DETECTING ASPECTUAL BEHAVIOR IN UML INTERACTION DIAGRAMS

Amir Abdollahi Foumani
IBM Rational Software Group, Montreal, Quebec, Canada

Constantinos Constantinides
Department of Computer Science and Software Engineering, Concordia University, Montreal, Quebec, Canada

Keywords: Object-oriented design, Unified Modeling Language, production system, crosscutting, aspectual behavior, aspect mining.

Abstract: In this paper we discuss an approach to detect potential aspectual behavior in UML interaction diagrams. We use a case study to demonstrate how our approach can be realized: We adopt a production system to represent the static and dynamic behavior of a design model. Derivation sentences generated from the production representation of the dynamic model allow us to apply certain strategies in order to detect aspectual behavior which we categorize into "horizontal" and "vertical." Our approach can aid developers by providing indications over their designs where restructuring may be desired.

1 INTRODUCTION

During object-oriented development, certain concerns cannot be directly mapped from the problem domain to the solution space. As a result, their implementation tends to be scattered throughout the class hierarchy of the system. Even though this "crosscutting phenomenon" was initially observed over implementation artifacts, it does not originate in implementation but it propagates over from previous development stages. In this work we are investigating the crosscutting phenomenon during design. Artifacts during this stage are illustrated by the Unified Modeling Language (UML) which provides a notation for representing a model of a system, capturing its structure and behavior. We can represent the semantics and metadata behind the UML model in a form that indicates the ways in which the knowledge can be used. More precisely each item of knowledge can be represented by a rule which combines data and semantics and specifies when to use it and how to use it. A rule of this kind is called a production and a system which represents knowledge as a set of productions is called a production system. Our approach is based on obtaining a production system from a UML model and applying certain strategies in order to capture crosscutting (or "aspectual") behavior. The expected benefit of this approach is to provide developers with an early indication of design spots where model transformation (restructuring) might be required or desirable.

Recently there has been an increasing number of approaches into mining crosscutting concerns from implementation artifacts and from requirements. Very little work has been done on mining crosscutting concerns from design artifacts and with this work we aim in contributing to bridging this gap.

The rest of this paper is organized as follows: In Section 2 we discuss the necessary theoretical background to this research. In Section 3 we discuss the problem and motivation behind this research. In Section 4 we present our proposal and in Section 5 we use a case study to demonstrate and discuss our proposal. In Section 6 we discuss related work and comparisons to our proposal. We conclude our work in Section 7 with a summary, discussion and pointers to future research directions.

2 THEORETICAL BACKGROUND

This section provides the necessary theoretical background to this research project.

Abdollahi Foumani A. and Constantinides C. (2007).
DETECTING ASPECTUAL BEHAVIOR IN UML INTERACTION DIAGRAMS.
In *Proceedings of the Second International Conference on Software and Data Technologies - SE*, pages 378-386
DOI: 10.5220/0001341203780386

2.1 Aspect-oriented Programming

The principle of separation of concerns (Parnas, 1972) refers to the realization of system concepts into separate software units and it is a fundamental principle to software development. The associated benefits include better analysis and understanding of systems, high readability of modular code, high level of reuse, easy adaptability and good maintainability. Despite the success of object-orientation in the effort to achieve separation of concerns, certain properties cannot be directly mapped in a one-to-one fashion from the problem domain to the solution space, and thus cannot be localized in single modular units. Their implementation ends up cutting across the inheritance hierarchy of the system. Crosscutting concerns (or “aspects”) include persistence, authentication, synchronization and logging. The “crosscutting phenomenon” creates two implications: 1) the scattering of concerns over a number of modular units and 2) the tangling of code in modular units. As a result, developers are faced with a number of problems including a low level of cohesion of modular units, strong coupling between modular units and difficult comprehensibility, resulting in programs that are more error prone.

Aspect-Oriented Programming (AOP) (Kiczales et al., 1997; Elrad et al., 2001) explicitly addresses those concerns which “can not be cleanly encapsulated in a generalized procedure (i.e. object, method, procedure, API)” (Kiczales et al., 1997) by introducing the notion of an aspect definition, which is a modular unit of decomposition. There is currently a growing number of approaches and technologies to support AOP. One notable technology is AspectJ (Kiczales et al., 2001), a general-purpose aspect-oriented extension to the Java language, which has influenced the design dimensions of several other general-purpose aspect-oriented languages, and has provided the community with a common vocabulary based on its own linguistic constructs. In the AspectJ model, an aspect definition is a new unit of modularity providing behavior to be inserted over functional components. This behavior is defined in method-like blocks called *advice* blocks. However, unlike a method, an advice block is never explicitly called. Instead, it is activated by an associated construct called a *pointcut* expression. A pointcut expression is a predicate over well-defined points in the execution of the program which are referred to as *join points*. When the program execution reaches a join point captured by a pointcut expression, the associated advice block is executed. Even though the specification and level of granularity of the join point model differ from one language to another, common join points in current aspect-oriented language specifications include calls to methods and constructors as well as executions of methods and constructors. Most aspect-oriented languages provide a level of granularity which specifies exactly when an advice block should be executed, such as executing before, after, or instead of the code defined at the associated pointcut. Furthermore, much like a class, an aspect definition may contain state and behavior. It is also important to note that AOP is neither limited to object-oriented programming (OOP) nor to the imperative programming paradigm. However, we will restrict this discussion to the context of object-orientation.

2.2 Production Rules and Derivation Sentences Over Uml Semantics

A formal description of semantics is essentially a means to represent knowledge. We can represent knowledge in a form that indicates the ways in which the knowledge can be used. More precisely, each item of knowledge can be represented by a rule which specifies when to use it and how to use it. Such a rule takes the form “When a condition of type C occurs, execute action A.” A rule of this kind is called a *production* and a system which represents knowledge as a set of productions is called a *production system*. A sequence of rule applications is called a *derivation* and the result of this process is called a *derivation sentence* (Aho et al., 1986).

With UML we are able to model and visualize a real world system based on object definitions and object relationships. The semantics and metadata behind the model can be represented as a set of abstract rules, which can be production rules. By definition, production rules must be finite, implying that in order to represent the semantics of UML artifacts we need a limited number of production rules. As a consequence, our knowledge, as incorporated in the production rules, must also be finite. It is the process of using this knowledge that will be “productive.” In effect, production rules can define an infinite set of scenarios. *Derivation sentences* can be deployed over these production rules for representing a scenario.

In (Foumani and Constantinides, 2005a) and (Foumani and Constantinides, 2005b) we presented a set of production rules to represent the semantics behind UML artifacts, where the static model is represented by a class diagram and the dynamic model is represented by a set of interaction diagrams. More specifically, the semantics of object-oriented artifacts, `G`, can be defined in terms of a set of five elements, each of which is finite. Let `G = {C, A, M, P, R}`, such that

1. `C` is a set of classes.
2. `A` is a set of attributes.
3. `M` is a set of methods.
4. `P` is a set of transformation rules that defines object-oriented design semantics in terms of: a) definition of classes, b) hierarchy of classes, c) relationships between classes and d) system scenarios in terms of message passing between class instances.
5. `R` is set of relationships and concepts defined by the object-oriented methodology. We define this set as `R = {[declares], [has], [collection], {(guard)[calls]}*, [extends]}`, where `(guard)[calls]*` is used to model dynamic behavior in UML artifacts (message passing between objects and control flow in interaction diagrams). The rest of the elements in the set are used for defining the semantics of the static model.

3 PROBLEM AND MOTIVATION

One might argue that the implementation stage should be the appropriate place to look for aspectual behavior, because this is where the phenomenon is obvious. Indeed our knowledge about crosscutting (despite the fact that the term was coined in the mid 90's) dates back to the 80's from observations, over code artifacts, of conflicts between OOP and concurrency (Briot and Yonezawa, 1987) which in the early 90's became known as "inheritance anomalies" (Matsuoka and Yonezawa, 1993). In the mid 90's composability issues in OOP over several domains started to surface[1]. A question one might put forward is whether or not one should be concerned with crosscutting at the design level, or whether crosscutting can even manifest itself at the design level. A response to this can be argued as follows: Mainly due to the fact that design artifacts during maintenance are many times not synchronized with code and thus become inconsistent, the code is indeed the only reliable source of the current state of the system. Thus, focusing on code is unavoidably the only option. As a result, maintainers would normally not be highly concerned with documentation other than code. However, during development, developers aim to produce a clean (tangled-free) implementation and achieve the maximum benefits of advanced separation of concerns. To meet this objective, the design artifacts themselves (such as UML interaction diagrams and the class diagram) must in turn explicitly address crosscutting concerns. We therefore need to provide the means to identify and model crosscutting concerns from the early stages of the software life cycle. As a result, the explicit capture of crosscutting concerns in code should be the natural consequence of good and clean modularity and not the result of a corrective measure (refactoring activity) due to a tangled implementation. Is this a real problem? One can argue that, as "early aspects" approaches (see related work in Section 6) aim in detecting and identifying crosscutting concerns from the requirements stage, the design would be aspect-free. Our response to this is that early-aspects can perhaps guarantee an aspect-free requirements model, but would not guarantee an aspect-free design, since some aspects may appear during design. We need to focus on the crosscutting which a) either has remained after having deployed early-aspects approaches, or b) originates in design. The motivation behind this research is to be able to detect aspectual behavior in order to aid developers in performing any necessary model restructuring before mapping the design into code.

[1]cf. Proceedings of the ECOOP 2006 Workshop on Composability Issues in Object-Orientation (entire volume)

4 PROPOSAL: DETECTING ASPECTUAL BEHAVIOR

Consider the sequence of object-oriented analysis and design activities as described in (Larman, 2004): A scenario describes the interaction between an actor and a system and it is usually described in a narrative form as part of a use-case which is a collection of related scenarios. The scenario refers to real-world concepts which can be captured by the domain model. The scenario can also be translated into a system sequence diagram, illustrating the request events (generated by the actor) and the corresponding system operations which define system responsibilities. Each system operation can then be translated into a UML interaction diagram. The interaction diagram will capture the collaboration of software entities (extracted or influenced from the domain model) in order to implement the responsibility defined by the corresponding system operation. In creating UML interaction diagrams, developers may apply general responsibility assignment software patterns (GRASP) and design patterns in order to support software quality through low coupling and high cohesion and to fully utilize the benefits associated with object-orientation which include high level of reusability of elements and easy adaptability of the system.

Our analysis is focused on the crosscutting which manifests in UML interaction diagrams. As the interaction diagrams can be represented by a production system, we plan to analyze the production system by applying strategies in order to detect aspectual behavior.

5 CASE STUDY: A PROJECT MANAGEMENT SYSTEM

We provide a demonstration of our proposal through a case study of a project management system. In this system, resources can be given one or more assignments to be undertaken on a given task. Each assignment has a start date, finish date and duration and can be "rolled-up" to the corresponding task, deliverable and project level. Each project is a composition of deliverables and tasks and each deliverable can contain tasks (or other deliverables). Furthermore, resources can also be assigned to the deliverables and projects with different roles and responsibilities. The system provides a number of services to project managers such as 1) assign-unassign resources and 3) level projects.

5.1 Static Model

A (partial) class diagram of the system is shown in Figure 1, where the Work Breakdown Structure (WBS) is organized in terms of a hierarchy between `Project`, `Deliverable` and `Task`. Tasks as leaf nodes cannot have children. This condition can be enforced by defining an invariant for class `Task`. We can provide a production system to represent the static structure of the system as follows:

```
<class>::= {Assignment, Resource,
            Deliverable, Task, Project}
<Assignment>::=[has]<assign()>
<Assignment>::=[has]<unassign()>
<Assignment>::=[has]<level()>
<Assignment>::=[declares]<Resource>
<Assignment>::=[declares]<Deliverable>
<Resource>::=[declare]<name>
<Resource>::=[collection]<Assignment>
<Deliverable>::=[declares]<start_date>
<Deliverable>::=[declares]<finish_date>
<Deliverable>::=[declares]<duration>
<Deliverable>::=[declares]<name>
<Deliverable>::=[declares]<Project(Parent)>
<Deliverable>::=[has]<rollup()>
<Deliverable>::=[has]<add()>
<Deliverable>::=[has]<remove()>
<Deliverable>::=[has]<getChild()>
<Deliverable>::=[collection]<Assignment>
<Task>::=[extends]<Deliverable>
```

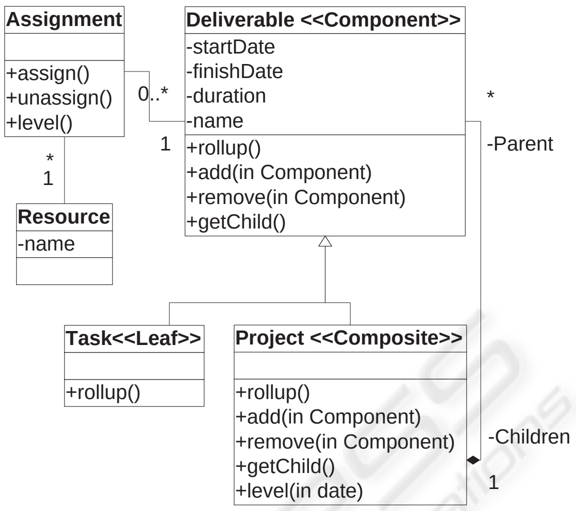


Figure 1: UML class diagram illustrating the WBS hierarchy.

```
<Project>::=[extends]<Deliverable>
<Project>::=
     [collection]<Deliverable(Children)>
<Project>::=[has]<level()>
```

5.2 Dynamic Model

The dynamic model is depicted by two scenarios: 1) assign-unassign resources and 2) level projects.

5.2.1 Resource Assignment

Resource assignment entails obtaining an available resource and assigning the resource to a node in the WBS hierarchy, or unassigning a resource from a node of the WBS hierarchy. The effect of these operations must be rolled-up to the parent deliverable(s) and eventually to the project level, followed by an update of `startDate`, `finishDate` and `duration` attributes appropriately. We define a `rollup()` operation which is responsible to consistently update the WBS hierarchy whenever an operation modifies one of the `startDate`, `finishDate` and `duration` attributes. We may have any level of nested deliverables in a WBS hierarchy so that in order to model the rollup process in a UML interaction diagram a loop statement can be used. Figures 2 and 3 illustrate the process of assigning and unassigning a resource to a task. In these interaction diagrams, an `Assignment` object would invoke a new session before sending any other message. Upon completion of the process, `Assignment` will decide to commit or

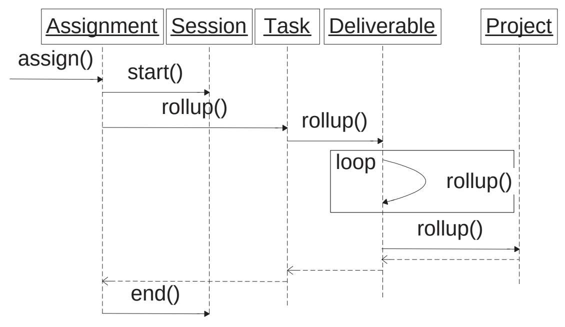


Figure 2: Assigning a resource.

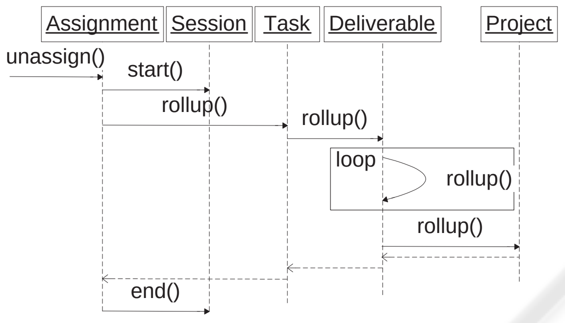


Figure 3: Unassigning a resource.

rollback the assignment operation based on the situation and status of the objects. Object `Session` is responsible for the manipulation of the transaction session. Class `Session` has methods `start()` and `end()` in order to indicate a session start and end respectively.

5.2.2 Leveling

Figure 4 illustrates the project leveling process. By leveling a project, a manager is able to move the start date of a project and to adjust the WBS hierarchy appropriately.

5.2.3 Building Derivation Sentences

By deploying production rules we can build the following derivation sentences for the operation of resource assignment and leveling, as follows:

```
<Assignment.assign()>::=<Session.start()>
<Assignment.assign()>
       ::=[calls]<Task.rollup()>
       ::={(parent is Deliverable)
          [calls]<Deliverable.rollup()>}*
```

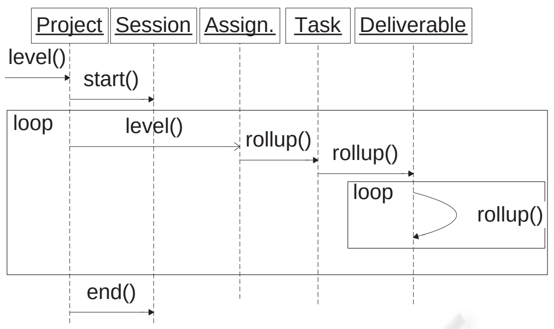


Figure 4: Project leveling.

```
       ::=[calls]<Project.rollup()>
<Assignment.assign()>::=<Session.end()>

<Assignment.unassign()>::=<Session.start()>
<Assignment.unassign()>
       ::=[calls]<Task.rollup()>
       ::={(parent is Deliverable)
          [calls]<Deliverable.rollup()>}*
       ::=[calls]<Project.rollup()>
<Assignment.unassign()>::=<Session.end()>

<Project.level()>::=<Session.start()>
<Project.level()>
       ::={[calls]<Assignment.level()>
       ::=[calls]<Task.rollup()>
       ::={(parent is Deliverable)
          [calls]<Deliverable.rollup()>}*
       ::=[calls]<Project.rollup()>}*
<Project.level()>::=<Session.end()>
```

5.3 Detecting Horizontal and Vertical Aspects

We provide the following two definitions:

Definition 1 *Behavior, defined by sequences of message passing between objects in a collection of UML interaction diagrams over multiple levels of interaction, can be referred to as "horizontal crosscutting." Subsequently we can adopt the term "horizontal aspect" to define this behavior.*

Definition 2 *Behavior, defined by sequences of message passing between objects in a collection of UML interaction diagrams over a single level of interaction, can be referred to as "vertical crosscutting." Subsequently we can adopt the term "vertical aspect" to define this behavior.*

Given a collection of UML interaction diagrams (corresponding to one or possibly more use-case scenarios) of Figures 2, 3 and 4 and their corresponding derivation sentences representing the execution paths already defined in 5.2.3, our objective is to detect common subsequences. To do that, we proceed in three steps discussed in the subsequent subsections and illustrated in Figure 5.

5.3.1 Step 1: Bit Representation of Method Invocations

We reserve a bit for each method of each class in the system. In this case study, we can allocate 4-bits for each class implying that each class is not expected to have more than four methods. A bit representation of a message sequence is built by joining these 4-bit sequences: We allocate the first 4-bits to class `Assignment`, the next four bits to class `Task` and similarly to classes `Deliverable`, `Project` and `Session`. The class-method pair bit representations are shown as follows:

```
class-method pair              Reserved bit
                               in
                               bit-pattern

Assignment.assign()             bit(i)
Assignment.unassign()           bit(i) + 1
Assignment.level()              bit(i) + 2
reserved bit                    bit(i) + 3

Task.rollup()                   bit(i) + 4
reserved bit                    bit(i) + 5
reserved bit                    bit(i) + 6
reserved bit                    bit(i) + 7

Deliverable.rollup()            bit(i) + 8
reserved bit                    bit(i) + 9
reserved bit                    bit(i) + 10
reserved bit                    bit(i) + 11

Project.rollup()                bit(i) + 12
Project.level()                 bit(i) + 13
reserved bit                    bit(i) + 14
reserved bit                    bit(i) + 15

Session.start()                 bit(i) + 16
Session.end()                   bit(i) + 17
reserved bit                    bit(i) + 18
reserved bit                    bit(i) + 19
```

To model method invocations in a message sequence we set the corresponding bit of an invoked method to 1. We can therefore represent the UML interaction diagrams of Figures 2, 3 and 4 as follows:

```
Sa (Assign resource):
                  1000-1000-1000-1000-1100

Sb (Unassign resource):
                  1000-1000-1000-1000-1100

Sc (Project leveling):
                  0010-1000-1000-1100-1100
```

5.3.2 Step 2: Detecting Common Subsequences

In addition, we reserve a bit for each message sequence. To detect a pattern of aspectual behavior, we perform a bitwise OR over the patterns for message sequences to find out which patterns are introduced by which message sequences and we perform a bitwise AND over the patterns for method invocations. The result of the bitwise AND operation identifies the methods that can be grouped together to form an aspect.

In the example, we perform a bitwise AND over `Sa (Assign resource)`, `Sb (Unassign resource)` and `Sc (Project leveling)`. The bitwise AND operation will yield `0000-1000-1000-1000-1100` which represents the method invocations `Task.rollup()`, `Deliverable.rollup()`, `Project.rollup()`, `Session.start()` and `Session.end()`. The order of the method invocation identified by bit patterns is defined by production rules defined for the message sequences. The result of the bitwise AND operation illustrates that {`Task.rollup()`, `Deliverable.rollup()`, `Project.rollup()`, `Session.start()`, `Session.end()`} can be an aspect, whereas the production rules illustrate that we will have two different aspects: {`Task.rollup()`, `Deliverable.rollup()`, `Project.rollup()`} defines a horizontal aspect, while {`Session.start()`, `Session.end()`} defines a vertical aspect based on our definitions.

5.3.3 Step 3: Defining Aspectual Behavior

We can follow the design dimensions and the vocabulary introduced by the AspectJ programming language to capture this common behavior: For the horizontal aspect, the pointcut is a predicate over all three external messages and it can be defined as the disjunction of `Assignment.assign()`, `Assignment.unassign()` and `Assignment.level()`. The advice block would be the sequence `<Task.rollup(),`

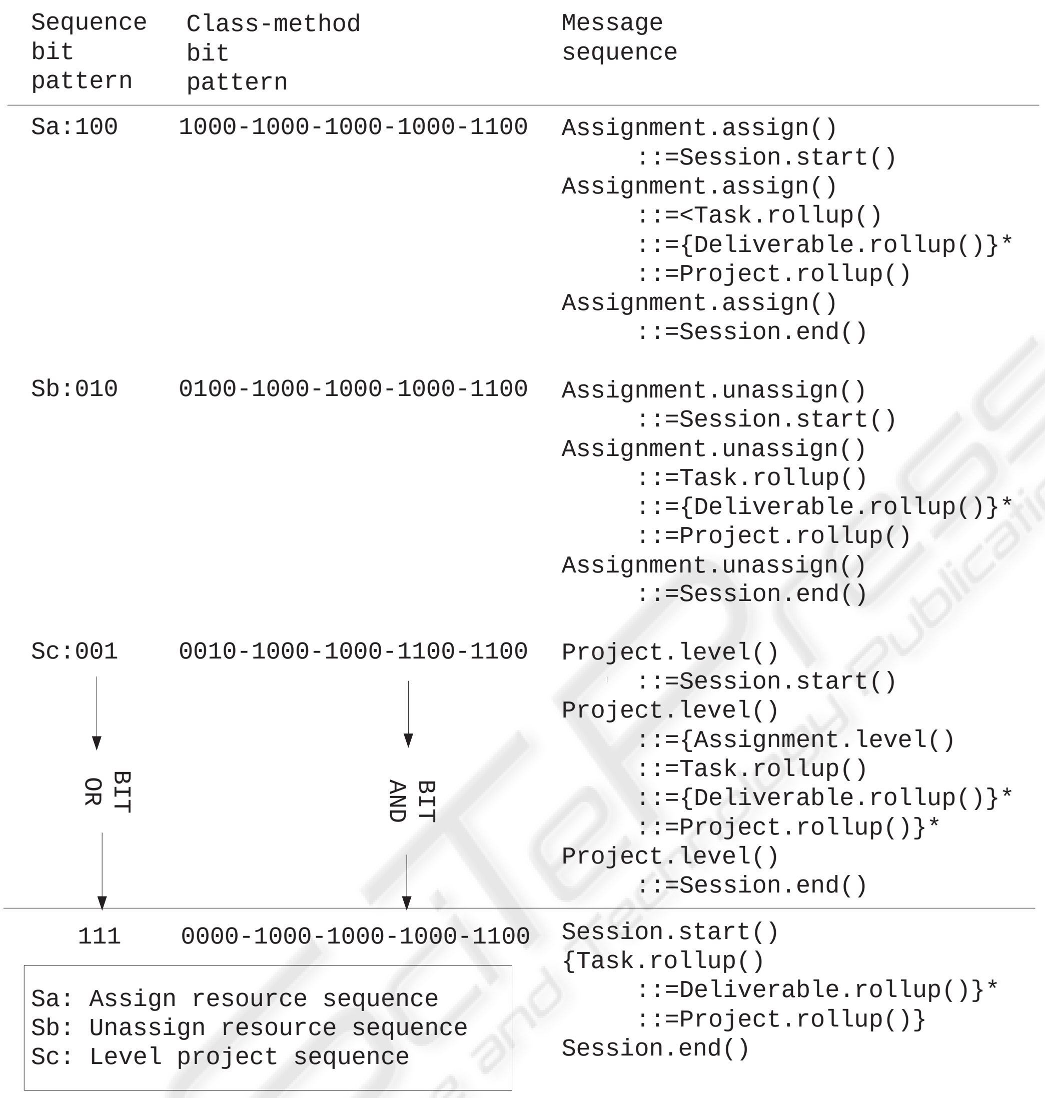
Deliverable.rollup()*, Project.rollup()>. For the vertical aspect, the pointcut can be defined as the disjunction of Assignment.assign(), Assignment.unassign() and Project.level(). We introduce two different advice blocks for starting a session and ending a session: {Session.start()} and {Session.end()}.

6 RELATED WORK

A number of authors have discussed aspect mining techniques. Almost all of them focus on mining of aspects from source code or from execution traces. In (Breu and Krinke, 2004) the authors describe an automatic dynamic aspect mining approach which deploys program traces generated in different program executions. These traces are investigated for recurring execution patterns based on different constraints, such as the requirement that the patterns have to exist in a different calling context in the program trace. In (Krinke and Breu, 2004) the authors describe an automatic static aspect mining approach, where the control flow graphs of a program are investigated for recurring executions based on different constraints, such as the requirement that the patterns have to exist in a different calling context. In (Robillard and Murphy, 2002) the authors introduce a concern graph representation that abstracts the implementation details of a concern and it makes explicit the relationships between different elements of the concern for the purpose of documenting and analyzing concerns. To investigate the practical tradeoffs related to this approach, they present a tool (Feature Exploration and Analysis Tool – FEAT) that allows a developer to manipulate a concern representation extracted from a Java system and to analyze the relationships of that concern to the code base. In (Robillard and Murphy, 2001) the authors describe concerns based on class members. This description involves three levels of concern elements: use of classes, use of class members and class member behavior elements (use of fields and classes within method bodies). Use of classes is expressed by class-use production rules. These rules specify whether an entire class or certain features of it implement a given concern. In the latter case, the rules indicate the class features which implement this concern. In (Bruntink, 2004), the author defines certain clone class metrics for known maintainability problems such as code duplication and code scattering. Subsequently, these metrics are combined into a grading scheme designed to identify interesting clone classes for the purpose of improving maintainability using aspects. In (Baxter et al., 1998), the authors initially deploy parsing to obtain a syntactical representation of the source code, typically an abstract syntax tree (AST). They then deploy algorithms to search for similar subtrees in the AST which would indicate duplicated code (“clones”). In (Parsamanesh et al., 2006) we discussed a method of running use-case scenarios and capturing the corresponding execution traces in a relational database followed by the detection of patterns of messages, indicating candidate crosscutting concerns. In order to identify an optimal solution while choosing an aspect among a collection of candidate aspects, we deployed dynamic programming algorithms.

Aspect mining at stages earlier to implementation have mainly focused on requirements, giving rise to the notion of “early aspects”, see for example (Moreira et al., 2002) and a large collection of resources at (Early Aspects Portal, 2007). In (Sutton, 2002) the author introduces a general-purpose, multidimensional, concern-space modeling schema that can be used to model early-stage concerns to identify crosscuttings.

Aspect mining at the design stage has hardly been explicitly addressed in the literature. We refer the reader to the discussion in (van den Berg et al., 2006) in which the authors adopt a cross matrix between two consecutive phases such as design and implementation in order to identify the scattering of source elements along a collection of target elements, as well as the tangling of source elements into single source elements. In earlier work discussed in (Foumani and Constantinides, 2005a) we proposed two strategies for the detection of crosscutting concerns. Our first strategy used graph theory to detect entities with independent roles in scenarios, marking these entities as candidate aspects. Our second strategy has two goals: First, it works on graphs of communicating entities and it deploys production rules and derivation sentences as a tool to locate cycles among communicating entities, thus complementing the first strategy. Second, it deploys production rules and derivation sentences in order to detect duplication and scattering of behavior. As a consequence of our second strategy, we can detect aspects across various dimensions of design. In (Foumani and Constantinides, 2005b) we focused on our first strategy for aspect mining and we illustrated that alternative designs (remodularization) over the same set of requirements cannot eliminate crosscutting. We argued that the solution to the crosscutting phenomenon is the reengineering of the system.



```
Sequence   Class-method               Message
bit        bit                        sequence
pattern    pattern
Sa:100    1000-1000-1000-1000-1100    Assignment.assign()
                                           ::=Session.start()
                                      Assignment.assign()
                                           ::=<Task.rollup()
                                           ::={Deliverable.rollup()}*
                                           ::=Project.rollup()
                                      Assignment.assign()
                                           ::=Session.end()

Sb:010    0100-1000-1000-1000-1100    Assignment.unassign()
                                           ::=Session.start()
                                      Assignment.unassign()
                                           ::=Task.rollup()
                                           ::={Deliverable.rollup()}*
                                           ::=Project.rollup()
                                      Assignment.unassign()
                                           ::=Session.end()

Sc:001    0010-1000-1000-1100-1100    Project.level()
                                           ::=Session.start()
                                      Project.level()
                                           ::={Assignment.level()
                                           ::=Task.rollup()
                                           ::={Deliverable.rollup()}*
                                           ::=Project.rollup()}*
                                      Project.level()
                                           ::=Session.end()
   111    0000-1000-1000-1000-1100    Session.start()
                                      {Task.rollup()
                                           ::=Deliverable.rollup()}*
                                           ::=Project.rollup()}
                                      Session.end()

Sa: Assign resource sequence
Sb: Unassign resource sequence
Sc: Level project sequence
```

Figure 5: Detecting common method invocations in message sequences.

7 CONCLUSION AND RECOMMENDATIONS

Software design is an integral activity of the development lifecycle. An AOP implementation should ideally be the natural mapping from a clean (well-modularized) design, rather than a corrective measure (refactoring activity) due to a tangled implementation. Scattering and tangling of concerns can cause problems with comprehensibility, traceability and reusability of concerns and the overall adaptability of the system.

In this paper we presented an approach to detect aspectual behavior in UML interaction diagrams. We built on earlier work discussed in (Foumani and Constantinides, 2005a) by focusing on an aspect mining strategy which adopts a production system to represent the static and dynamic behavior of a design model, demonstrating our approach with a case study. Our approach can aid developers by providing indications over their designs where restructuring may be desired by explicitly capturing crosscutting concerns (aspects).

In the future we plan to shift our focus on aspect mining over implementation artifacts. To aid the maintenance of legacy object-oriented systems, we plan to deploy the same strategies over implementation artifacts by parsing code and generating derivation sentences.

REFERENCES

Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers: Principles, techniques, and tools*. Addison Wesley.

Baxter, I. D., Yahin, A., Moura, L., Sant'Anna, M., and Bier, L. (1998). Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance (ICSM)*.

Breu, S. and Krinke, J. (2004). Aspect mining using event traces. In *Proceedings of the 19th International Conference on Automated Software Engineering (ASE)*.

Briot, J.-P. and Yonezawa, A. (1987). Inheritance and synchronization in concurrent OOP. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*.

Bruntink, M. (2004). Aspect mining using clone class metrics. In *Proceedings of the WCRE Workshop on Aspect Reverse Engineering*.

Early Aspects Portal (2007). Early aspects: Aspect-oriented requirements engineering and architecture design. URL: http://early-aspects.net.

Elrad, T., Filman, R. E., and Bader, A. (2001). Aspect-oriented programming. *Communications of the ACM*, 44(10):29 – 32.

Foumani, A. A. and Constantinides, C. (2005a). Aspect-oriented reverse engineering. In *Proceedings of the 9th World Multiconference on Systemics, Cybernetics and Informatics (WMSCI)*.

Foumani, A. A. and Constantinides, C. (2005b). Reengineering object-oriented designs by analyzing dependency graphs and production rules. In *Proceedings of the 9th IASTED International Conference on Software Engineering and Applications (SEA)*.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*.

Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*.

Krinke, J. and Breu, S. (2004). Control-flow-graph-based aspect mining. In *Proceedings of the 1st WCRE Workshop on Aspect Reverse Engineering*.

Larman, C. (2004). *Applying UML and patterns. An introduction to object-oriented analysis and design and iterative development*. Addison-Wesley, 3rd edition.

Matsuoka, S. and Yonezawa, A. (1993). Analysis of inheritance anomaly in object-oriented concurrent programming languages. In Agha, G., Wegner, P., and Yonezawa, A., editors, *Research directions in concurrent object-oriented programming*, pages 107–150. MIT Press.

Moreira, A., Araújo, J., and Brito, I. S. (2002). Crosscutting quality attributes for requirements engineering. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE)*.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053 – 1058.

Parsamanesh, P., Foumani, A., and Constantinides, C. (2006). Mining anomalies in object-oriented implementations through execution traces. In *Proceedings of the 1st International Conference on Software and Data Technologies (ICSOFT)*.

Robillard, M. P. and Murphy, G. C. (2001). Analyzing concerns using class member dependencies. In *Proceedings of the ICSE Workshop on Advanced Separation of Concerns in Software Engineering*.

Robillard, M. P. and Murphy, G. C. (2002). Concern graphs: Finding and describing concerns using structural program dependencies. In *Proceedings of the 24th International Conference on Software Engineering (ICSE)*.

Sutton, S. M. (2002). Early-stage concern modeling. In *Proceedings of the AOSD Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design*.

van den Berg, K., Conejero, J. M., and Hernández, J. (2006). Identification of crosscutting in software design. In *Proceedings of the AOSD International Workshop on Aspect-Oriented Modeling*.