Weaving Aspects into Real-Time Operating System Design Using Object-Oriented Model Transformation

Jiyong Park, Saehwa Kim, and Seongsoo Hong
School of Electrical Engineering and Computer Science
Seoul National University, Seoul 151-742, Korea
{parkjy, ksaehwa, sshong}@redwood.snu.ac.kr

Abstract

Despite of the proliferation of object-oriented and component technology, their application to real-time operating systems (RTOS) has been limited since most design concerns in RTOSes crosscut software components and these are critical to deliver required performance and functionality. Aspect-Oriented Programming (AOP) is a very effective means to solve the crosscutting problem. However, we have observed the following limitations of the current AOP framework: (1) the current text-based AOP languages cannot clearly show how aspects are weaved together, (2) their granularity is too coarse to capture all aspects in an RTOS, (3) it is difficult to control the weaving process, since aspect weavers are usually hard-coded.

In this paper, we propose a new AOP framework that provides (1) a graphical aspect programming environment that visualizes aspects, crosscutting classes, and method structures, (2) a new aspect model that supports a sub-method level granularity where an aspect is defined as a set of classes, and (3) an aspect weaving process specified by an object-oriented meta-model transformation. Since our aspect-oriented programming framework improves the expressiveness of the crosscutting concerns of RTOSes and automates aspect weaver generation, it can enhance RTOS customization.

1. Introduction

In order to meet the increasing demand for application-specific embedded systems, embedded real-time operating systems (RTOS) need to be highly customizable to adapt to the varied applications' needs. The usual solution was the use of object-oriented and component-based technology. In this approach, the features of an RTOS are modularized as components, and programmers can then customize the RTOS by selecting, configuring, and binding the needed components. Unfortunately, this approach does not provide full customizability, because object-oriented and component-based technology cannot modularize all of the features of the RTOS. This is because some features such as scheduling, synchronization, fault-tolerance, and path-specific optimizations crosscut other features. These crosscutting features are critical to deliver required performance and functionality.

Using Aspect-oriented programming (AOP) [1], programmers can explicitly describe a crosscutting feature in a separate module called an aspect. The programmers describe the functional structure using an object-oriented language, and describe aspects using an aspect language. Then a code transformer called an aspect weaver applies the aspect code to the object-oriented language code and generates output source code that is also written in object-oriented language.

Unfortunately, we found that current AOP languages are not adequate for modularizing RTOSes. First, they are text-base languages, so they cannot clearly show how aspects are weaved together. Especially while designing an RTOS, it is necessary to see the program execution of woven code, and know which aspects are applied in which order at certain points without jumping between the code of various different aspects. Second, they provide only granularity at the level of a method, which is too coarse for separating aspects in an RTOS. The methods in an RTOS tend to be long and typically contain many features complexly intertwined. In order to modularize the deeply tangled code of features in an RTOS, granularity finer than the level of a method is needed. Third, most of the current AOP mechanisms are implemented in a dedicated language. This makes it hard to port an AOP mechanism to another language.

In this paper, we provide a graphical aspect programming framework to be used for the design and implementation of RTOSes. In this programming framework, an aspect represents a feature, which can be turned on or off. In an aspect, programmers can define multiple classes and the merging of definitions from multiple aspects introduces a complete definition of a class. This idea was motivated from open classes [12, 13]. An open class is one to which new methods can be added without editing the class directly. To achieve sub-method

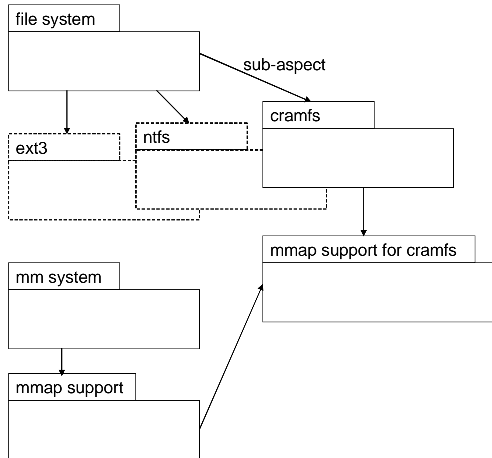

Figure 1. Aspect diagram.

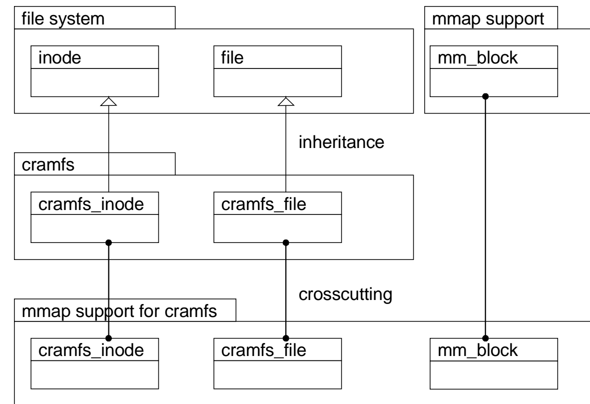

Figure 2. Crosscutting class diagram.

level granularity, we view methods as consisting of basic blocks and each basic block can belong to a different aspect. Our programming framework visualizes the above relationships, thus it is very easy to see how aspects are weaved together, while achieving sub-method level granularity.

We also present the detailed design of our aspect model and a description of the aspect weaving process. Current aspect weaving processes have the following limitations. (1) Because the aspect weaver is usually hard coded, programmers have no control over the aspect weaving process. (2) If programmers need to modify the weaver, they cannot avoid the complexity of the target language, such as C++ or Java. (3) Modifying the aspect weaver is difficult, tedious, time-consuming, and error-prone.

From these observations, we have decided to make the aspect weaving process modifiable and language independent. This idea is motivated by object-oriented model transformation that was introduced in [5]. In our framework, (1) an aspect weaver is automatically generated from specifications written by programmers. Thus, programmers can easily customize the aspect weaving process to meet their own needs. 2) The generated aspect weaver yields as its output a general object-oriented model that can be specified with UML [10]. Then this UML model can be transformed into the target source code via a commercial tool such as RoseRT [11].

The rest of this paper is structured as follows. In Section 2, we describe the design of our graphical aspect programming framework. In Section 3, we describe the design of our aspect model in detail. In Section 4, we specify the aspect weaving process using object-oriented model transformation. Finally, we conclude this paper in Section 5.

1.1 Related work

Previous attempts to modularize features of operating systems were introduced in [2, 6, 8]. In their work, they modularize four features; waking the page daemon, prefetching for mapped files, enforcing quotas for disk usage, and tracing blocked processes in device drivers in the FreeBSD operating system. For this purpose, they used aspectC [4], and showed that aspectC can modularize the features without altering other code, and the performance penalty caused by AOP is negligible.

Other research has attempted to represent aspects using UML [7, 9, 14]. In [9], they showed that the current UML specification was insufficient for modeling aspects, so they added the new modeling elements, pointcut and aspect. Unfortunately, their notation could not describe an open class, which is a very powerful AOP mechanism. They also did not support sub-method level granularity.

Our work was greatly motivated by [15]. This work introduced the notion of visual separation of concerns (VSC). VSC presents separate views of crosscutting aspects, allowing programmers to read and edit an aspect in isolation, while leaving the semantic structure of the system untouched.

2. Graphical aspect programming framework

In this section, we describe the design of our aspect programming framework that includes four diagrams: (1) aspect diagram, (2) crosscutting class diagram, (3) class structure diagram, and (4) method structure diagram. The first diagram provides the highest-level view and the last diagram provides the lowest-level view.

When programmers add an aspect to an operating

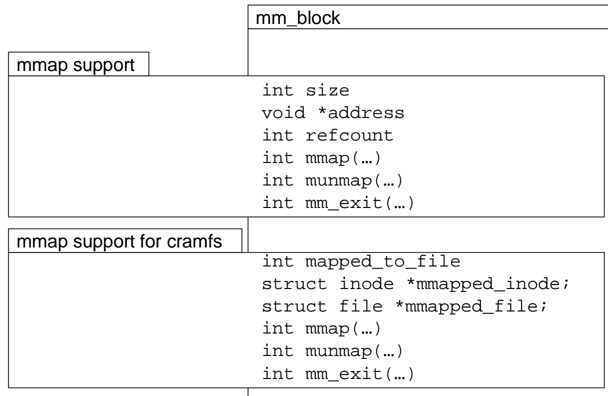

Figure 3. Class structure diagram (definition of `mm_block` class).

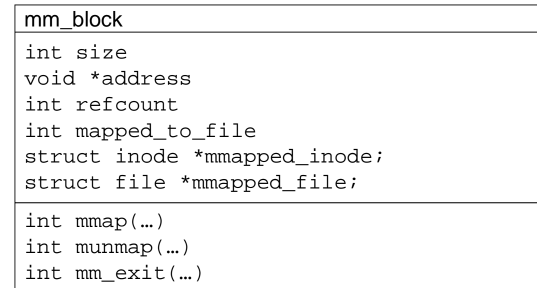

Figure 4. Class structure diagram (complete definition of `mm_block` class).

system, they start from the aspect diagram and proceed to lower level diagrams. The aspect diagram shows the entire system as a collection of aspects and their dependencies. Programmers can add, activate, and deactivate aspects in the diagram.

Then, the crosscutting class diagram magnifies the view by grouping aspects that have a direct relationship with a certain aspect. In this diagram, the programmers specify which classes the aspect crosscuts. After specifying that, the programmers can construct a class in the class structure diagram. In that diagram, attributes and methods are added to the class.

Finally, the method structure diagram shows the internal structure of a method as a collection of blocks. In this diagram the programmers can freely change the structure of a method by adding code to the individual blocks. By dividing methods into blocks, sub-method level granularity is achieved. In the following section, we will explain each diagram beginning with the highest-level view first.

2.1 Aspect diagram and crosscutting class diagram

In our programming framework, an aspect represents a feature. Networking, ARM CPU support, logging, and the enforcement of file system quotas are examples of aspects. RTOSes consist entirely of aspects. Programmers can customize an RTOS by turning aspects on or off. An aspect also can have sub-aspects that can be turned on only when all its parent aspects are turned on. For example, a networking aspect can have the TCP/IP protocol stack, the BSD socket interface and Packet filtering as sub-aspects. Likewise, the TCP/IP protocol stack sub-aspect also can have TCP, UDP, and IP protocols as its sub-aspects. Inside an aspect, programmers can define multiple classes. If another aspect also defines the same class, the two definitions are merged into one class definition.

The aspect diagram in Figure 1 shows how to add the `mmap support for cramfs` aspect. Initially, there are two top-level aspects that we are interested in, `file system` and `mm system`, which represent the file system and memory management system features. The `file system` aspect contains various file systems as sub-aspects, such as `cramfs`, `ntfs`, and `ext3`. The `mm system` also has the `mmap support` sub-aspect, which is the memory mapping feature. Aspects that have dotted lines are inactive aspects. They are turned off. We can add the `mmap support for cramfs` aspect as a sub-aspect of both `mmap support` aspect and `cramfs` aspect, because our new aspect depends on both the two aspects.

The crosscutting class diagram in Figure 2 shows aspects that have a relationship with the new aspect. In this figure, we can see how the new aspect extends multiple classes through multiple aspects. The new aspect `mmap support for cramfs` defines the `cramfs_inode`, `cramfs_file` and `mm_block` classes. Note that those classes are also defined in the other aspects `cramfs` and `mmap support`.

The complete definition of such a class is the union of all class definitions. So, we can regard the classes that belong to two or more aspects as having a crosscutting relationship. This relationship has no direction, whereas an inheritance relationship is directional. Class definitions linked together with a crosscutting relationship form a complete class definition.

2.2 Class structure diagram

As mentioned above, multiple aspects may crosscut a class. As mentioned above, multiple aspects may crosscut a class. In our example, two aspects, `cramfs` and `mmap`

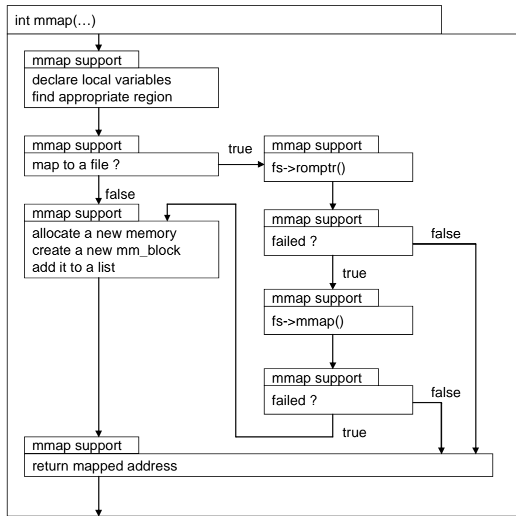

Figure 5. Method structure diagram (before adding **`mmap support for cramfs`** aspect).

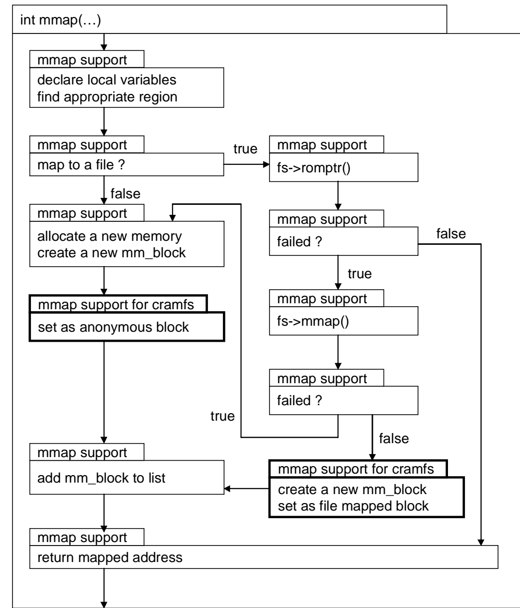

Figure 6. Method structure diagram (after adding **`mmap support for cramfs`** aspect).

`support for cramfs`, crosscut the `cramfs_file` class. These two aspects define each attribute and method inside the class.

Figure 3 shows the definition of class `mm_block`, which is crosscut by the `mmap support for cramfs` and `mmap support` aspects. Figure 4 shows the complete class definition created by merging these two aspects. Initially the `mmap support` aspect 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 aspect, `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.

2.3 Method structure diagram

Just as multiple aspects can crosscut a class, multiple aspects also can crosscut a method. This allows the internal structure of a method to be modified by aspects, thus achieving granularity finer than the level of a method. A method consists of basic blocks connected together. We have defined two types of basic blocks: code blocks and conditional blocks. A code block is a sequence of code that does not contain a conditional branch statement, and a conditional block is a conditional branch statement. The control flow of a method is structured by basic blocks and their connections.

Figure 5 shows an example of basic blocks inside the `mmap()` method of the `mm_block` class. Each basic block is tagged with the aspect that it belongs to. A basic block can be represented by simple comments about its function. Programmers can expand a basic block and see the actual code in it and from here they can edit the basic block. Usually, the programming framework ignores the code inside a block. However, if the programming framework detects a conditional statement inside a block, that conditional statement is automatically split from the basic block and becomes a new conditional basic block.

Figure 6 shows what happens to the `mmap()` method when the `mmap support for cramfs` aspect is applied. In order to set the attributes added to the `mm_block` class by the new aspect (`mapped_to_file`, `mmapped_inode`, `mmapped_file`), additional code needs to be inserted. If mapping was 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 was done by allocating a new empty area, then `mm_block` is set to an anonymous block (`mapped_to_file = 0`). Programmers can add basic blocks for this code by cutting a link between basic blocks or by splitting a basic block into two basic blocks. Then, the basic blocks can be populated with code. Basic blocks from inactive aspects are not visible, or programmers can configure it so that they are shaded. Consequently, our scheme does not hurt readability while visualizing how

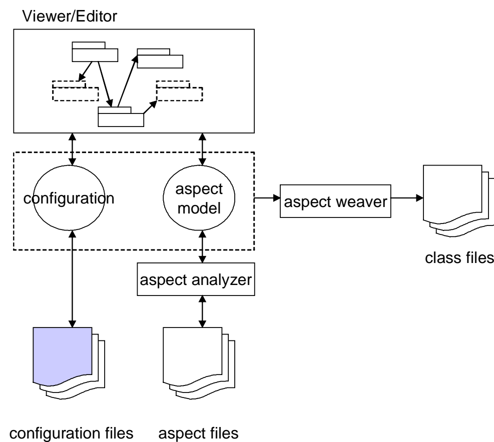

Figure 7. Architecture of the programming framework.

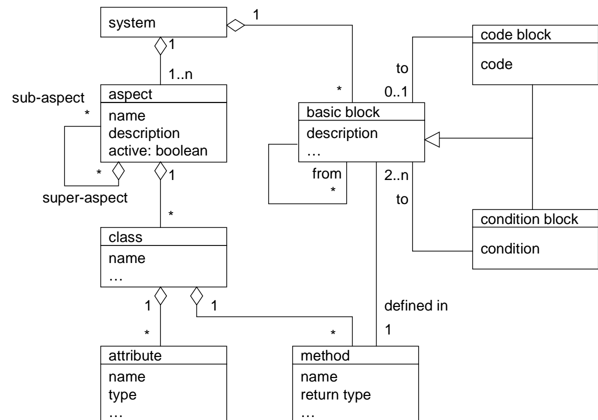

Figure 8. UML class diagram of our aspect model.

different aspects are weaved together.

2.4 Architecture of our framework

Now, we describe the architecture of our graphical aspect programming framework shown in Figure 7. The programming framework consists of two views. One is the current configuration that specifies whether a certain aspect is turned on or off. The other is the aspect model, which describes what aspects are in the system and their definitions. The detailed design of the aspect model is provided in Section 3. Using the current configuration and the aspect model, the viewer shows various diagrams previously introduced. Using the editor, programmers can configure an RTOS by turning aspects on or off. They can add or delete aspects, and input the actual code inside basic blocks.

The current configuration can be saved to a file and the aspect model is also stored in aspect files for later use. In our framework, we have chosen to store a single aspect in a single file. However, the exact format of the aspect file has not yet been decided. Simple `#ifdef` methods in C/C++ can be used, or some aspect language such as AspectJ [3] can also be used. In either case, the aspect model remains unchanged. The aspect analyzer absorbs language and aspect technology dependencies.

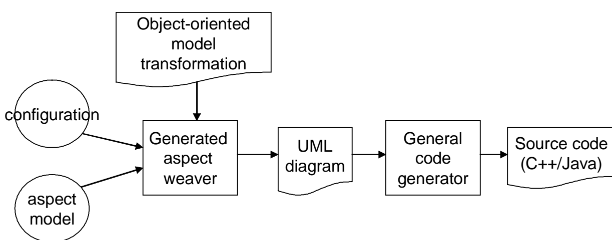

Figure 9. Aspect weaving process.

The aspect weaver generates executable target code from the current configuration and aspect model. The detailed operation of the aspect weaver is described in Section 4.

3. Detailed specification of the aspect model

In this section, we describe our aspect model in detail. Figure 8 is the design of our aspect model described in a UML class diagram. The top-level class is the system. In the system there exist many aspects, each representing a specific feature. An aspect has a unique name and a description and it also has a boolean flag that indicates whether the aspect is activated or not. An aspect can have multiple sub-aspects and a sub-aspect may depend on multiple super-aspects. A sub-aspect can be activated only when all its super-aspects are activated.

An aspect consists of multiple class definitions. An aspect may also have no class definitions; in that case the aspect is used only for grouping sub-aspects. Each class has its name, but the name is not necessarily unique in the system, because a class in another aspect may have the same name.

Inside a method basic blocks are defined, and the basic blocks are connected to each other. Since a basic block may be connected to a basic block that belongs to a different aspect, a basic block is not a member of the method, but of the system. However, a basic block has a relationship with a method to indicate in what method it is defined.

Basic blocks can be one of two sub-types, a code block and a condition block. The main difference between them is that the code block is connected to zero or one other basic block whereas the condition block is connected to two or more basic blocks, depending on its condition. Each basic block has a description about what it is for, as

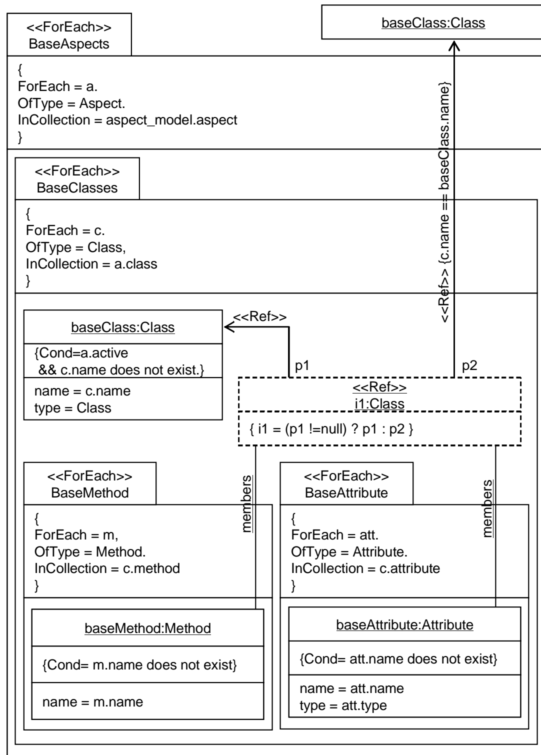

Figure 10. Extended UML object diagram that specifies meat-model transformation.

well as the actual code or condition that is written in the target language.

4. Aspect weaving process

Now we will explain the aspect weaving process. In Figure 7, the aspect weaver is a tool that generates class files written in the target language from the configuration and aspect model. Rather than using a hard-coded aspect weaver, we use an aspect weaver automatically generated from an object-oriented meta-model transformation specification, as in Figure 10. If we use a hard-coded aspect weaver, the following limitations arise. (1) The aspect weaver is tied to a target language. Thus, programmers have no control over the process of aspect weaving. (2) Modifying the aspect weaver cannot be done without dealing with the complexity of a target language. (3) Modification of the weaver is difficult, error-prone, and time-consuming.

In our aspect weaving process, programmers specify aspect weaving using an object-oriented meta-model transformation. We give a sample specification in Figure 10. This specification is transformed into the actual aspect weaver. This aspect weaver takes the current configuration and aspect model as inputs and generates UML diagrams. In turn, UML diagrams can be transformed into target source code by the general code generator provided by a commercial tool such as Rose.

For object-oriented model transformation specification, we have adopted the extended object diagrams proposed by Milicev [14]. We have also slightly extended the diagrams of Milicev since they do not support the notion of importable instances. Specifically, we extend the semantics and syntax of <<Ref>> instances used in the substructure definition.

Figure 10 shows our example extended object diagram for meta-model transformation. By changing this object diagram, programmers can modify the weaving process without the need to modify the weaver code directly. The object diagram on Figure 10 describes the following meta-model translation specification.

```
For each Aspect a in system,
    If a.active is true
        For each Class c in a
            If c.name does not exist then generate class
              named c.name and set it as target;
            Else set target as the existing class whose
              name equals c.name;
            End if
            For each Method m in c
                If m.name does not exist then add m to c
                as a member
                End if
            End for
            For each Attribute att in c
                If a.name does not exist then add att to
                c as a member
                End if
            End for
        End for
    End if
End for
```

<<ForEach>> stereotyped packages are used for repetitive element creation. In Figure 10, the innermost <<ForEach>> package named BaseMethod creates a method element for each method m contained in class c that is also contained in aspect a. Cond tagged values are used for conditional repetition or conditional creation of instances. As an example for conditional creation, a baseClass:Class instance is created only if an instance of the same class name does not exist.

A <<Ref>> instance represents a variable instance that imports other instances according to the parameter setting from its tagged values. A <<Ref>> instance is not created even if it is within a <<ForEach>> package. In Figure 10, the target:Class <<Ref>> instance imports a created class within the BaseClasses

<<ForEach>> package or an existing class depending on its tagged value.

5. Conclusion

We have presented a new graphical aspect programming framework for the design and implementation of highly customizable RTOSes. Specifically, we have proposed (1) a graphical aspect programming framework that visualizes aspects, crosscutting classes, and method structures, (2) a new aspect model that supports a sub-method level granularity where an aspect is defined as a set of classes, and (3) an aspect weaving process specified by an object-oriented meta-model transformation.

One of the benefits of the proposed programming framework is that it can clearly show the relationships between aspects, which is extremely difficult in the current text-based aspect languages. Also, our framework achieves granularity finer than at the level of a method; whereas other aspect languages achieve only method-level granularity. This makes our framework suitable for developing and implementing an RTOS, where many aspects are intertwined complexly.

Our aspect model regards the operating system as a group of configurable aspects, and the aspects may contain multiple classes. Unlike the existing object-oriented model, our aspect model allows multiple classes that have the same name to exist in an operating system, and these classes are merged to become a complete class. A method in a class can be divided into multiple basic blocks, which may be included in different aspects, thus achieving granularity finer than the level of a method.

We also specified an aspect weaving process that is easily modifiable and independent of target language. Since existing aspect weavers are hard-coded, they are very difficult to modify and are tied to the target language. To solve this problem, we specified the aspect weaving process using object-oriented model transformation, from which the aspect weaver is automatically generated.

We are currently implementing the graphical aspect programming framework, based on the design presented in this paper. We are also researching an automatic method that can extract crosscutting features into aspects from existing operating systems, such as Linux and FreeBSD.

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 D. Milićev, “Automatic Model Transformations Using Extended UML Object Diagrams in Modeling Environments”, In *IEEE Transactions on Software Engineering*, 2002.

6 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.

7 W. M. Ho, F. Pennaneac'h, and N. Plouzeau, “UMLAUT: A framework for weaving UML-based aspect-oriented designs,” In *Proceedings of Technology of object-oriented languages and systems (TOOLS Europe)*, 2000.

8 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.

9 M. Basch and A. Sanchez, “Incorporating Aspects into the UML”, In *Workshop on Aspect-Oriented Modeling with UML*, 2003.

10 Object Management Group, *OMG Unified Modeling Language Specification Version 1.3*, 1999

11 Rational Software, http://www.rational.com

12 C. Clifton, G. T. Leavens, C, Chambers, and T. Millstein, “MultiJava: Modular open classes and symmetric multiple dispatch for java.”. In *Procesdings of OOPSLA*, 2000.

13 T. Millstein and C. Chambers, “Modular Statically Typed Multimethods.”, In *Proceedings of the European Conference on Object-Oriented Programming*, 1999.

14 M. Kande, J. Kienzle, and A. Strohmeier, “From AOP to UML: A Bottom-Up Approach”, In *Workshop on Aspect-Oriented Modeling with UML*, 2002.

15 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.*