Not applicable
Mosconi	Not specified	UML	No	Not apply	Not applicable
Pulido-Prieto	Java, AspectJ, CaesarJ	UML	Yes	No	Both

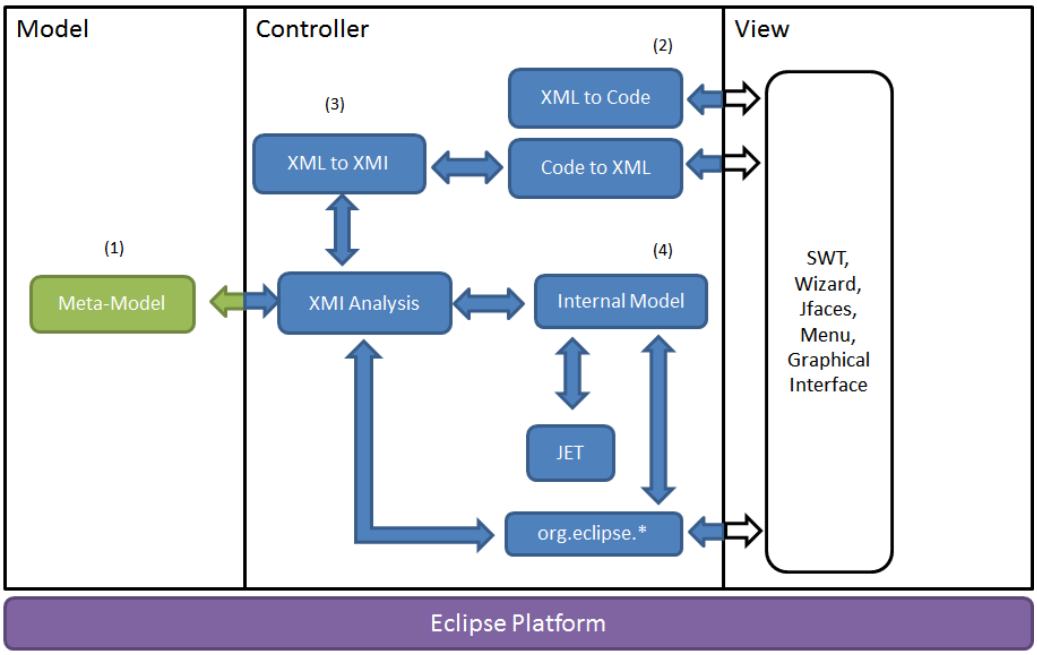

Figure 1. Plug-in logical architecture

In Figure 2, the plug-in's physical architecture is shown. The plug-in consists of five components, two of which are *parsers*, one for AspectJ and the other for CaesarJ; another one is the *view*, which allows the design of a system class diagram and generates the XMI representation or takes an XMI document and generates a class diagram; another is the *javamodel*, which allows an XMI document to be read, validated, and for source code to be generated through JET templates; and last, there is the *controller*, which allows communication among the other components.

IV. PLUG-IN CAPABILITIES

This plug-in allows the generation of aspect-oriented source code from UML class diagrams, to achieve a reverse engineering process in order to obtain the model from source code and generate an XML representation of it. Figure 3 shows how the graphic interface is implemented.

This project extends the previous work support [4]; in these work, a prototype for CaesarJ source code transformation into XML was developed. The support for bit operations has been added, as well as full support for signature patterns, support for more than two mixins, class element access was restructured, and anonymous class definition and class definition inside of blocks, such as classes, initializers, methods, iterations, or bifurcations, were added.

On the other hand, an AspectJ parser was developed, which works similarly to the project described in [4]. Besides, full support for signature patterns, bit operations, static crosscutting support, and error and warning declarations such as superclasses, interfaces, methods, and field inter-type declarations were added.

Additional work from a previous study was also incorporated [5]; it includes source code generation from a model. In addition, a graphical interface was implemented to avoid the external tool requirement for model generation; thus, the reverse engineering capability was also added, whereby the model is obtained from source code.

 ISBN: 978-1-61208-283-7

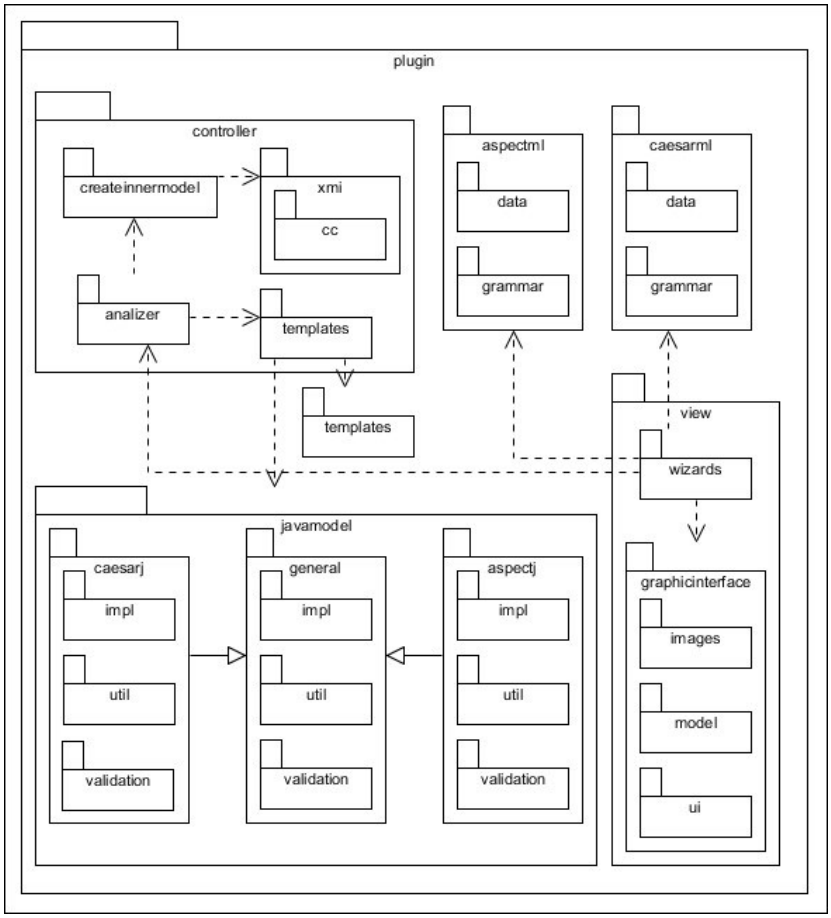

Figure 2. Plug-in physical architecture

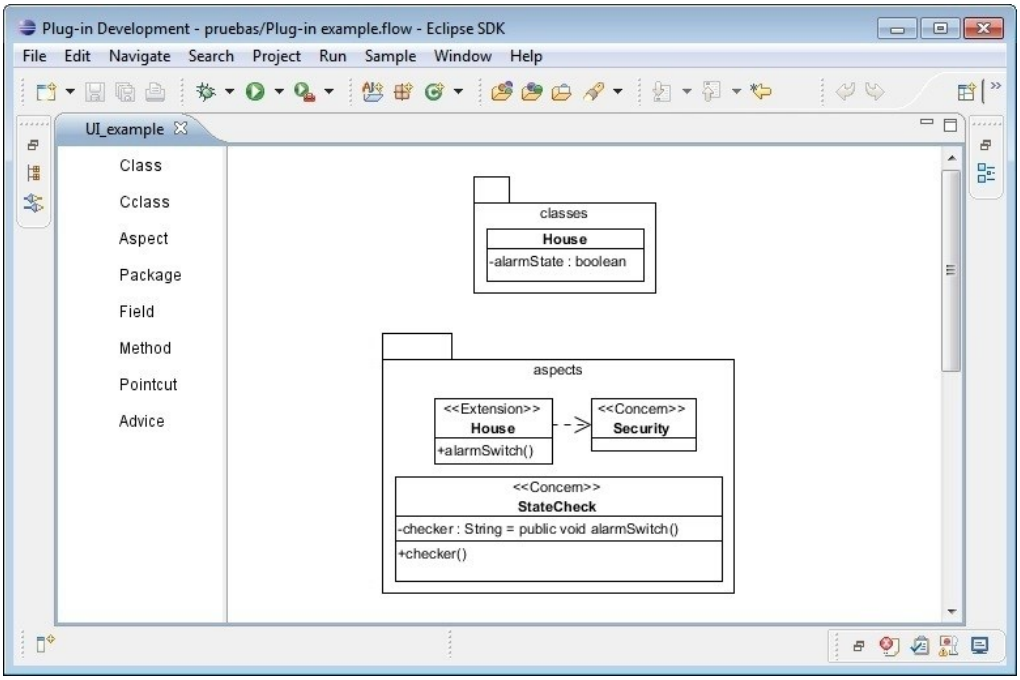

Figure 3. Plug-in graphical interface

 ISBN: 978-1-61208-283-7

Since a reverse engineering process is performed, this plug-in allows equivalences to be made between AspectJ and CaesarJ when source code is generated from a model. This is achieved with limitations in implementation in both languages, where CaesarJ lacks static crosscuttings and AspectJ lacks wrappers and mixins. If a model obtained from AspectJ implementation with static crosscuttings is used to generate CaesarJ source code, a wrapper is used to generate classes that have these static crosscuttings. When a model is obtained from CaesarJ implementation with wrappers and mixins, it uses an inner subclass of wrapper class for the wrapper. If a class has more than one superclass indicating a mixin composition, a warning message is shown, indicating modeling inconsistency.

V. DISCUSSION

With this plug-in, an XML representation of Java, AspectJ and CaesarJ is generated. In the cases of Java and AspectJ, this plug-in lacks support for generics, but uses enhanced *for* and *multi-catch* exceptions to improve the equivalences between AspectJ and CaesarJ. Another limitation is a lack of support to store and represent methods and initializer implementation; thus, when the reverse engineering process is realized, a code loss occurs since the abstraction level is higher. Method, constructor, and initializer implementation through activity diagrams is proposed, where the flux for each method, constructor, and initializer is modeled.

VI. EXAMPLE

In this section, a code generation example is presented, where the class *House*, with one field and without methods, is extended through a *Security* aspect, which provides one method for the alarm turn-on and turn-off, while another aspect called *StateCheck* provides a pointcut, which applies over the alarmSwitch() method. Modeling of this is shown in Figure 3.

For AspectJ, the *Extension* stereotype indicates that static crosscuttings will be added. The resulting source code is shown in Figure 4.

```
1  package aspects;
2  import classes.House;
3  public aspect Security {
4     public void House.alarmSwitch() {
5        //TODO Auto-generate code
6     }}
```

Figure 4. AspectJ source code for aspect Security.

An XML representation for the AspectJ code is shown in Figure 5, where line 3 is equivalent to line 1 in Figure 4, line 4 is equivalent to line 2 in Figure 4, line 5 is equivalent to line 3 in Figure 4, and lines 6 to 9 are equivalent to line 4 in Figure 4.

For CaesarJ, the *Extension* stereotype indicates a wrapper declaration. The resulting source code is shown in Figure 6.

An XML representation for the CaesarJ source code is shown in Figure 7, where line 3 is equivalent to line 1 in Figure 6, line 4 is equivalent to line 2 in Figure 6, line 5 is equivalent to line 3 in Figure 6, lines 6 to 8 are equivalent to line 4 in Figure 6, and lines 9 to 13 are equivalent to line 4 in Figure 6.

```
01  <aspect-source-program>
02     <aspect-class-file name="Security.aj">
03        <package-decl name="aspects"/>
04        <import module="classes.House"/>
05        <aspect name="Security" visibility="public">
06           <intertype-declaration target="House"
07              kind="method">
08              <method name=" alarmSwitch"
09                 visibility="public">
10                 <type name="void" primitive="true"/>
11                 <formal-arguments/>
12                 <block/>
13              </method>
14           </intertype-declaration>
15        </aspect>
16     </aspect-class-file>
17  </aspect-source-program>
```

Figure 5. XML representation for Security AspectJ code.

```
1  package aspects;
2  import classes.House;
3  public cclass Security {
4     public cclass SecurityWrappsHouse wraps House {
5        public void alarmSwitch() {
6           //TODO Auto-generate code
7        }}}
```

Figure 6. CaesarJ source code for aspect Security.

```
01  <java-source-program>
02     <java-class-file name="Security.java">
03        <package-decl name="aspects"/>
04        <import module="classes.House"/>
05        <cclass name="Security" visibility="public">
06           <cclass name="SecurityWrappsHouse"
07              visibility="public">
08              <wraps name="House"/>
09              <method name="alarmSwitch"
10                 visibility="public">
11                 <type name="void"
12                    primitive="true"/>
13                 <formal-arguments/>
14                 <block/>
15              </method>
16           </cclass>
17        </cclass>
18     </java-class-file>
19  </java-source-program>
```

Figure 7. XML representation for Security CaesarJ code.

For AspectJ, the *Concern* stereotype indicates an aspect, if a class with that stereotype contains a String field with a

crosscutting primitive, a pointcut is generated in the source code; if a method name is equal to this field, three definition of advice are generated to this pointcut, as is shown in Figure 8. For CaesarJ, the Concern stereotype indicates a *cclass*, as is shown in Figure 9. As is shown in Figure 9 and Figure 10, the only differences between the two languages are in line 2: the *aspect* keyword for AspectJ and the *cclass* keyword for CaesarJ are placed there.

```
01  package aspects;
02  public aspect StateCheck {
03     public pointcut checker() :
04        call(public void alarmSwitch());
05     before() : checker() {
06           //TODO Auto-generate code
07     }
08     void around() : checker() {
09           //TODO Auto-generate code
10     }
11     after() : checker() {
12           //TODO Auto-generate code
13     }}
```

Figure 8. AspectJ source code for aspect StateCheck.

```
01  package aspects;
02  public cclass StateCheck {
03     public pointcut checker() :
04        call(public void alarmSwitch());
05     before() : checker() {
06           //TODO Auto-generate code
07     }
08     void around() : checker() {
09           //TODO Auto-generate code
10     }
11     after() : checker() {
12           //TODO Auto-generate code
13     }}
```

Figure 9. CaesarJ source code for aspect StateCheckAspect.

An XML representation of an aspect with pointcuts is shown in Figure 10 for AspectJ and Figure 11 for CaesarJ. The only differences between the two XML documents is in line 4: a node aspect is placed there for AspectJ and a *cclass* node is placed there for CaesarJ. For Figure 10 and Figure 11, lines 5 to 13 are equivalent to lines 3 and 4 for Figure 8 and Figure 9, lines 14 to 21 in Figure 10 and Figure 11 are equivalent to line 5 in Figure 8 and Figure 9, lines 25 to 33 in Figure 10 and Figure 11 are equivalent to line 8 in Figure 8 and Figure 9, and lines 35 to 43 in Figure 10 and Figure 11 are equivalent to line 11 in Figure 8 and Figure 9.

VII. CONCLUSIONS

This paper presented a plug-in for AspectJ and CaesarJ code generation from a UML class diagram through their XMI specification. This plug-in applies reverse engineering to obtain a UML class diagram from source code, obtain an XML source code representation, or make equivalence between the two aspect-oriented languages. In addition, two previous studies were integrated and extended in order to obtain a functional tool for bidirectional software engineering; furthermore, the plug-in capabilities and limitations were presented.

```
01  <aspectj-source-program>
02     <aspectj-class-file name="C:\StateCheck.aj">
03        <package-decl name="aspects"/>
04        <aspect name="StateCheck" visibility="public">
05           <pointcut name="checker"
06              visibility="public">
07              <pointcut-arguments/>
08              <pointcut-expressions>
09                 <pointcut-expression designator="call"
10                    signature="public void
11                    alarmSwitch()"/>
12              </pointcut-expressions>
13           </pointcut>
14           <advice spec="before">
15              <formal-arguments/>
17              <pointcut-expressions>
18                 <pointcut-expression
19                    reference="checker">
20                    <arguments/>
21                 </pointcut-expression>
22              </pointcut-expressions>
23              <block/>
24           </advice>
25           <advice spec="around">
26              <type name="void" primitive="true"/>
27              <formal-arguments/>
28              <pointcut-expressions>
29                 <pointcut-expression
30                    reference="checker">
31                    <arguments/>
32                 </pointcut-expression>
33              </pointcut-expressions>
34              <block/>
35           </advice>
36           <advice spec="after">
37              <formal-arguments/>
38              <pointcut-expressions>
39                 <pointcut-expression
40                    reference="checker">
41                    <arguments/>
42                 </pointcut-expression>
43              </pointcut-expressions>
44              <block/>
45           </advice>
46        </aspect>
47     </aspectj-class-file>
48  </aspectj-source-program>
```

Figure 10. XML representation for StateCheck AspectJ source code.

VIII. FUTURE WORK

As future work, method and initializer body definition will be implemented through activity diagrams, and AspectJ

annotations and inclusion of generics for this language will be analyzed.

```
01  <java-source-program>
02    <java-class-file name="C:\StateCheck.java">
03      <package-decl name="aspects"/>
04      <cclass name="StateCheck" visibility="public">
05        <pointcut name="checker"
06          visibility="public">
07          <pointcut-arguments/>
08          <pointcut-expressions>
09            <pointcut-expression designator="call"
10              signature="public void
11              alarmSwitch()"/>
12          </pointcut-expressions>
13        </pointcut>
14        <advice spec="before">
15          <formal-arguments/>
17          <pointcut-expressions>
18            <pointcut-expression
19              reference="checker">
20              <arguments/>
21            </pointcut-expression>
22          </pointcut-expressions>
23          <block/>
24        </advice>
25        <advice spec="around">
26          <type name="void" primitive="true"/>
27          <formal-arguments/>
28          <pointcut-expressions>
29            <pointcut-expression
30              reference="checker">
31              <arguments/>
32            </pointcut-expression>
33          </pointcut-expressions>
34          <block/>
35        </advice>
36        <advice spec="after">
37          <formal-arguments/>
38          <pointcut-expressions>
39            <pointcut-expression
40              reference="checker">
41              <arguments/>
42            </pointcut-expression>
43          </pointcut-expressions>
44          <block/>
45        </advice>
46      </cclass>
47    </java-class-file>
48  </java-source-program>
```

Figure 11. XML representation for StateCheck AspectJ source code.

ACKNOWLEDGMENT

The authors thank CONACyT (Consejo Nacional de Ciencia y Tecnología) and DGEST (Dirección Nacional de Educación Superior Tecnológica) for the support granted for the completion of graduate study.

REFERENCES

[1] G. Kiczales, et al, "Aspect-oriented programming," in European Conference on Object-Oriented Programming (ECOOP'97), vol. 1241, June. 1997. pp. 220-242. doi: 10.1007/BFb0053381

[2] L. Vogel, "Eclipse IDE," vogella.com, 3rd edition, pp. 492.

[3] E. Y. Rosas-Sánchez, "Desarrollo de un plug-in para el ide eclipse para la generación de código de aspectj y caesarj basado en xmi y perfiles de uml," Master's thesis, Instituto Tecnológico de Orizaba, 2011.

[4] A. Salinas-Mendoza, "Caesarml: una representación basada en xml para código fuente de caesarj," Master's thesis, Instituto Tecnológico de Orizaba, 2011.

[5] R. Paige, L. Kaminskaya, J. Ostroff, and J. Lancaric, "Boncase: an extensible case tool for formal specification and reasoning," Journal of Object Technology, vol. 1, no. 3, Aug. 2002, pp. 77-96.

[6] S. Huang, D. Zook, and Y. Smaragdakis, "Domain-specific languages and program generation with meta-aspectj," ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 18, no. 2, Nov. 2008, article no. 6, pp. 1-32, doi:10.1145/1416563.1416566.

[7] C. Constantinides, A. Bader, T. Elrad, P. Netinant, and M. Fayad, "Designing an aspect-oriented framework in an object oriented environment," ACM Computing Surveys (CSUR), vol. 32, no. 1es, p. 41, Mar. 2000, pp. 1-12, doi:10.1145/351936.351978.

[8] T. Xie and J. Zhao, "A framework and tool supports for generating test inputs of aspectj programs," in Proceedings of the 5th International Conference on Aspect-oriented Software Development, Mar. 2006, pp. 190-201, doi:10.1145/1119655.1119681.

[9] J. Evermann, "A meta-level specification and profile for aspectj in uml," in Proceedings of the 10th International Workshop on Aspect-oriented Modeling, Mar. 2007, pp. 21-27, doi:10.1145/1229375.1229379.

[10] M. Hecht, E. Piveta, M. Pimenta, and R. Price, "Aspect oriented code generation," 20. Simpsio Brasileiro de Engenharia de Software (SBES'06), Florianpolis, SC, Brazil, 2006, pp. 209-223.

[11] J. Júnior, V. Camargo, and C. Chavez, "Uml-aof: a profile for modeling aspect oriented frameworks," in Proceedings of the 13th Workshop on Aspect-oriented Modeling, 2009, pp. 1-6, doi:10.1145/1509297.1509299.

[12] M. Mosconi, A. Charfi, J. Svacina, and J. Wloka, "Applying and evaluating aom for platform independent behavioral uml models," in Proceedings of the 2008 AOSD Workshop on Aspect-oriented Modeling, 2008, pp. 19-24, doi:10.1145/1404920.1404924.

 ISBN: 978-1-61208-283-7