An AOP-based Programming Environment with Code Fragment Numbering Supporting Multi-Type Artifacts and Multiple Programming Languages

Jiyong Park and Seongsoo Hong

School of Electrical Engineering and Computer Science
Seoul National University, Seoul 151-742, Korea
{parkjy, sshong}@redwood.snu.ac.kr

Abstract. There is a growing demand for highly customizable complex software systems, such as real-time operating systems (RTOS), which are composed of existing modules. In an RTOS, crosscutting features frequently occur and they cannot be modularized using the object-oriented and/or component-based software technologies. Aspect-oriented programming (AOP) is a solution for this, because it allows programmers to modularize such crosscutting features via new modules called aspects. However, we have found that current AOP language implementations have limitations that make them unfit for customizing RTOSes. In this paper, we present enhancements which overcome these limitations. Also, we provide an aspect-oriented programming environment which supports these enhancements. It visualizes the crosscutting relationships of features and enables easy navigation of source code. This is often difficult when using most current AOP languages because the various source files are not easily integrated. We argue that our programming environment enables rapid, highly flexible, and easy customization of complex software systems.

1 Introduction

With the emergence of a wide variety of devices and application domains on the market, the demand for customizable software systems is increasing, since the development of an entire software system from scratch is inefficient and lengthens the time-to-market. In order to be easily customized, software needs to be modular so that application specific software can be built by a composition of selected modules that encapsulate specific features of the software.

Approaches to designing customizable software are often based on object-oriented programming (OOP) and/or component-based design (CBD) technologies in which the features of software are modularized as classes or components. However, it is not always possible to achieve full customizability with these approaches since neither OOP nor CBD can modularize all features in some software systems. Specifically, if a feature crosscuts other features that are already implemented as classes or

components the crosscutting feature cannot be modularized as a separate class or component; instead, its code is scattered throughout the program.

Aspect-oriented programming (AOP) [1] provides a novel method to explicitly describe a crosscutting feature via a separate module called an aspect. AOP, however, is not a replacement for object-oriented programming or component based design. The basic functional structure of a design is still described with an object-oriented language while crosscutting features are described with an aspect-oriented language, thus augmenting the object-oriented language. In AspectJ [3], the most famous AOP language, an aspect is implemented as code fragments for a specific feature and location descriptors which describes the specific contexts in which the code fragment will be executed.

Unfortunately, we have found that current AOP languages have some limitations when used in developing complex software systems like an RTOS. In an RTOS, there are many places that multiple features are intertwined inside a function or a method. Current AOP languages can not cleanly separate features of this kind. There are other limitations for current AOP languages as well, and we describe these limitations in detail in the Section [2].

In this paper, we present enhancements that overcome these limitations. The main enhancement is the code fragment numbering which permits insertion of a new code fragment at an arbitrary point inside a function or a method. Using this, programmers can separate crosscutting features that can not be cleanly separated using current AOP languages. We provide an aspect-oriented programming environment which supports these enhancements. The features of the programming environment are: (1) It adopts the concept of feature-oriented programming (FOP) [12], in which a system purely consists of modules representing features. This differs from AOP where a system consists of a base program and aspects. (2) The meaning of crosscutting is broadened to encompass multiple programs written in different languages (e.g., across source files, makefile scripts, and manual pages). This idea was first introduced by [11]. Most current AOP languages only apply to the modularization of crosscutting features contained in an individual program. (3) It visualizes the crosscutting relationships of features and enables easy navigation of source code. This is often difficult when using most current AOP languages because the various source files are not easily integrated.

The rest of this paper is organized as follows. In Section 2, we explain in detail the limitations of current AOP languages and programming environments when used for customizing complex software, especially an RTOS. From Section 3 to Section 5, we explain our programming environment and enhancements applied to it. Section 6 summarizes related work. Finally, we conclude this paper in Section 7.

2 Motivation

In this section, we describe the limitations of current AOP languages and programming environments when used for customizing complex software, especially an RTOS.

First, the current AOP languages can not insert code for a new crosscutting feature at an arbitrary location inside a method or a function. The insertion is possible only at well defined points which are called join points. For AspectJ, a join point is a method call, method return, attribute set, attribute get, etc. To the best of our knowledge, there is no AOP language that supports insertion of code at an arbitrary location inside a method or a function.

This level of granularity is too coarse to separate features in an RTOS. Methods in an RTOS are often quite long and typically contain many intertwined features. For example, the `do_fork()` function, which implements the `fork()` system call in the Linux operating system, contains code fragments for several features; resource limit, thread, swap, wait queue, signal, SMP support, file system, signal system, memory management system, and so on. Thus, to modularize such deeply tangled code programmers must be able to insert code at an arbitrary point inside a method.

```
int do_fork() {
    …
    dummy();
    …
}
void dummy() {}
```

```
aspect signal {
  before(): call(void dummy()) {
    /* copy signal mask from parent */
  }
}
```

Fig. 1. Trick to insert new code in the middle of a method using dummy method (written in AspectC)

Murphy et al. [5] proposed possible tricks to insert new code at an arbitrary point in a method when using AspectJ and Hyper/J [8]. Fig. 1 shows an example of AspectC [4], which is very similar to AspectJ. To demonstrate one such trick, we will use an example. Suppose that signal functionality is not implemented in Linux, and we want to implement it using AOP. To do this, we must add code that copies the signal mask from a parent process inside a `do_fork()` function, since this must be done during forking. We can add the code as advice of an aspect for signal functionality. Then, a dummy method invocation is inserted at the point where the new code is desired. With this, the advice is applied before or after the dummy method using the AspectC pointcut designator (`before(): call (void dummy())`).

However, this requires modification of the base program, by adding a dummy method invocation, thus breaking the modularity. Also, if the new code uses local variables of the method, the dummy method must transfer the required local variables to the aspect code which is bad for performance and is difficult to program. Furthermore, this kind of program is difficult to understand.

Second, since current AOP languages are mostly based on adding aspect code to separate aspect files, it is often very difficult to figure out how the aspect code is weaved together with the base program and in what order because the complete source of a particular method is scattered throughout multiple files. In developing an RTOS, it is necessary to know the exact execution sequence of the chosen features. Chu-Carrol et al. [7] stated that current compositional approaches to AOP have two problems: invisible semantic transformation and scatter. When an aspect is defined

for a certain section of the base program, the effect of this definition is not reflected in the base program source, thus transforming the semantics of the base program source invisibly. Since a complete method is assembled from the method defined in the base program and a collection of aspects that affects the method, the flow of a method is scattered and the programmer has to reconstruct it manually to fully understand the method.

There are tools that try to solve these problems. For example, AspectJ has a plug-in to the Eclipse IDE [9] that provides hyperlinks to the advice of aspects if there are aspects that are applied to a certain join point. This tool solves the problem of invisible semantic transformation, but still has the scatter problem, since the tool does not reconstruct a complete method from all aspect code that affects the method.

3 Programming Environment

In our programming environment, programming is done using four views: (1) system view, (2) feature view, (3) artifact view, and (4) content view. The first view provides the highest-level view and the last view provides the lowest-level view. When programmers add a feature to a software system (e.g., RTOS), they start from the system view and proceed through the lower level views.

The system view shows the entire system as a collection of features and their dependencies. Programmers can add, select, and deselect features in the view. From this view, the programmer can also browse a feature more closely by changing to the feature view of the chosen feature.

The feature view magnifies the previous view by grouping all features that have a direct relationship with the chosen feature. In this view, programmers can create new artifacts or modify existing artifacts. Here, an artifact refers to a module of a specific type. In object oriented languages, a class is considered an artifact. Programmers can then browse existing artifacts by changing to the artifact view.

From the artifact view, programmers can browse the content of artifacts (e.g., attributes and methods in the case when the artifact is a class) and new content may be added to the artifact. The final step is to fill in the details of the artifact's content, which is done in the content view.

The content view shows the internal structure of the content of an artifact. For example, for a class, programmers can browse and edit the internals of the methods in the class. A method consists of code fragments and new code fragments can be added or existing code fragments can be removed. We applied the code fragment numbering technique to this view to enable insertion of new code at an arbitrary location inside a method. We discuss the code fragment numbering and the details of its integration with the content view in Section 3.4.

In the following subsections, we explain each view beginning with the highest-level view first.

3.1 System view

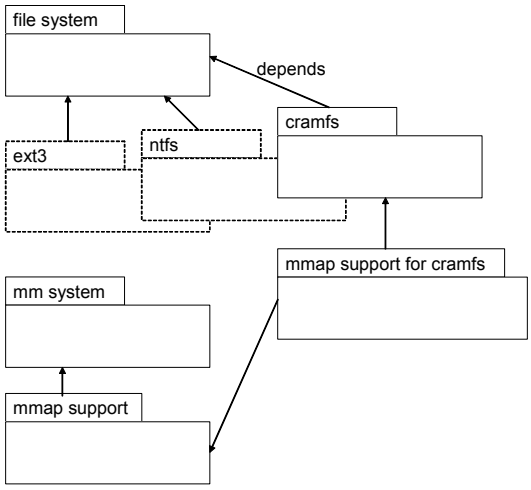

Fig. 2. System view for part of Linux operating system

Our programming environment considers the 'feature' as the basic unit of customization that represents a specific function of a software system. Examples of features found in an RTOS are networking, ARM CPU support, logging, or the enforcement of file system quotas. In fact, there is no functionality found in an RTOS that cannot be thought of as a feature. This concept is known as feature-oriented programming (FOP) [12]. This differs from AOP where a system consists of a base program and aspects. However, both AOP and FOP have the same goal of modularizing crosscutting features. We adopt the concept of FOP since it achieves better integration with RTOS than AOP does.

We want to build a software system that can be customized for a particular application by selecting the required features from a repository of predefined features. The features in the pool may be quite simple, or they may contain sub-features each of which can be turned on only when all its parent features are turned on. This relationship is described by a dependency graph. An example of this is a networking feature that has the TCP/IP protocol stack, BSD socket interface, and packet filtering as sub features. Likewise, the TCP/IP protocol stack sub-feature also can have TCP, UDP, and IP protocols as its sub-features. Inside a feature, a programmer can define a new artifact or modify an existing one. If another feature defines a fragment in the same artifact, the two fragments are merged into one complete artifact.

In the remainder of this section, we explain our programming environment by example. In this example, we enhance Linux to support memory mapping when cramfs (compressed ram file-system) is used, as the original cramfs does not support memory mapping. We modularize the new enhancement as a feature and show how to add the features to an existing set of features. In order to understand our framework,

there is no need to understand the mechanisms of the Linux memory management or file systems.

The system view in Fig. 2 shows how to add the `mmap_support_for_cramfs` feature. Initially there are two top-level features that we are interested in, `file_system` and `mm_system`, which represent the file system and memory management system features, respectively. The `file_system` feature contains various file systems as sub-features, such as `cramfs`, `ntfs`, and `ext3`. The `mm_system` also has the `mmap_support` sub-feature, which is the memory mapping feature. Features that have dotted lines are unselected features, and will not be included in the final merged program. We must add the `mmap_support_for_cramfs` feature as a sub-feature of both the `mmap_support` and `cramfs` features since our new feature is meaningless unless both features co-exist.

3.2 Feature view

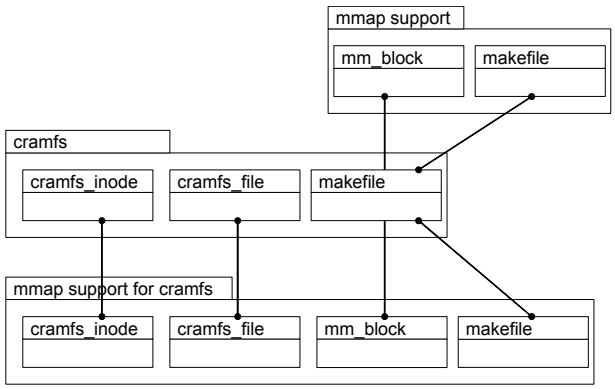

Fig. 3. Feature view of `mmap_support_for_cramfs` feature

The feature view of the example system is given in Fig.3. It shows all features that have a relationship with the specified feature and the artifacts in which the features define part of the artifact. For our purposes, two features are considered related if they contain artifacts of the same name. The new feature, `mmap_support_for_cramfs`, defines fragments in `cramfs_inode`, `cramfs_file`, `mm_block`, and `makefile`. Note that those artifacts are also defined in the other features `cramfs` and `mmap_support`.

The complete definition of such an artifact is the union of all partial definitions of the artifact which are defined by various features. Consequently, we can also regard an artifact with partial definitions in two or more features as having a crosscutting relationship. Thus, this view is helpful for analyzing the parts of the system that are affected by a specified feature.

3.3 Artifact view

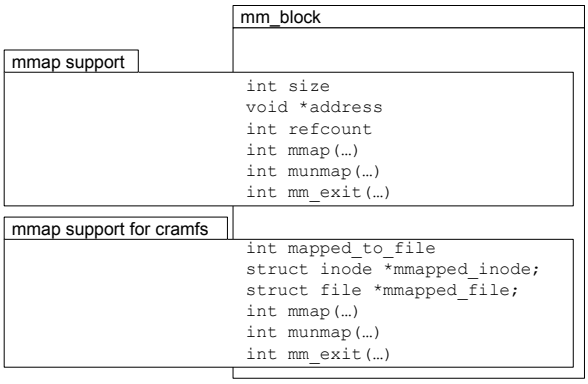

Fig. 4. Artifact view of `mm_block` class

As mentioned above, multiple features may crosscut an artifact. The artifact view shows an artifact and its content crosscut by multiple features. In our example, two features `cramfs` and `mmap_support_for_cramfs` crosscut the `cramfs_file` class that is an artifact. These two features define all attributes and methods that make up the content of the class.

Fig. 4 shows the artifact view of class `mm_block`, which is crosscut by the `mmap_support_for_cramfs` and `mmap_support` features. Initially, the `mmap_support` feature defines three attributes and three methods. The three attributes are the size, address, and reference count of a memory mapped block. The three methods are for mapping a block, un-mapping a block, and un-mapping whole blocks belonging to a process. The new feature `mm_support_for_cramfs` adds three additional attributes. The first records whether this `mm_block` is mapped to a file. If it is mapped, the two remaining attributes record the location of the inode and the file where this `mm_block` is mapped to.

3.4 Content view

This view shows the internals of the content of an artifact. Examples of content are methods in a class, actions in a makefile, and paragraphs in a manual page. Fig. 5 is the content view of the `mmap()` method in the `mm_block` class. In the figure, the code fragments are linked together. Each code fragment is tagged with the feature name that the fragment is defined in, and also has a short description of what the fragment is for. Note that simply tracking the link does not necessarily give the execution sequence. For example, since C programs contain branch statements the actual program execution may not follow the path directly from top to bottom.

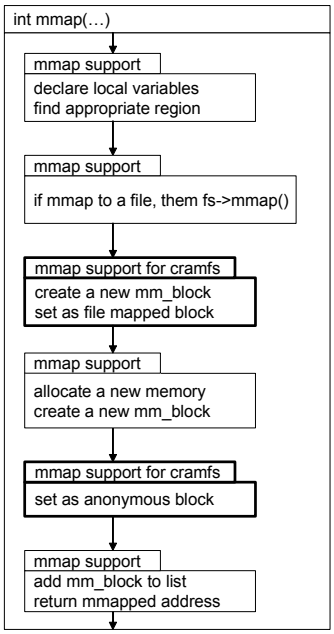

Fig. 5. Content view of method `mmap()`

This figure shows what happens to the `mmap()` method when the `mmap_supprt_for_cramfs` feature is applied. In order to set the attributes (`mapped_to_file`, `mmapped_inode`, and `mmapped_file`) added to the `mm_block` class, additional code needs to be inserted. If mapping is done by a file system mapping routine (`fs->mmap()`), then `mm_block` is set to a file mapped block (`mapped_to_file = 1`). If mapping is done by allocating a new empty area, then `mm_block` is set to an anonymous block (`mapped_to_file = 0`).

We mentioned the coarse granularity of existing AOP languages in the Section 2. In order to insert new code at an arbitrary location inside a method, we view a method as an ordered sequence of code fragments, not as an atomic and indivisible entity. Code fragments are then the basic building blocks of a method.

Code fragments consist of an ordered sequence of individual statements. We can categorize statements into two groups, those that do not have internal statements and those that do. The former, such as variable assignments and method calls, can not be divided. However, the latter, such as if, while, and for statements, can be divided. As shown in Fig. 6, the `if` statement (consisting of 6 lines) is divided into three code fragments. Note that code fragments 0 and 2 can not belong to different features because they are meaningful only if both exist. Here, `Feature X` denotes a base program and `Feature Y` denotes an aspect to the base program that checks a certain condition and does exception handling when the condition is not met.

In order to represent this kind of order information, code fragments are numbered in sequence. When features are merged, code fragments in them are sorted by sequence number. There are two different situations where code fragments are

inserted. Either a new fragment is inserted between existing fragments, or it replaces an existing fragment. Insertion between existing fragments is simply done by comparing the sequence numbers of the fragments. For example, if there exist code fragments numbered as 0 and 2 and new code fragment is numbered as 1 then we know that the new code fragment is supposed to be inserted between the two existing code fragments.

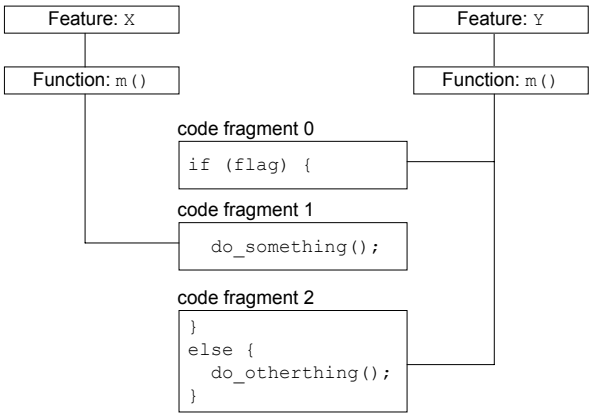

Fig. 6. An example of a method consisting of code fragments

In general, the second case is more complicated. To model the replacing relationship, we let code fragments that are defined in different features have a major number and a minor number. Ordinarily, they are sorted by their major number. If two of them have the same major number, the one that has the larger minor number replaces the other. If their minor numbers are also the same then it is assumed that they represent a common code fragment for the two features, and their code will be identical.

Fig. 7 is an example of code fragment replacement. There are two features, `feature X` and `feature Y`, and both of them have a method named `method M`. In both versions of the method, there are several code fragments and these are numbered. Since the two methods have the same method name, the two definitions should be merged. While merging `feature X` and `feature Y`, the code fragments, `code frag.D` and `code frag.F` are encountered which both have the same major number, 5. In this case, `code frag.F` replaces `code frag.D` since it has the larger minor number, 1

While this scheme enables insertion of new code at an arbitrary location inside a method, we realize that it has the following problem. Since we do not know how many code fragments will be inserted at a certain point, at some point there may be no room in the numbering scheme to insert new code. In that case, the sequence numbers should be shifted to make room for the new code. This shifting and sorting of the sequence numbers is very tedious and error prone and may require an unacceptable amount of work.

Therefore, there is a need for automatic management of this code fragment numbering. Our programming environment does just this. The existence of sequence numbers is hidden from programmers, and they only see the sorted code fragments. Also, since the average number of code fragments in a single method is small (typically from 1 to 10), the overhead associated with the shifting operation is not significant.

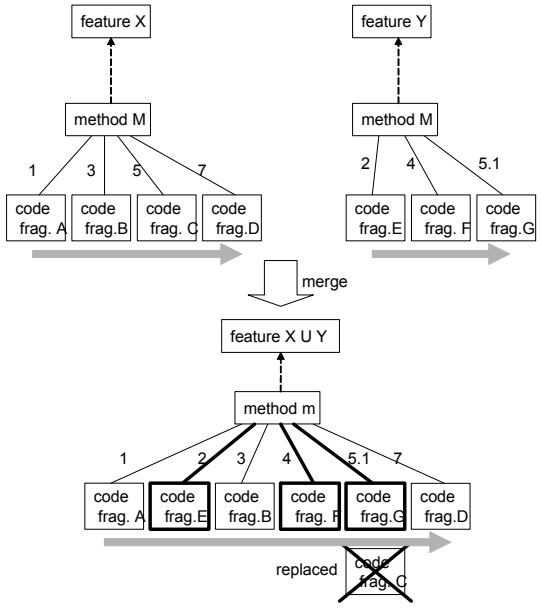

Fig. 7. How the code fragment numbering works

Programmers can insert code fragments by cutting a link between code fragments or by splitting a code fragment into two separate fragments. In addition to inserting a new code fragment, direct editing of a code fragment is possible in this view by further expanding a fragment to show the actual statements and from here the programmer can edit the statements. Code fragments from inactive features are usually not visible, but programmers can configure it so that they are shaded. Consequently, our scheme does not hurt readability while visualizing how different features are weaved together.

4 Additional features of the programming environment

Most complex software systems consist of multiple programs written in more than one language. For example, an RTOS is a group of collaborating programs; kernel, boot loader, shell, libraries, and other utilities. Each program can be written in a different language and some of them may even be written in non object-oriented

languages. Furthermore, a single program is not made up of only source code. Besides source code, a program may additionally require other artifacts such as build scripts (e.g., makefile), manual pages, or HTML files, which are not object-oriented. These also can be manipulated by our programming environment. When a feature is added to the source code, build scripts and other artifacts of the program may also change to adapt to the modifications made in the source code, and vice versa. Since this is the case for most systems, we conclude that a feature should have the following two capabilities.

- Modularize non object-oriented artifacts
- Modularize code scattered across multiple artifacts and programs

In the following sub sections, we explain how these two capabilities can be realized.

4.1 Modularizing non object-oriented artifacts

In order to modularize non object-oriented artifacts into features, we use the ideas introduced in [10], [11]. In the paper, Batory et al. treat non object-oriented artifacts as classes and use them in an analogous manner to object-oriented code. They argue that most types of artifacts have, or can have an equivalent class structure and thus are object-based.

However, we have found that their solution has the following problems. First of all, they do not provide a method for using non object-oriented artifacts as classes except for the case of makefile. Since there is no general rule to transform arbitrary types of artifacts to class-like structures, their solution is quite limited.

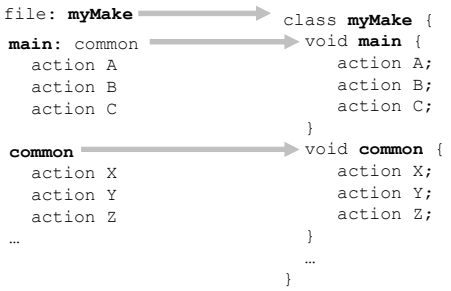

Fig. 8. Makefile and its equivalent class

Second, their method for treating a makefile as a class hides too many characteristics of the makefile. Fig. 8 taken and modified from [14], shows a makefile and its equivalent class. A makefile consists of targets and a sequence of actions for each target. In their method, each target name corresponds to a method name and actions in a target correspond to the statements in a method. However, a target may depend on other targets. By using the fore mentioned approach in this case, it is not

possible to represent these dependencies even though this is a very important characteristic of a makefile.

As an ad-hoc solution for this, prerequisite targets can be represented as arguments of a method. If a programmer wants to add more prerequisite targets to an existing target, which is often the case in practice, it is analogous to adding parameters to a method. However, the resulting class cannot be modularized by a feature since methods with the same name but different argument lists are considered different methods in object-oriented programming, and so cannot be merged into one method. In our solution, rather than trying to make an equivalent class for each type of artifact, we expose each type of artifact to a feature. Then a rule for merging multiple fragments of a particular type is provided for each type of artifact.

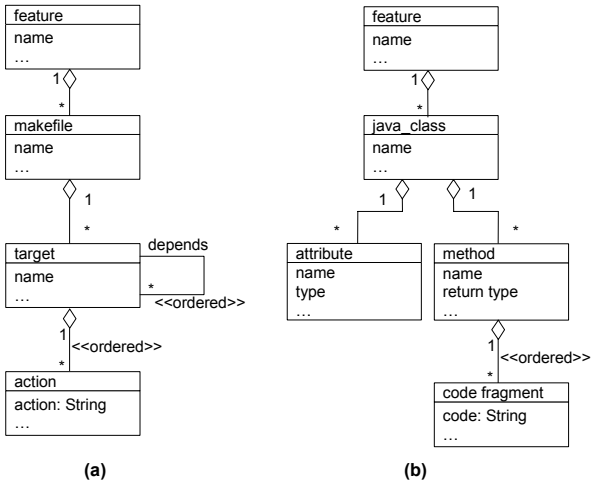

Fig. 9. Model for (a) makefile and (b) java class

Before defining the merge rule, the structure of the artifacts should be modeled. In Fig. 9, there are two class diagrams, that of a model of a makefile and that of a Java class. In the makefile model, there are several targets in each makefile and these targets may have dependencies between them. For each target, there are actions that make the target. Here it should be noted that the 'depends' relationship and relationships between target and action are stereotyped as <<ordered>>. The ordered relationship is labeled with a sequence number that represents the relative order of the objects targeted by the relationship. For example, in Fig. 10. (a) and (b), `feature1` and `feature2` are two instantiations of the previous makefile model. In `feature1`, the target named `main` depends on the target named `common` and the target named `clean`. Since, sequence numbers for `common` and `clean` are 1 and 3, `common` proceeds `clean`. That is, actions in `common` should be executed before the actions in `clean` are executed.

Once the model is provided for a specific type of artifact, the merge rule is similar among artifacts and quite straightforward. For non-ordered relationships, objects are

merged by the union operation and for ordered relationships, sorting the sequence numbers merges the objects.

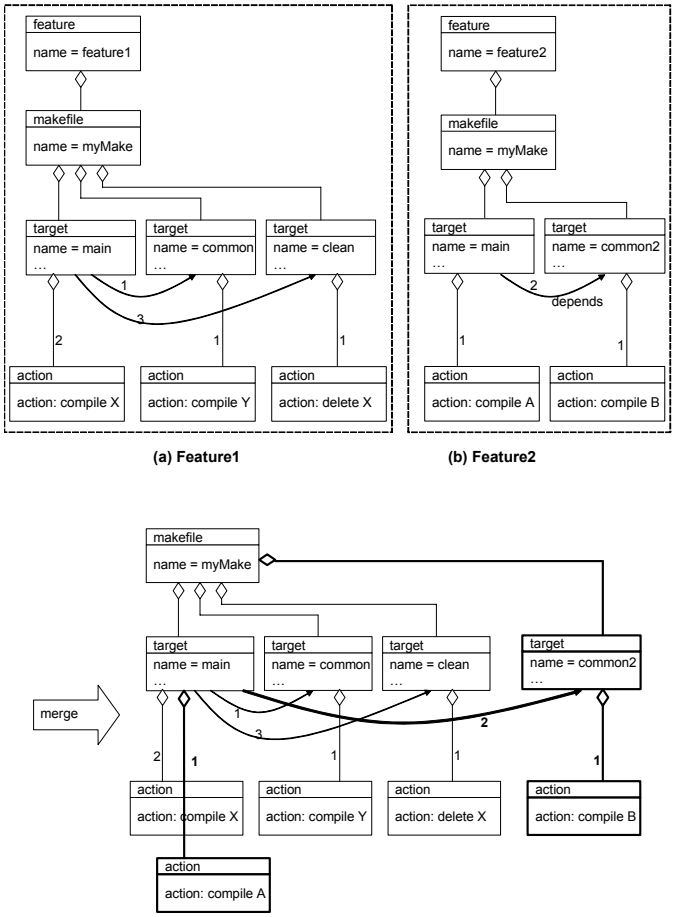

Fig. 10. Two makefiles are merged into one

`Feature3` in (c) is the result of merging the two makefiles. Since the relationship between the makefile object and the target object is non-ordered, the target objects in the merged makefile are simply the union of the target objects in the two makefiles. However, the dependency relationship between target objects is ordered. In this case, the dependency list of the target object named `main` is a sorted union of the dependency lists of the two versions of `main` found in the two makefiles. This also applies to the relationship between the target object and its action objects.

Using this technique, any type of artifact, object oriented or otherwise, can be crosscut by a feature. A new type of artifact can be added to the programming

environment by providing a plug-in module. For this, we expose APIs that can be used when making such modules.

4.2 Modularizing code scattered across multiple programs

We also adopt and enhance the idea introduced in [11] for modularizing code scattered across multiple programs. With this, to allow a feature to crosscut multiple programs the feature contains sub-features, each of which crosscut an individual program. Since each program is stored at a different place (e.g., directory), the sub-feature should have the information concerning the location of the program to which it is applied. When the sub-features are merged, their directory hierarchies are also merged as well.

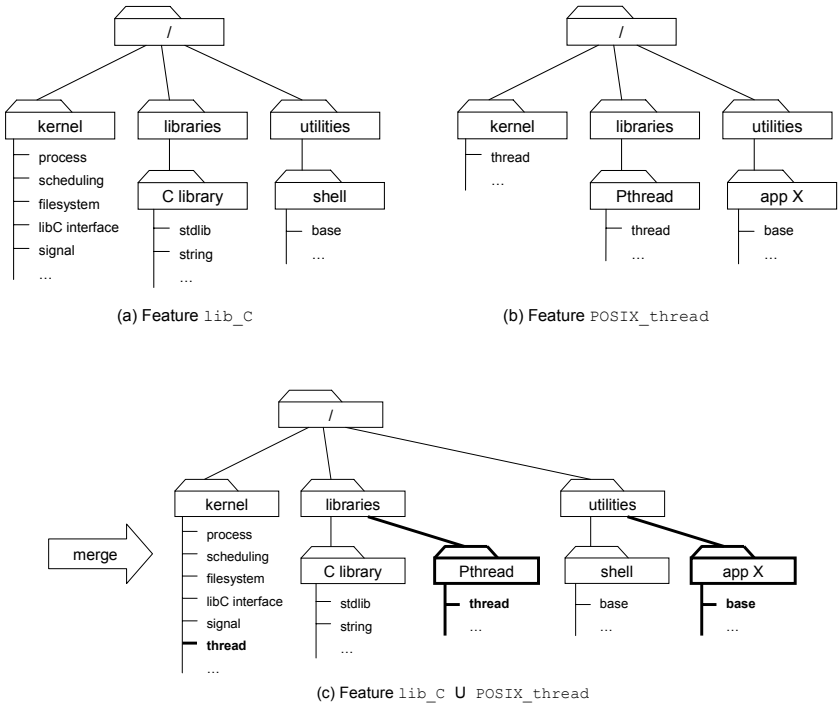

Fig. 11. Merging features that crosscut multiple programs

Fig. 11 shows an example of such features that crosscut multiple programs. Feature `lib_C` (a) is a feature for a shell utility. Since the shell depends on the C library, and the C library requires related services from the kernel, the feature must crosscut the shell, C library, and kernel. Feature `POSIX_thread` (b) is a feature for the `appX` utility. Let us assume that `appX` is an application that uses multi-threading, and the thread library and kernel are required to support multi-threading in a process. So the `POSIX_thread` also crosscuts multiple programs. They are `appX`, the thread

library, and the kernel. In `lib_C`, `kernel`, `libraries`, `utilities`, `C library`, and `shell` are directories. Each of these represents a program, and in each directory there exist sub-features for the program. `POXIS_thread` is of the same structure but the directory hierarchy differs. Merging `lib_C` and `POSIX_thread` produces a new composite feature (c), whose directory hierarchy is a union of those of `lib_C` and `POSIX_thread`. In this example, the `pthread` library is introduced in the libraries directory, `appX` is added to utilities, and in the `kernel` directory the thread feature is added. Thus the `kernel` now supports multi-threading.

Although the methods described are quite simple, we believe that they broaden the scope of a feature allowing it to be effectively applied to today's complex software systems that consist of multiple programs containing many types of artifacts.

4.3 Fine grained customization

In our framework, a feature is the unit of customization. We can customize software by selecting or deselecting features. However, we have found that systems cannot be characterized purely by the on/off status of features. For example, the `flash_device` feature requires configuration information such as start address, end address, and block size of the flash device, and the `Priority_scheduling` feature may be further customized by defining the number of priority levels that the scheduler supports. This information cannot be expressed by a binary mechanism such as turning on/off a feature. If these parameters are in-lined in the code, the level of customizability is severely decreased.

Thus, we extend the feature so that it can take parameters. For example, the `initial_ramdisk` feature has, '`name of the initial ramdisk device`', as its argument. A parameter consists of its name and value, and the value is a string. In the feature code, whenever the parameter name is encountered, it is replaced by the value of the parameter. This is quite similar to the use of macros in the C language or constants in Java. Although this is a very simple technique, to the best of our knowledge, there is no AOP language that exposes parameters at the design level.

5 Implementation

The overall structure of our programming environment is shown in Fig. 12. In the figure, the programming environment deals with the data stored in the configuration files and feature files. On the other side, there is a viewer/editor which is the interface for the programmer. The merged files are outputs of the programming environment, and are compiled by conventional compilers. The programming environment consists of a kernel which is independent on artifacts and plug-in modules each of which implements artifact dependent jobs such as parsing.

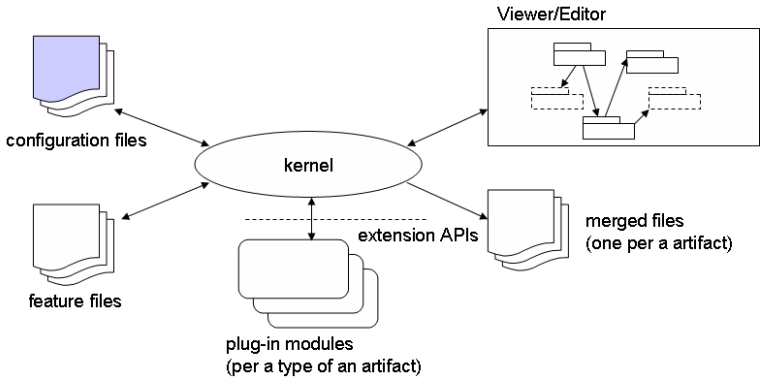

Fig. 12. Structure of the programming environment

The configuration files are used to save application-specific configuration information. For example, there may be configuration files for a network gateway, PDA, or a robot controller. A feature file stores the actual code for a specific feature and also contains meta-information about its relationship with other features. The meta-information is a feature name, dependencies with other features, and most importantly major/minor numbers of each code fragments that belong to a feature. There is an XML file for each feature which stores this meta-information.

The Viewer/Editor is the interface for programmers through which they can configure and modify a software system. It also receives the analyzed model of the system from the kernel and renders the model in a visible form. All modifications made to the visible form are mapped to corresponding operations on the model in the kernel. For example, when a programmer inserts code inside an existing method fragment, the original code fragment is divided into two and the inserted code becomes a new fragment that is placed between the two fragments of the original code. Major/minor numbers of the three code fragments are modified according to their order.

The kernel is the core of the programming environment. It manages the in-memory model of the feature files and configuration files. The model has a tree form, in which a node represents a particular language structure (e.g., a class, a method, a code fragment or an action in makefile). However, the kernel is not concerned with the details of each artifact. This is done by the plug-in modules of each artifact. The plug-in module parses artifact source code and creates a model of the source code for the kernel. Also, the plug-in modules give information about how to merge the contents of a particular artifact type. The kernel provides extension APIs for these purposes.

6 Related work

Previous attempts to modularize features of complex software like operating systems are described in [2], [5], and [6]. In [2], four crosscutting features of FreeBSD

operating system are modularized using AspectC and it is shown that AOP mechanisms can cleanly modularize crosscutting features without altering the original code and that the performance penalty caused by AOP is negligible. While the paper has shown that AOP can modularize some crosscutting features of an operating system, we find other crosscutting features that can not be modularized using current AOP languages.

Colyer et al. [16] modularized the EJB feature from a very large application server using AspectJ. While they showed that AspectJ can be used on very large and complex systems, they used the workarounds introduced in [15] and Section 2 in this paper that requires modification of the original source code.

As discussed in [17], the current version of AspectJ has limited capacity to specify join points. Josh [17] is one approach to increasing the flexibility of join point specification. Their approach is based on the open compiler technology javaassist, a compile-time reflection library [18]. Josh allows programmers to make a new join point designator. Therefore, programmers are not limited by the built-in join point designators in AspectJ. However, it still does not support insertion of new code at an arbitrary location in a method since it only exposes 6 kinds of expressions, and not all possible expressions that could be found in a method.

Batory et al. in [10], [11], broadens the scope of modularity from only encompassing the source code of a program written in a specific language, to the inclusion of different representations of a system. In their work, a feature can crosscut non-object-oriented programs such as makefiles and HTML files. They transformed these files to have the structure of a class, and then used them in an analogous manner to object code.

Our work, especially visualizing features and direct programming on the visual representation, is greatly motivated by [7] where Chu-Carrol et al. introduce the notion of visual separation of concerns (VSC). It offers separate views of crosscutting aspects, allowing programmers to read and edit an aspect in isolation while leaving the semantic structure of the system untouched.

This paper goes one step beyond our previous work on enhancing AOP for customizing RTOS [13]. In the previous work, we concentrated on a visual programming environment that could clearly show how a program works in weaved form, while distinguishing each code fragments from different aspects.

7 Conclusion

In this paper, we have presented limitations of current AOP languages to be used in modularizing crosscutting features of complex software systems such as RTOSes. The main limitations are that they can not separate crosscutting features intertwined in a method. We have presented enhancements to the existing AOP and a new aspect-oriented programming environment that incorporates the enhancements and provides easy navigation of the source code. The enhancements can be summarized as follows: (1) Code fragment numbering enables fine grained customization of complex software systems. It allows programmers to insert code for a new feature at an arbitrary location. Current AOP languages have the limitation that code insertion can

only be done at a certain set of predefined locations. (2) The code fragment numbering mechanism is untenable for programmers since an insertion may require the shifting of the major and minor numbers of other code fragments. Our programming environment hides this complexity from programmers. It visualizes relative order between code fragments and maps modifications made to the visualized code fragments to the necessary code fragment numbering modifications. (3) Our programming environment also visualizes the crosscutting relationships of features and enables easy navigation of source code. (4) Finally, The scope of crosscutting is broadened to encompass multiple programs written in different languages. We argue that our programming environment enables modularization of crosscutting features in RTOS in a flexible and fine grained way.

We have implemented the kernel of the programming environment and a plug-in module for Java language. We are working on implementing the viewer/editor and a plug-in module for the C language to use it for customizing existing operating systems such as Linux and/or FreeBSD. The results look promising.

References

1. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier and J. Irwin, Aspect-Oriented Programming, In Proceedings of European Conference on Object-Oriented Programming, 1997
2. Y. Coady, G. Kiczales, M. Feeley and G. Smolyn, Using ApectC to Improve the Modularity of Path-Specific Customization in Operating System Code, In Proceedings of European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, 2001
3. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, An Overview of AspectJ, In Proceedings of the European Conference on Object-Oriented Programming, 2001
4. O. Spinczyk, A. Gal, and W. Schröder-Preikschat, AspectC++: An Aspect-Oriented Extension to C++, In Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems, 2002
5. Y. Coady and G. Kiczales, Back to the Future: A Retroactive Study of Aspect Evolution in Operating System Code, In Proceedings of Aspect-Oriented Software Development, 2003
6. Y. Coady, G. Kiczales, M. Feeley, N. Hutchinson, and J. S. Ong, Structuring operating system aspects: using AOP to improve OS structure modularity, In Communications of the ACM, 44 (10), 2001
7. M. C. Chu-Carroll, J. Wright, A. T. T. Ying, Visual Separation of Concerns through Multidimensional Program Storage, In Proceedings of Aspect-Oriented Software Development, 2003
8. Harold Ossher and Peri Tarr, Hyper/J: Multi-Dimentional Separation of Concerns for Java, In Proceedings of the 22nd international conference on Software engineering, 2000
9. Aspectj project, http://www.eclipse.org/aspectj
10. Don Batory, Roberto E. Lopez-Herrejon, and Jean-Philippe Martin, Generating Product-Lines of Product-Families, In Proceedings of Conference on Automated Software Engineering, 2002
11. Don Batory and Jacob Neal Sarvela, Scaling Step-Wise Refinement, In Proceedings of Conference on Software Engineering, 2003

12. C. Prehofer, Feature-Oriented Programming: A Fresh Look at Objects, In Proceedings of European Conference on Object-Oriented Programming, 1997
13. Jiyong Park, Saehwa Kim, and Seongsoo Hong, Weaving Aspects into Real-Time Operating System Design Using Object-Oriented Model Transformation, In 9th International Workshop on Object-oriented Real-time Dependable Systems, 2003
14. Don Batory, Tutorial for Feature Oriented Programming, http://www.cs.utexas.edu/users/schwartz/Batory-Tutorial.pdf
15. Gail C. Murphy, Albert Lai, Robert J. Walker, and Martin P. Robillard, Separating Features in Source Code: An Exploratory Study, In Proceedings of International Conference on Software Engineering, 2001
16. Adrian Colyer and Andrew Clement, Large-scale AOSD for Middleware, In Proceedings of Aspect-Oriented Software Development, 2004
17. Shigeru Chiba and Kiyoshi Nakagawa, Josh: An Open AspectJ-like Language, In Proceedings of Aspect-Oriented Software Development, 2004
18. Shigeru Chiba, Load-Time Structural Reflection in Java, In Proceedings of European Conference on Object-Oriented Programming, 2000