Aspect Mining from a Modeling Perspective

Jing Zhang, Jeff Gray and Yuehua Lin

Department of Computer and Information Sciences
University of Alabama at Birmingham
Birmingham, Alabama, USA
{zhangj, gray, liny} @ cis.uab.edu

Aspect mining aims at identifying, analyzing, and refactoring crosscutting concerns throughout an existing legacy system for the purpose of improving software modularization. Current research on aspect mining prevails at the implementation level as applied to source code [12], [13], [15]. However, an aspect-oriented approach can be beneficial at different stages of the software lifecycle and at various levels of abstraction [5], [1], [8]. For instance, aspect-oriented analysis and design [5] is a new design philosophy for uniting aspects with requirements and design models.

This article presents our investigation into raising the benefits of aspect mining to a higher level of abstraction through application of aspect mining algorithms to domain-specific models. A key contribution of the approach is a capability to identify crosscutting concerns early in development, which assists in modularizing a design through aspects before proceeding to the implementation level. Furthermore, our experience has led us to believe that aspects are easier to identify and refactor at the modeling level because much of the accidental complexities of implementation concerns have been removed in the corresponding modeling abstractions.

Key challenges of aspect mining

The challenges of aspect mining are focused along three separate phases:

Aspect Identification: This phase is concerned with an analysis task that leads to identification of a suggested set of candidate aspects. This phase may require user interaction to provide initial seed information, or to assist in sifting through false positive noise (i.e., suggested aspects that are not really representative of a crosscutting concern).

Aspect Extraction: After a set of candidate aspects has been identified, the code representing the crosscutting concern must be extracted (i.e., all of the locations in the base program where the aspect appears must be removed).

Aspect Rephrasing: After extracting the aspects from the base program, an equivalent aspect must be codified in an aspect language in order to preserve the initial functionality of the base program. The result is improved modularization as captured in the newly created aspect.

With respect to these three phases of aspect mining, there appear to be no reports in the research literature on tools that perform all three of the above challenges successfully. Most aspect mining research tools are focused on one phase of aspect mining, with the majority of work (as summarized in the next section) focused on the aspect identification phase. With respect to extraction and rephrasing, a technique that uses program slicing to perform these two phases has been presented by Ettinger and Verbaere [7].

State-of-the-art in aspect mining research

Several existing aspect mining tools have been described in the literature, including a comparison of three approaches [4]. The current state-of-the-art in aspect mining is represented by the collection of tools described below.

Aspect Browser [10] enables users to enter regular expressions as patterns to identify aspects. An early contribution of Aspect Browser was an aspect visualizer that graphically conveyed a visual overview of the crosscutting effect of a specific aspect.

In Prism [15], users define a fingerprint that captures a certain property of a crosscutting concern in code. The Prism advisor autonomously computes the crosscutting property of the mining target and returns all of the matches, which are called footprints.

FEAT [12] introduces the concept of a concern graph that localizes an abstracted representation of program elements contributing to the implementation of the concern. FEAT enables users to perform maintenance tasks that involve non-localized changes. Users initiate the search process by providing a seed, which is expressed through a text file using a
declarative language to describe a concern. FEAT generates the concern graph automatically according to the declared concern. Users can visit the source file corresponding to each class in the concern graph.

Ophir [13] is a fully automatic mining and refactoring tool based on the combination of a programming dependence graph (PDG) and abstract syntax tree (AST). Ophir’s clone detection algorithm starts only at specific points of each method in order to speed up the processing time. However, this approach may overlook some potential aspects.

Aspect Browser, Prism, and FEAT all require user interaction. Users must understand the application domain and provide the pattern seed from their knowledge of the code. All of these aspect mining tools are focused at the source code level. To our knowledge, no other aspect mining research has been presented that focuses on the modeling concerns.

Aspects in domain-specific modeling

Domain-specific modeling is being adopted with more frequency in the development of computer based systems, especially in the domain of embedded control software [11] (e.g., avionics and automotive control systems). Meta-configurable domain-specific modeling environments provide support for customization of modeling tools that enable domain experts to construct models in notations that are familiar to them. Such tools also offer the ability to generate, or synthesize various artifacts from models. The ability to describe properties of a system at a higher level of abstraction, and in a technology-independent notation, can protect key intellectual assets from technology obsolescence.

The Generic Modeling Environment (GME) [1] is a meta-modeling environment that can be configured and adapted from meta-level specifications that describe a domain. The GME supports a set of generic modeling concepts to represent entities, relationships and attributes. An “atom” is the most basic type of entity that cannot have any internal structures. A “model” is another type of entity that can contain any other modeling types. A “connection” represents the relationship between two entities. “Attributes” are bound to entities and relationships and are used to store state.

In our previous work [8], we have made the observation that crosscutting concerns emerge in domain-specific models, as shown in Figure 1. For example, it is often the case that the meta-model forces a specific type of decomposition, such that the same concern is repeatedly applied in many places, usually with slight variations at different nodes in the model.

Examples of crosscutting modeling aspects include constraints [8], concurrency and state management [9]. Crosscutting concerns that are distributed across a model hierarchy often impede the comprehension and maintenance of a model, such as:

- Discovering or understanding a specific concern representation spread over the model hierarchy is difficult, since the concern is not localized in one single module.
- Changing a concern requirement is also difficult and time-consuming, because the designer must go into each relevant model and modify the specific elements one by one.

Manual inspection of models to discover potential aspects is a laborious task. Performing aspect mining techniques to existing non-aspectized models can offer insight into the identification of emergent aspects. Aspect mining from a modeling perspective allows the designer to locate the places in a model that must be changed when a particular concern needs to be modified.

This article provides a contribution representing the first modeling tool to provide aspect mining. The remainder of the paper discusses two different approaches that we have implemented to realize aspect mining on models.

Pattern matching for aspect mining

The pattern matching process is conducted by a human designer who suspects the existence of aspects in a model. The designer has to comprehend the domain information contained in a model and provide a pattern (“seed”) to indicate properties of potential
aspects. Such a seed serves as the starting point for discovering all matched concerns. There are two different representations of the seed for aspect mining of models through pattern matching. One representation is based on a textual expression, and another kind of seed is described by graphical models.

**Textual-based pattern matching**

In our past work [14], we used XPath expressions as the pattern description to search for properties within domain-specific models. The underlying search engine parses the XPath expression and traverses the internal representation of a model to compare the user-defined pattern with every model entity. This modeling search technique can be adapted to perform aspect mining. Although this technique is easily implemented and provides lightweight search power for simple textual pattern expressions, it lacks the capability to deal with complex patterns (e.g., a collection of sub-models that involve heterogeneous model elements and sinuous relationships among them).

**Graphical-based pattern matching**

Another approach to identify crosscutting modeling concerns is to represent a pattern in a graphical notation. As an example of this type of pattern matching, GReAT [2] defines a graph pattern specification language to express complicated patterns with a fixed and variable cardinality. This graph notation complements the shortcomings of textual pattern expression and supports complex and dynamic pattern matching.

**Clone detection for aspect mining**

Pattern matching techniques assist users in efficiently locating predefined crosscutting concerns. However, users of pattern matching are required to have a considerable amount of knowledge about the domain and overall model structure. The users must input a particular format of “seed” so that the aspect mining process can be partially automated. Moreover, pattern matching cannot explore unknown classes of crosscutting concerns (i.e., those for which no seed is known) and will often result in missing some desirable aspects. In order to overcome the deficiencies of pattern matching, we have investigated a clone detection technique applied to models. The intention of clone detection is to reveal the unknown crosscutting concerns through full automation of the aspect mining process. Clone detection identifies the maximally similar (clone) sub-models throughout the model hierarchy, following the similar approach of clone detection in source code through analysis on abstract syntax trees (AST) [3].

### Table 1. Three levels of similarity

<table>
<thead>
<tr>
<th>Level</th>
<th>Atom</th>
<th>Model</th>
<th>Connection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 1</td>
<td>Meta-type</td>
<td>Meta-type</td>
<td>Meta-type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>containment</td>
<td>Source</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Target</td>
</tr>
<tr>
<td>Level 2</td>
<td>Meta-type</td>
<td>Meta-type</td>
<td>Meta-type</td>
</tr>
<tr>
<td></td>
<td>Name</td>
<td>Name</td>
<td>Name</td>
</tr>
<tr>
<td></td>
<td></td>
<td>containment</td>
<td>Source</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Target</td>
</tr>
<tr>
<td>Level 3</td>
<td>Meta-type</td>
<td>Meta-type</td>
<td>Meta-type</td>
</tr>
<tr>
<td></td>
<td>Name</td>
<td>Name</td>
<td>Name</td>
</tr>
<tr>
<td></td>
<td>Attribute</td>
<td>Attribute</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>containment</td>
<td>Source</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Target</td>
</tr>
</tbody>
</table>

In the context of metamodeling, an atomic modeling element (e.g., an atom in GME models) is defined by its meta-type, name, and a set of attributes. Correspondingly, a model consists of a set of atoms, submodels and connections. Three levels of similarity can be defined based on the meta-type, name, and attribute (see Table 1) of the model elements. Level 1 indicates the most liberal policy (i.e., two atoms are considered as clones as long as they have the same meta-type). Level 3 defines the most stringent rule (i.e., two models are considered clones only when they have the same meta-type, name, and attribute set). Furthermore, all modeling elements (represented as containment in Table 1) should be correspondingly recognized as a Level 3 clone. Level 2 represents a moderate clone detection philosophy (e.g., two connections are considered clones if their source and targets are Level 2 or Level 3 clones, and each connection has the same meta-type and name). Based on the above levels of similarity, the three steps of the clone detection algorithm for models is:

**Step 1. Meta-model preprocessing**

The first step involves the partition of the meta-model entities into different groups that need to be compared. Each group includes a set of the meta-type pairs, such as \((\text{MT-models}) : \text{(MT2-containment)}\), where \((\text{MT-models})\) is a collection of meta-types whose model instances comprise some common elements, and \((\text{MT2-containment})\) is the collection of model elements that \((\text{MT-models})\) share. Because \((\text{MT2-containment})\) is contained...
by more than one model, it has the potential to become one of the selected crosscutting concerns.

In Figure 2, the meta-type “ModelA” and “ModelB” share the element “AtomAB.” “ModelB” and “ModelC” both contain “AtomBC.” So the partition of the illustrated fragment of the meta-model would be:

{ModelA, ModelB} : {AtomAB}
{ModelB, ModelC} : {AtomBC}

The preprocessing of the meta-model partition will facilitate the desired steps of the algorithm, because during comparison, only those models that have the same meta-type or fall into the same group will be compared. Furthermore, only the shared elements of the two models should be compared. For example, imagine that there is one instance of “ModelA” and one instance of “ModelB.” In such a case, we only need to consider whether their shared atoms (instances of “AtomAB”) are clones. Any other irrelevant elements will not be considered.

Step 2. Model fragment comparison

The second step of clone detection in models determines if the model fragment pairs are clones by comparison. From the root of the model hierarchy, each sub-model is compared with the other sub-models that either have the same meta-type or fall into the same group in step 1. As an example, suppose a comparison is to be made between model instance X and model instance Y:

- If X and Y are of the same meta-type, every element inside should be compared correspondingly.

- If X and Y are in the same group, only their shared elements need to be compared. The comparison is followed by the three level similarity definition stated in Table 1. Each time, the atoms are compared first, then the models, followed by the connections.

- If X and Y do not have the same meta-type, and do not fall into the same group, it means that they cannot have an intersection; thus, further comparison is not necessary.

Step 3. Maximal similar sub-models grouping

For all of the clone elements that X and Y share, group them together as a common property named $P$. If $P$ is not null, the next task is to find out whether $P$ is already stored in the list of maximally similar sub-models. An efficient way to search for commonalities on a list is to construct a hash function $h(P)$, which computes the number of a bucket (hash value) based on $P$. The hash function will always return the same bucket number given the same $P$. If $P$ is not in the bucket $h(P)$, then X, Y, and $P$ will be added to this bucket. If $P$ is already in such a bucket, only X or Y will be added into the collection of the sub-models that share the same property $P$.

Step 4 - Aspect filtering

The maximal similar sub-models generated from the above steps (i.e., the initial result) may contain too much noise and need to be refined further. Based on our experimentation, we found that if one model entity in a maximal similar sub-model group has a connection (in or out) that does not fall into the same group, then this model entity is seldom considered as an aspect and can be filtered out.

Aspect Mining Embedded System Models

ESML is a domain-specific graphical language for modeling real-time mission computing embedded avionics applications. It has been defined within the GME and used on several DARPA funded research projects. The ESML provides the following modeling categories to allow representation of an embedded system: a) Components, b) Component Interactions, and c) Component Configurations. The ESML has been applied to Boeing’s Bold Stroke, which is a product-line architecture for a variety of military aircraft written in several million lines of C++. There are over 20 representative ESML models for all of the Bold Stroke usage scenarios that have been defined by Boeing. For each specific scenario within Bold Stroke, the components and their interactions via an event channel are captured by ESML models.
In our previous work [9], we manually performed aspect mining from our own domain experience. The manual approach was a tedious process that identified crosscutting concerns such as concurrency and state management. We manually extracted these concerns one by one in order to demonstrate aspect weaving at the modeling level, which led to model driven program transformation. Much time was spent in understanding the ESML model ontology to support the manual process of searching the model for crosscutting concerns.

In general, an ESML model has a tree-like hierarchical structure. Figure 3 partially illustrates the internal representation of an ESML model named “InteractionModel.” The model on the first layer is the root of “InteractionModel,” which specifies a particular scenario that involves certain configurations of various models. These models belong to the second layer. In this figure, only two component models are depicted on the second layer, i.e., “BM_UserInputComponentImpl” and “BM_OpenEDComponentImpl.” Several models and atoms representing the containments of the second layer models are depicted separately on the third layer. The fourth layer is the last layer shown in Figure 3. The solid line between any two layers represents containment, and the dotted line with an arrow represents connections that may occur on the same layer or across layers.

In the case where users have no knowledge of the system (or, they have some knowledge, but not enough to express textual or graphical patterns), the clone detection technique for aspect mining must be applied. According to the clone detection algorithm presented earlier, the level of similarity is set to Level 2 (i.e., only consider the meta-type and the name, regardless of what the attributes). The maximal similar sub-models of “BM_UserInputComponentImpl” and “BM_OpenEDComponentImpl” are:

\{data2_, Data2Cond, AddCondition\}
{data1_, LogOnRead, AddLog}
These concerns are circled with different colors in Figure 3, representing 5 different concerns. In addition, the last two groups both contain model entities that carry connections out of the group (e.g., “ANY_sub” in “BM_UserInputComponentImpl” and “ANY_pub” in “BM_OpenEDComponentImpl”). Therefore, these two elements (as well as their relationships in the group) should be filtered out. Thus, the resulting aspect candidates are:

{InternalLock}
{any_sub, ANY_ref, EventTyping}
{any_pub, ANY_ref, EventTyping}

By coincidence, these three concerns are exactly the same aspects we manually mined in our previous research on aspect weaving and model-driven program transformation [9]. For the sub-models “BM_OpenFunctionalFacetInterface” and “BM_OpenFunctionalFacet,” the maximal similar sub-models are:

{data2_, Data2Cond, AddCondition}
{data1_, LogOnRead, AddLog}
{InternalLock}

“SetData1” will be removed in the filtering process because it has connections coming into or going out from the group.

Future directions in model aspect mining

From our experience, it is advantageous to perform reengineering techniques, such as aspect mining, at different stages throughout the software development lifecycle and on software artifacts other than source code. This article presented our initial investigation into aspect mining on domain-specific models.

We investigated two approaches to aspect mining - pattern matching is useful for identifying the location of pre-known aspects, and clone detection assists in identifying unknown aspect candidates. The pattern matching technique is useful only when the users are able to offer a concern pattern (i.e., the “seed”), but the clone detection technique is more powerful because it can suggest multiple unknown aspects with little human interaction.

There are several areas that need additional investigation to further the maturity of model-driven aspect mining:

Noise filtering: The result of the clone detection is usually adulterated with too much undesired noise. Currently, we only use one filter layer that is based on connections. We are considering other metrics that will be integrated into the filtering analysis.

Visualization of Modeling Aspects: An aspect mining tool enables identification of the potential aspects and often provides the capability to visualize the various locations affected by an aspect. Traditional aspect mining techniques work on the source code level, thus their corresponding visualization tools are based on a graphical notation that is particular for line-oriented software statistics [10]. Because a model is a containment hierarchy of entities, it is necessary to develop a specific means to visualize the crosscutting aspects over different levels of models. Our future visualization tool will use a tree structure to display the model hierarchy natively with potential aspects highlighted across the whole structure. Users will have the option to expand or collapse any level of a specific model.

Model Refactoring: With respect to model refactoring, we have already implemented a model refactoring browser in GME by means of a model transformation engine [16]. The research on aspect-oriented refactoring is still under investigation, which aims to extract the mined crosscutting concerns into the separately described aspects.

References


