Making Aspect Oriented System Evolution Safer

Miguel A. Pérez Toledano[1], Amparo Navasa Martínez[1],
Juan M. Murillo Rodríguez[1], Carlos Canal[2].

[1] University of Extremadura (Spain), Department of Computer Science, Quercus Software Engineering Group,
{toledano, amparonm, juanmamu}@unex.es.
[2] University of Málaga (Spain), Department of Computer Science, GISUM Group,
canal@lcc.uma.es

Abstract

The information systems of enterprises change rapidly and it is necessary for the existing software to evolve without the comprehension, modularity or quality of the built systems being affected. In this context, Aspect Oriented Programming reveals as an adequate way of working because it makes the encapsulation of methods easier and reduces development times. The inclusion of aspects (woven code) inside an existing software code could, nevertheless, cause the resulting system behaviour not to be that expected by the developer. In this paper, we propose to evolve system specifications at the same time as the software itself. In this way, we can start from these specifications in order to obtain state machines and algebraic descriptions of the components of the system. These can be used to perform verification, simulation or testing operations of the systems built. We also propose the use of extended state machines that allow us to describe the evolution of the system in a more detailed way, facilitating more complete Model Checking operations.

1. Introduction

The information systems of enterprises change rapidly and it is necessary for the existing software to evolve without the comprehension, modularity or quality of the built systems being affected. The use of Aspect Oriented Programming (AOP) facilitates this task, allowing us the encapsulation of methods that, otherwise, would be scattered in the code of the different software elements of the system. This encapsulation makes the comprehension of the code easier, reduces the time of system development and, on the whole, allows software systems to evolve rapidly. The evolution of the systems in which woven aspects exist and the difficulty to access to the source code of these applications could, nevertheless, produce a series of problems, described in [1], which could then cause the final system behaviour not to be that expected by the developer.

Currently, there is no tool that allows a complete study of every possible problem when aspects are included inside a system. Existing papers in this area are focused on some of the possible situations. Some works study how the addition of new aspects could affect a system (and also which are the properties affected by the new woven code), by means of code analysis or static Model Checking [2] and are based on the use of state machines or algebraic descriptions of the system components, while other proposals study the increased system (analysing the woven code and comparing its properties with the underlying software system) using tools to check the resulting code as Bandera [3] and Java Pathfinder [4].

Moreover, in order to generate quality software and to document the built system it is necessary to achieve a previous detailed Analysis and Design of the system to build. When systems are object-oriented, the Unified Modelling Language (UML) is usually employed as the modelling tool [5]. Several papers use UML as a tool to model aspects and to add its behaviour inside the system to build [6].In [7], state charts are used to describe aspects behaviour and also to integrate this behaviour into the state charts that describe the joint points associated. Other works, like [8], separate the specification of aspects behaviour from the system business rules. For this, the use of sequence diagrams to describe aspectual scenarios, with Interaction Patterns Specifications (IPS) [9] is proposed. These IPS will subsequently be instantiated inside those system sequence diagrams that describe the associated joint points. The objective is to obtain state charts from system components and use them later to achieve simulation and validation operations with the existing requirements.

In this paper we propose to study the integration of aspects into a software system starting from its UML specifications. This study will be focused, nevertheless, on the interactions described by sequence diagrams, assuming that the modelling will probably affect also other types of diagrams, which are beyond the scope of this paper.

As regards the contribution of this paper, we propose to build more complete state machines than the currently used state charts [7,10]. These extended machines will allow us to represent information about time counters, about fragments (as used in UML 2.0) and alsoabout system variables. All this information will permit to achieve more precise model checking operations of the system. Moreover, we study the effects of adding new aspects inside a system before they have been woven and, furthermore, to study the behaviour of the woven code. In order to facilitate this study, a technique for grouping components is proposed, so that simulation traces can be reduced due to the omission of internal interactions among grouped components, focusing on the interaction with the surrounding environment.

This paper is structured into the following points: some of the problems derived from integrating aspects inside a system are described in Section 2; Section 3 describes our proposal; while an example is presented Section 4; Section 5 contains the conclusions and future works.

2. Problem Description

The construction of software systems using Java makes the use byte code possible, protecting in this way the source code of classes. This originates that the evolution of these systems by means of AOP lacks precise information about the underlying system in which aspects must be applied. Because of that, errors can be caused when languages as AspectJ [11] are used in order to weave aspects inside Java code. These problems [1] can be summarized in:

- **Unintended aspects effects.** When pointcuts of new aspects of the underlying system are applied, they may be applied to undesired joint points of classes, and this could provoke unintended side effects.
- **Arbitrary aspect precedence.** When pointcuts of new aspects of the underlying system are applied, they may be applied to the same joint point as other (unknown) aspects already are. This may cause problems with the sequence of application of aspects
- **Unknown aspect assumptions.** When pointcuts of new aspects of the underlying system are applied, they may not find joint points matching existing requirements.
- **Partial weaving.** When the code of a system is modified, the aspects inside it may not be applied to future modifications.

These problems are caused by the difficulty of knowing the existence of previous woven aspects inside the code and the ignorance of the pointcuts defined in them.

3. Proposal

As discussed in the previous section, the evolution of Java systems by means of AOP is hindered by the frequent ignorance of the source code. In this context, the creation of an adequate specification of the system and its later adequate use to evolve the system are necessary. The following steps are proposed in order to document the system (graphically described in Figure 1):

1. System modelling by means of UML. It is necessary to separate the aspect description from the rest of the system. The idea consists of describing the behaviour of aspects in such a way that, when the system evolves, a complete documentation about its behaviour exists. This paper is focused on the study of interaction diagrams but the specification of aspects must be described inside every affected UML diagram. Interaction Patterns Specifications (IPS) will be used in order to describe aspect interactions. This tool is based on describing patterns that represent the expected interactions of the aspect to be integrated by means of sequence diagrams [8].
2. Instantiating aspect patterns (IPS) inside the sequence diagram of the system. We need to find points in the sequence diagram matching the requirements stated by aspects description where aspects can be introduced. Note that different operation exist in order to instantiate the patterns,

depending on the description of the aspects. It is possible to find further information about design and instantiation of IPS in [12].

3. The information described during system modelling is frequently not precise enough, as it is necessary to describe the system completely. In order to detect errors and gaps, the obtained specifications must be validated [13].For that purpose, algebraic descriptions of the specifications obtained are built and model checking is applied to detect deadlocks and inconsistencies. This point will allow us to complete specifications in such a way that, once an error is detected, it is necessary to return to point number one to solve it.
4. Once specifications have been validated, extended state machines can be automatically obtained for each element of the system. There exist several algorithms to achieve this task [14]. In this paper, we will use the algorithm proposed in [15]. It consists of obtaining a state machine for each scenario in which a system is involved and then machines are assembled. State labels, described in sequence diagrams, are usually employed to assemble machines in order to identify the state in which the component is inside the scenario. Nevertheless, the machines obtained [18] provide more precise descriptions than statecharts.
5. The specifications must be renewed to evolve the system. In order to do that, it is necessary to return to point number 1 and update documentation. Once new behaviours have been described, instantiating each aspect inside the system into adequate joint points is again necessary. This will allow us to detect problems in the sequence of execution of aspects and to prove if there exist adequate joint points in which aspects can be applied. Finally, aspects must be woven inside the code again in order to avoid partial weaving problems. Once the whole process is finished, new machines are obtained. Its behaviour will reflect the new described behaviour. However, it is still possible that problems arise when proving if the behaviour of the obtained woven code agrees with design.

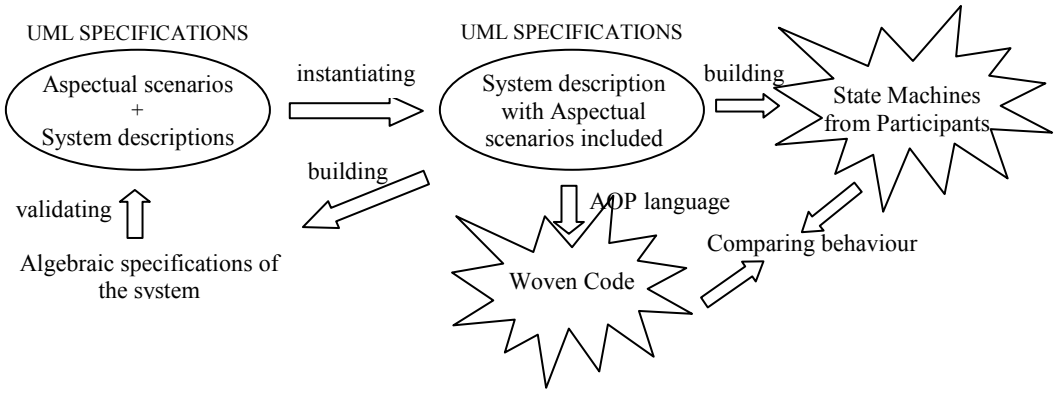

Figure 1. Building of system specifications steps

Extended state machines describe the planned behaviour of the system and the woven code describes the final behaviour obtained. Studying possible unintended effects of the aspects inside the system will consist of studying if state machines and code behaviour match. In order to achieve this task, model checking techniques can be used. These operations must study the properties of both systems and must complete the study by means of simulating the same execution traces in both of them. To execute model checking operations inside Java code obtained, Java Pathfinder can be used, whereas for state machines, UPPAAL tool is proposed [16]. UPPAAL is an integrated tool environment for modelling, simulation and verification of systems. It is appropriate for systems that can be modelled as a collection of processes with finite control structure and real-valued clocks, communicating through channels or shared variables. The UPPAL simulator enables examination of possible dynamic executions of a system during modelling stage and, thus, provides an inexpensive means of fault detection prior to the verification by the model-checker. The UPPAAL model-checker covers the exhaustive dynamic behaviour of the system; it can also check invariant and reachability properties by exploring the state-space.

To compare machines and code simulation, two alternatives exist:

1. Generating traces from UPPAAL. These traces simulate the execution of machines and can be used as inputs into the generated code.
2. Weaving one aspect inside the code that does not modify the behaviour and be limited to monitor the code execution and create traces able to be used in the UPPAAL simulator.

This second option seems to be more elegant because it permits the use of AOP concepts in order to study systems built by means of AOP. Nevertheless, execution traces obtained may be too large due to the size of the built systems. In order to reduce size, it is achievable to group components and to monitor only the interesting events to facilitate the simulation and to focus the study on the attractive points. Components grouping [17] allow us to obtain descriptions suited to sets of components, abstracting from internal interactions among them. These groups will be adapted to developer needs and will allow us to create traces exclusively containing the eventsof interests.

4. Example

Two scenarios are presented in order to illustrate our proposal (Figure 2). The first one represents the behaviour of one aspect, depicted by means of an IPS. The second one represents one scenario of the system that achieves the necessary requirements to apply the mentioned aspect. In order to prove if a certain aspect is applicable to a given scenario, we must check the associated restrictions, described by means of state labels. In the example, to instantiate the aspect, it is necessary to establish some binds between the role "|rd:" and the element "c2:class4", and between the method "|notif()" and the method "operat1()". Once the aspect is composed, the specification of the system is available. Notice that the type of instantiation of the aspect will be executed depending on its description and there exists several operations to achieve it [12]. Once final specifications have been obtained, algebraic descriptions are built with CCS, and Model Checking analysis is performed for detecting errors and deadlocks. The specifications will be modified until they are correct.

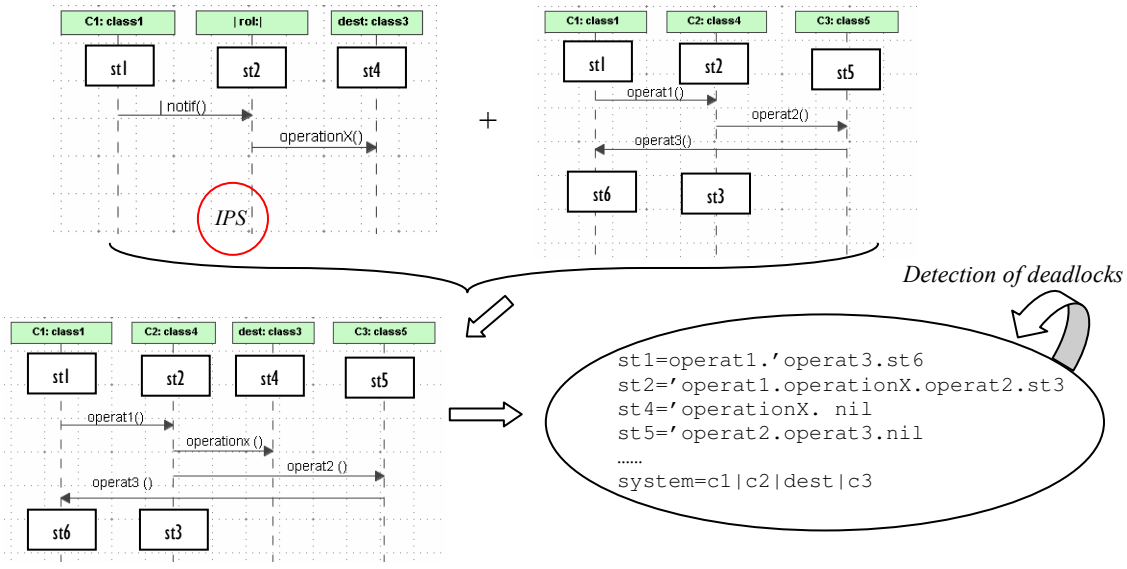

Figure 2. Instantiation of aspects and refinement of specifications described

Extended state machines are built with validated specifications. The possibility of representing time requirements and complex operations over groups of events (as critical regions) described with UML fragments, and the possibility of achieving a continuous monitoring in the evolution of the state variables of the system are advantages as regards state charts. Each vertex of a machine consists of a structure: <*par, ord, crit, st, variables*> where *par, ord* and *crit* are positive integer variables used for representing parallelism, sequences and critical regions; *st* is a string variable, used for describing the state in which the component is, and *variables* is a set of strings employed for describing the variables used in the conditions and iterations represented on the graph. Focused on the example, it is possible to build the machines associated to each element of the system. These can be used to simulate the behaviour and to obtain execution traces, useful to prove the built code.

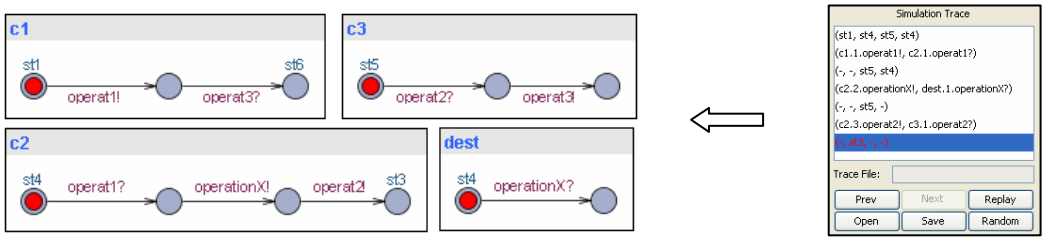

Figure 3. Obtained machines to apply traces in order to simulate its behaviour

On the other hand, when the system is implemented in Java and AspectJ is used to weave the described aspects, the resulting code can be proved by means of using the sequence of traces obtained from the previous simulation to check if the same results are returned. Another possibility of comparing designed machines with the obtained code consists of creating a new aspect to monitor the execution and build execution traces. Sometimes, these traces can be large and then, there exists the possibility of focusing the study on a series of events. In order to achieve it, the aspect can be designed in such a way that it only monitors the events of a series of classes and then, state machines whose behaviour is not interesting can be grouped, avoiding references to events that occur inside grouped components. This possibility allows us to focus the study on the classes affected by the pointcuts of the aspects of the system. For example, in Figure 4 the result of grouping components 2 and 3, from Figure 3, and aspect code used for making traces are depicted.

```
Public aspect Traceaspect {
 pointcut trace(): execution (c1.*(..))||execution (c2.operationX())
    ||execution (c2.operat1())||execution (c3.operat3());
 After(): trace(){
    Signature sig = thisJointPointStaticPart.getSignature();
    System.out.println(sig.getDeclaringType().getname()+”.”
    + sig.getName());
                }
 }
```

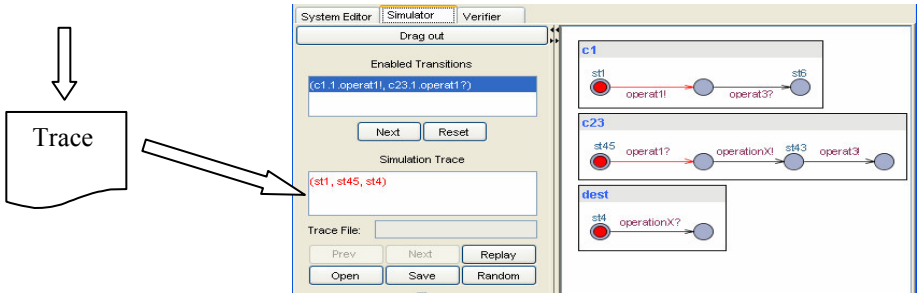

Figure 4. Aspect code to monitor events with c2 and c3 components grouped.

5. Conclusions

AOP facilitates the evolution of a software system. However, the lack of access to the source code of the applications can cause problems in existing software. This paper introduces a practical approximation that evolves the requirements of the system described by means of UML with the built code. These specifications, once validated, will allow us to obtain state machines whose behaviour could be compared with the code of the system, before and after aspects have been included. In order to make these operations easier, extended state machines are introduced, more complete than traditional statecharts. These machines are adequate to obtain all the possible information from UML sequence diagrams. With these machines, it is possible to study how a software system will be affected when an aspect is included. It is also possible to perform Model Checking with the properties of the code of the existing system and, finally, the results can be compared with those belonging to the designed machines. Besides, it will be possible to study the resulting woven code when an aspect is included, by means of Model Checking, and trace simulation between state machines and the built code. In order to facilitate these simulations, grouping state machines has been proposed (grouping algorithms is beyond the scope of this paper) to reduce the size of the traces in study and to focus the study on the involved classes.

References

[1] N. McEachen, R.T. Alexander. “Distributing classes with woven concerns: an exploration of potential fault scenarios”. Proceedings of the 4th international conference on Aspect-oriented software development.2005, Pages: 192 – 200, ISBN:1-59593-042-6.

[2] S. Katz. “A Survey of Verification and Static Analysis for Aspects”. AOSD-Europe-Technion-1. 10 July 2005.

[3] Bandera. http://bandera.projects.cis.ksu.edu

[4] Java Pathfinder. http://javapathfinder.sourceforge.net.

[5] UML homepage. http://www.uml.org

[6] O. Aldawud, T. Elrad, A. Bader. “UML Profile for Aspect-Oriented Software Development",In Proceedings of Third International Workshop on Aspect-Oriented Modeling, March 2003”.

[7] M. Mahoney and T. Errad. ” Distributing State-Charts to Handle Pervasive Crosscutting Concerns”. In Proceeding of Building Software for Pervasive Computing Workshop. OOPSLA 2005.

[8] J. Araujo, J. Whittle, D. Kim, "Modeling and Composing Scenario-Based Requirements with Aspects," re, pp. 58-67, 12th IEEE International Requirements Engineering Conference (RE'04), 2004.

[9] R. B. France, D. Kim, S. Ghosh, E. Song. “A UML-Based Pattern Specification Technique”. IEEE Transaction on Software Engineering. March 2004 (Vol. 30, No. 3) pp. 193-206

[10] J. Whittle, J. Schumann. “Generating statechart designs from scenarios”. International Conference on Software Engineering. Proceedings of the 22nd international conference on Software engineering. Pages: 314 – 323, 2000, ISBN:1-58113-206-9.

[11] AspectJ. http://www.eclipse.org/aspectj

[12] J. Whitlle, J. Araujo. “Scenario Modeling with Aspects”. IEE Proceedings - Software -- August 2004 -- Volume 151, Issue 4, p. 157-171.

[13] S. Uchitel. “Incremental elaboration of scenario-based specifications and behavior models using implied scenarios”. ACM Transactions on Software Engineering and Methodology (TOSEM). Volume 13, Issue 1 (January 2004), Pages: 37 – 85, Year of Publication: 2004, ISSN:1049-331X.

[14] 4th International Workshop on Scenarios and State Machines: Models, Algorithms and Tools. in Saint-Louis, Missouri, on 21 May 2005.

[15] J. Whittle, J. Saboo, R. Kwan, "From Scenarios to Code: An Air Traffic Control Case Study," icse, p. 490, 25th International Conference on Software Engineering (ICSE'03), 2003.

[16] UPPAAL. http://uppaal.com

[17] L. Blair, G. Blair. “Composition in Multi-paradigm Specification Techniques”. Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS). Pag. 401 – 417, 1999.

[18] M.A. Pérez, A.Navasa, J.M. Murillo, C.Canal. “Definición de máquinas de estados extendidas usadas en descripción de protocolos de interacción”. Technical Report TR-23/2006. University of Extremadura (Spain). An English short version can be found in: http://es.geocities.com/merceyahdes/appendix.pdf