Building a Customizable Embedded Operating System with Fine-Grained Joinpoints Using the AOX Programming Environment

ABSTRACT
Aspect-oriented programming (AOP) has been successful in modularizing crosscutting concerns in complex software systems. In this paper, we present our aspect-oriented approach to building a highly customizable embedded operating system. This is a challenging task since embedded operating systems consist of intertwined concerns often implemented using a mixture of multiple programming languages including an assembly language. Furthermore, they often contain hand-optimized code that makes clear modularization extremely difficult. We provide a two-step approach that addresses these difficulties. First, we devised an aspect-oriented programming environment AOX (Aspect-Oriented eXtension). It supports both modularization and customization of complex software via a set of aspect-oriented mechanisms. AOX extends existing approaches in the sense that it is entirely programming language independent and provides fine-grained joinpoints. Second, using AOX, we built a customizable embedded operating system we call the HEART OS. It is highly configurable and very user-friendly. AOX has been implemented and integrated into the Eclipse IDE as a plug-in module. The HEART OS has also been implemented and ported to the XScale and x86 platforms. Our experience with AOX in building the HEART OS was very positive.

Categories and Subject Descriptors
D.2.6 [Software Engineering]: Programming Environments – Graphical environments

General Terms
Management, Design, Languages

Keywords
AOP, Operating Systems, Fine-granularity, Language Independence

1. INTRODUCTION
Embedded operating systems are required to adapt to a wide variety of hardware devices and application domains. However, various non-functional constraints inherent in embedded systems such as limited memory space, processing time and energy make one-fits-all operating systems impossible. Therefore, developers need to design embedded operating systems in a modularized and highly customizable fashion so that they can easily scale down their embedded operating system to a customized version that only contains exactly required features.

The existing customizable operating systems often rely on conventional techniques such as macro preprocessing, conditional compilation, object-oriented programming or component-based development over micro-kernels. However, it is not always possible to achieve full customizability with these approaches since they fail to modularize crosscutting concerns that are inherent in most operating systems. Examples of such crosscutting concerns include various types of policies and path-specific optimizations such as scheduling, memory management, synchronization and prefetching. It is thus unavoidable that code for such concerns is scattered across the boundaries of source files, classes and components.

The most viable solution to this problem is the adoption of the aspect-oriented programming (AOP) technology [1]. In AOP, a crosscutting concern is modularized via a separate module called an aspect. It is now an 11-year-old technology and has already been widely used in various stages of software development from requirement engineering to implementation.

In this paper, we present an aspect-oriented approach to building a highly customizable embedded operating system. Our contributions are two-fold. First, to render our approach reusable for a wide range of embedded software, we provide an aspect-oriented programming environment we call AOX (Aspect-Oriented eXtension). Instead of using existing AOP languages such as AspectJ [2] and AspectC++ [3], we have devised our own AOP language. In the language, a system is modularized as a collection of features each of which implements a specific concern in the system. The feature is a language construct that groups artifacts for a specific concern that is scattered across the boundaries of directories. The artifact is a language construct that groups codes and texts for a specific concern that is scattered across the boundaries of files. Our language is designed to be independent of a base programming language and to have a fine-grained control over the weaving mechanism. Currently, AOX has been implemented and integrated into the Eclipse IDE as a plug-in. Second, using AOX, we implement a customizable embedded operating system we call the HEART OS. From the design stage to the implementation stage, it has been built with AOP in mind. It can be configured from being, among other things, a very tiny monitor program, that only supports interrupt management, to a medium-sized, embedded operating system with multi-threading, synchronization and message queues.

The rest of this paper is organized as follows. In Section 2, we enumerate design requirements for our aspect-oriented programming environment. In Sections 3 and 4, we present AOX and the HEART OS, respectively. Section 5 summarizes related work. Finally, we conclude this paper in Section 6.
2. DESIGN REQUIREMENTS AND OUR SOLUTION APPROACHES
Before delving into the details of AOX, we first enumerate its design requirements for supporting customizable embedded software development. Then, we briefly explain our solution approaches to satisfy these requirements.

Language independence: An operating system commonly consists of various types of artifacts written in different programming languages. Core functionalities are usually written in C or C++, while machine-level functionalities such as context switching, initialization and interrupt handling are written in assembly languages. Along with them, makefiles are used to define software build processes and linker scripts are employed to specify the memory layout of the target hardware. In order to modularize a concern that is scattered on such artifacts, we need to make our aspect model independent of languages used.

Fine-grained joinpoints: Due to performance constraints, operating systems often require highly optimized code. As a result, functions and data structures are deeply intertwined beyond module boundaries. For example, a function that creates a new process usually contains code fragments for seemingly unrelated features such as context management, synchronization, signal, file, and memory management. An interrupt handler array may contain elements for different devices. In order to enable such optimization, our aspect model needs to support fine-grained joinpoints.

Advice ordering and replacement: Programmers have to be able to decide the advising order of pieces of advice, if a single pointcut is advised by multiple pieces of advice [4, 5]. This is particularly required since the execution order of code fragments, the placement order of fields in a structure and the order of elements in an array have important meanings in operating systems design. Along with the advice ordering, developers have to be able to replace existing pieces of advice with new ones. This is needed when developers want to design default pieces of advice that can be overridden by other pieces of advice. This effectively reduces the size of an operating system since replaced pieces of advice are removed.

Given these design requirements, our approach proposes the following solutions.

XML annotations: In our AOP language, an artifact is annotated by predefined XML tags. The <advice></advice> tag and <joinpoint/> tag represent a piece of advice and a joinpoint defined in an artifact respectively. As the text that surrounds the tags is not parsed or interpreted, our model can be applied to artifacts written in any programming language. Also, this model achieves fine-grained joinpoints since the tag for a joinpoint can be declared anywhere in an artifact without restriction. We are well aware that our model may be unsafe since it does not prevent programmers from declaring joinpoints in inappropriate places or writing pieces of advice with illegal instructions. To overcome this drawback, our programming environment automatically performs weaving and compilation as a background task and immediately notifies programmers of errors caused by such unsafe joinpoints and pieces of advice.

Timestamp-based advice arrangement: In order to support the advice ordering and replacement, we devise the timestamp-based advice arrangement mechanism, in which ordering and replacement relationships among pieces of advice are represented in a two-level list structure. We store this structure in the most recently modified piece of advice to ensure that the ordering and replacement information is always up to date. To determine the most recently modified one, we associate each piece of advice with a timestamp value that is updated when its ordering or replacement relationship is changed.

3. AOX PROGRAMMING ENVIRONMENT
Our solution approaches in the previous section are incorporated in AOX. It is a graphical aspect-oriented programming environment that helps programmers modularize and customize software systems. In this section, we give an overview of AOX by presenting the model of its AOP language. We also explain its weaving mechanism in detail. We then briefly show the structure of its feature repository and XML files described by the aspect model. Finally, we present the implementation of the GUI and the weaver of AOX.

3.1 Model
We depict the model of our AOP language in Figure 1 by using the UML class structure diagram. Note that some classes, attributes and relationships are omitted in the figure for the sake of simplicity. At the top level, there is a repository that represents a dedicated software system. It consists of modules that we call features, interfaces and configurations. We explain these three elements in following sections.

3.1.1 Features
A feature is an entity that encapsulates a specific concern of the system. It is implemented by one or more elements called artifacts. The artifacts are organized in a tree structure via an element called a group. When features are woven, a group and an artifact are mapped to a file and a directory in a file system, respectively.

A feature in AOX can modularize two types of crosscutting concerns. First, it can modularize concerns that crosscut the directory boundaries. This is done by creating a group that has the same path as the group to be crosscut. When weaving features, the groups with the same path are merged into a single group. This idea of merging the tree structure is inspired by [6].

Second, a feature can modularize concerns that crosscut the file
boundaries. This is done by the well known advice-pointcut-joinpoint mechanism used in the conventional AOP languages. In our model, an artifact consists of a text, joinpoints and pieces of advice. Here, a joinpoint represents a specific location (offset) in the text and a piece of advice represents a text fragment that can be copied to the joinpoints. A pointcut is defined as a named group of joinpoints. These three elements are associated as shown in Figure 2. In the figure, six joinpoints scattered in three artifacts and two features are captured by three pointcuts. Among them, one pointcut is advised by two pieces of advice defined in the other feature.

Since programmers can associate multiple joinpoints from different features to the same pointcut, a pointcut can be used to capture joinpoints across features. If we advise a piece of advice to the pointcut, the piece of advice is replicated to multiple joinpoints. Thus, our model supports the quantification [7].

On the other hand, our aspect model sacrifices obliviousness [7] in some degree since programmers are required to declare joinpoints explicitly. Nevertheless, obliviousness still remains in the sense that a feature is obliviousness to which joinpoints it will advise, because the advising is indirectly done via pointcuts. From that perspective, our aspect model enforces strong encapsulation since only the exposed joinpoints can be used by other features.

3.1.2 Functional and Crosscutting Interfaces
Features interact with each other only via interfaces. In our model, the interfaces are categorized into two types: functional and crosscutting interfaces. The former is the ordinary interface that consists of conventional programming constructs such as types, functions, variables, etc. The latter is the interface that consists of pointcuts. It is inspired by the idea of OpenModule [8] and XPI [9].

An interface can be implemented and/or used by features as shown in Figure 1. As the implementation and usage of a functional interface are trivial (e.g., defining functions and calling the functions), we will forego discussing them and proceed to the discussion of crosscutting interfaces. A feature can implement a crosscutting interface by declaring joinpoints and associating them with the pointcuts in the interface. Similarly, the interface can be used by creating pieces of advice that advise some pointcuts in the interface.

By separating interfaces and implementations from each other, programmers can implement each feature concurrently given that the system architect defines all abstract features (features with no implementation) and their interfaces. This clearly shortens development time and reduces integration costs.

3.1.3 Configurations
In addition to features and interfaces, there are configurations in a repository. A configuration represents a customized version of the repository. Since a feature is the unit of customization, a configuration has a list of features that are enabled when the configuration is selected.

In order to prevent illegal configurations, programmers can specify constraints among features and interfaces. For example, an x86 CPU support feature and an ARM CPU support feature cannot be enabled in the same configuration. Another example is that a scheduler feature must be always accompanied by a thread feature.

3.2 Weaving Mechanism
AOX uses text-based static weaving; the content (text fragment) of a piece of advice is copied into the locations where the associated joinpoints are declared. In this section, we explain the two key mechanisms that control the weaving: the timestamp-based advice arrangement and the argument passing mechanisms.

3.2.1 Timestamp-based Advice Arrangement
The order and replacement information is represented in a two-level list structure that we call order info. It is an ordered list of advice groups each of which is comprised of ordered lists of pieces of advice. Here, the first-level list determines the order of advice, while the second-level list determines the priority that controls the replacement of pieces of advice. In a second-level list, the last piece of advice in the list gets the highest priority and it replaces the remaining pieces of advice. In Figure 3. (a), there are seven pieces of advice that advise the same pointcut and their arrangement is a2â†’a3â†’a6â†’a7. Pieces of advice a1, a4 and a5 are replaced, so they cannot participate in weaving. Using the two-level list structure, we can express any ordering and replacement information.

However, the problem of where to store the order info remains. When a piece of advice changes its location in the arrangement, new order info is created that reflects the new arrangement. In order to determine the most recent order info, we choose to store the new order info in the piece of advice that has been moved the most recently. To that end, each piece of advice has a timestamp field and it is updated when the piece of advice is changed.

This process is also shown in Figure 4. (b). Here, a2 is moved between a3 and a6. This new arrangement is described by new order info a2. The timestamp value of a2 is set to 9 which is the highest timestamp value among the pieces of advice. Finally, a2 is stored in a2. When weaving, the weaver search for the piece of advice that has the highest timestamp value and finds that it is a2. Then the weaver arranges the pieces of advice by using the order info stored in a2.

The previous order info a1 is still stored in a5 which was the most recent one before the change. This is required in preparation of a case where a2 is deleted by the programmer and, as a result, a2 is also deleted. In that case, a1 is used to restore the previous arrangement.

It is important to note that it is difficult for human programmers to manipulate the order info structure and update the timestamp values that are distributed among multiple pieces of advice. Obviously, in AOX, they are automatically done by the thread of the programming environment. AOX visualizes the
arrangement and allows programmers to easily change it by dragging and dropping an icon that represents a piece of advice. The details of the GUI will be discussed in Section 3.4.

3.2.2 Argument Passing
A piece of advice can advise multiple joinpoints. However, their context may not be the same. Thus, a piece of advice needs to adapt to the context of each joinpoint where it is currently advising. As do most aspect-oriented languages, AOX also supports argument passing for this purpose.

Arguments are passed from a joinpoint to a piece of advice via a pointcut. To specify what kinds of arguments are supported, a pointcut has an argument list, which is a list of argument names, e.g., declaration, number, name. The type of argument is now fixed to a string. Then a joinpoint can supply values to the arguments. For example, they may be “int x;”, “30”, “foo”. A piece of advice that advises the pointcut can use the arguments in its content. Specifically, when weaving, if $declaration$ and $name$ are encountered in the content of a piece of advice, then they are replaced with “int x;” and “foo”, respectively. This way, a piece of advice can adapt to the context of each joinpoint it is currently advising.

3.3 Structure of Repository and XML Files
The features, artifacts, interfaces, configurations, etc. are all represented in XML files stored in a hierarchy of directories in a file system for a repository. In this section, we briefly show the structure of the directories, and then explain the format of the XML file for an artifact. We do not explain XML files of other types such as features, interfaces, etc. since they are trivial.

3.3.1 Structure of Feature Repository
The aspect model of AOX is realized in a file system subdirectory. A repository is represented as a top-level directory and an XML file called repository.aox. The file contains the version and other meta-information of the repository. All interfaces and configurations are stored in the subdirectory named interfaces and configurations. Both are described as XML files and they can be organized hierarchically using directories. As a result, the path name from the root uniquely identifies them. All features are stored in the subdirectory features. Unlike an interface or a configuration, a feature is represented as a sub-directory that contains an XML file named feature.aox. The file records the links to the interfaces that are used or implemented by the feature. Dependency information is also stored in the file. In a directory for a feature, there are XML files for artifacts and sub-directories for groups.

3.3.2 Structure of XML Files
The XML file structure for an artifact is shown in Figure 4. In the figure, dotted lines represent text that is not interpreted by AOX. It may be program code, script, comments, etc. A joinpoint is represented by <joinpoint> tag. The location of the joinpoint is implicitly determined to be the point where the tag appears in the text. Similarly, <advice> tag denotes a piece of advice. It contains a list of <advicegroup> tags to represent the ordering and replacement information. Next to the tag, comes the content of the piece of advice. In the content, a substring surrounded by a pair of $ characters, is replaced by the argument values given by joinpoints.

3.4 Graphical User Interface and Weaver
AOX is implemented as a plug-in module of the Eclipse IDE. We have two goals in designing and implementing AOX. First, programmers need not to be familiar with the complexities of XML files described in the previous section. Second, AOX should not prevent programmers from using existing plug-in modules. We will give details about in the following two sections. Due to

![Figure 3](image-url)  
Figure 3. The timestamp-based advice arrangement mechanism. (a) The original arrangement. (b) $a_2$ is moved to the middle of $a_3$ and $a_6$. The dotted lines and bold lines represent removed and added part respectively.
space limitations, we do not explain all aspects of the AOX module.

3.4.1 Graphical User Interface
Figure 5 shows the GUI of the AOX programming environment. As shown in the figure, three parts are particularly important. They are the AOX navigator, the text editor and the crosscutting viewer. The AOX navigator visualizes the structure of repositories. Programmers can navigate through features, artifacts, interfaces, configurations, pointcuts and so on. We also provide dedicated form-based editors for creating and modifying the above-mentioned elements except the artifacts. For the artifacts, programmers can use any text editors that are provided by Eclipse or by other plug-in modules. In the figure, the C/C++ editor is used since the artifact contains C language code. AOX does not provide a special editor for artifacts. Instead, it augments the current text editor with graphical annotations and icons that represent joinpoints and pieces of advice.

Figure 6 shows the annotations in more detail. Here, a vertical I-bar denotes a location where a joinpoint or a piece of advice is declared. They are distinguished via different colors. They can easily be added by dragging and dropping a pointcut from the AOX navigator to the specific location in the text editor. After that, a dialog is opened and the programmer is asked to choose whether a joinpoint or a piece of advice will be added in the location.
The details of joinpoints and pieces of advice are visualized and can be edited in the crosscutting viewer as shown in Figure 7. It shows a list of joinpoints and pieces of advice that exist in the artifact that is currently being edited. In the Arguments/Content column, programmers can set argument values for a joinpoint or text content for a piece of advice.

The most important feature of the viewer is the ability to show and manipulate the arrangement of the pieces of advice. When a joinpoint or a piece of advice is expanded by the + button, all pieces of advice that advise the same pointcut are shown even if some pieces of advice are come from other features. The pieces of advice are listed in the order that they will be woven together. In this way, programmers can see the result of weaving before the actual weaving is performed.

The arrangement of pieces of advice can be changed by drag and drop. A piece of advice can be dropped between two adjacent pieces of advice or it may be dropped on top of another piece of advice to replace it. The replaced one is then shown as shadowed. Note that only the pieces of advice that are defined in the artifact that is currently being edited can be rearranged. This ensures that programmers cannot modify pieces of advice from other artifacts by mistake.

3.4.2 Weaver
An artifact in a repository is woven into a file in an eclipse project. Weaving is activated by the build command that is invoked from the GUI. The command may be set to automatic so that the weaving is performed in the background whenever the repository is modified.

Since there are many configurations in a repository, programmers need to indicate to the weaver which configuration should be used during the weaving. The indicated configuration is called the active configuration. This can be set in the AOX navigator.

As previously mentioned, our second goal is to ensure that AOX does not prevent programmers from using the existing plug-in modules. To do so, we make use of the document model provided by the Eclipse framework and devise our own artifact model. The document model is an in-memory buffer that abstracts the text of a file. This is used by all plug-in modules that manipulate text-based elements. For example, text editors are the GUI wrappers of the document model. The artifact model abstracts an artifact. Specifically, it parses the XML file for an artifact and separates text, joinpoints and pieces of advice.

Since other plug-in modules cannot recognize the artifacts in the repositories, only the document model for woven files can be used by the modules. In order to cope with this problem, we parse the XML of an artifact into the artifact model. Among the parsed elements, we give the text part to the document model associated...
with the woven file. Joinpoints and pieces of advice from the artifact model are converted to graphical annotations and applied to the text editor.

In this way, programmers can use project specific text editors, navigators, debuggers and useful functionalities, such as code assist and auto completion and the AOX module simultaneously. This allows AOX to be applicable to any existing plug-in modules and possibly to new plug-in modules that have not yet been developed.

4. HEART OS

Using AOX, we have built a customizable embedded operating system that we call the HEART OS (Highly Expandable Aspect-oriented Real-Time Operating System). In this section, we first give an overview of the HEART OS. Then, we choose three representative mechanisms of the operating system and explain them to show that crosscutting features in these mechanisms have been successfully modularized by using AOX.

4.1 Overview of HEART OS

Like most embedded operating systems, the HEART OS is designed as a library kernel and consists of heterogeneous files that are written in C, assembler, makefile and linker script.

The HEART OS consists of 15 features that interact with each other via 20 functional and crosscutting interfaces. When all features are enabled, the HEART OS provides functionalities similar to that of Nucleus or the uC/OS II real-time kernel. When only the minimal set of features is enabled, it becomes a very tiny operating system that only supports interrupt management. Currently, it is ported on the XScale and x86 platforms and there exist a feature for each of the platforms. Due to space limitations, we do not enumerate the list of features and interfaces.

Obviously, new features and interfaces can be added to the repository of the HEART OS as needed. Because features depend only on interface specifications, this can be done without knowing the implementation details of existing features. This makes the HEART OS highly expandable.

4.2 Representative Mechanisms

In this section, we explain implementation details of three representative mechanisms of the HEART OS: scheduling, interrupt handling and building.

4.2.1 Scheduling

The scheduling mechanism is implemented as shown in Figure 8. Feature arm, which implements the behaviors specific to the ARM architecture, exposes pointcut when_reset at the reset handler function. Feature scheduler advises the pointcut to invoke init_schedule() during the reset.

The scheduling mechanism requires additional fields in thread control block (struct thread), e.g. priority and state. This is achieved by advising pointcut new_field. It is available through feature thread at the end of the declaration of struct thread.

Those fields must be initialized when a thread is being created. Therefore, thread makes pointcut when_created available at the appropriate point in thread_create() function. By advising the pointcut, scheduler initializes the priority and state and inserts the thread into the ready queue. Note that the pointer to the thread structure is passed from thread to scheduler via the argument passing mechanism. Lastly, scheduler implements several scheduling functions as APIs, e.g. schedule() and set_priority().

4.2.2 Interrupt Handling

The interrupt handling mechanism is implemented as shown in Figure 9. First, arm makes pointcut when_interrupt available at the point where hardware interrupt is first handled. The pointcut is advised by feature base so that global_irq_handler() function is invoked at every interrupt. Feature base is the default feature that is enabled in all configurations. It implements a generic OS architecture for the interrupt handler dispatching and provides a generic build script for the OS.

The function gets an IRQ number for the current interrupt and dispatches a handler from array handlers[ ] using the IRQ number as an index. In order for other features to add their own specific handlers to the array, the initialization part of the array is available through pointcut handler defined in interface interrupt.

This pointcut is first advised by feature pxa255, which knows the
Among the 32 interrupts, we have chosen to program XScale, which is 32.

This means the interrupts are ignored by default. The number of interrupts is 32, which is a fixed number and the name of interrupt sources in XScale platform. However, since the feature may or may not provide handler routines for the interrupts, pxa255 initially advises the handler pointcut with the pieces of advice that simply contain "0" (NULL). This means the interrupts are ignored by default. The number of pieces of advice is same as the number of interrupts supported in XScale, which is 32.

Among the 32 interrupts, we have chosen to program pxa255 to provide a handler for the timer interrupt (timer0_handler()). Since it is the most important interrupt. In addition to that, as an example, we have programmed feature user to provide a handler for the serial device interrupt (serial_handler()). These two new interrupt handlers can be registered by advising pointcut handler and replacing the existing pieces of advice.

Let us look at the details of how this interrupt handler registration is done using AOX. It is shown in Figure 10. In the figure, we see that the contents of the existing default pieces of advice are set to "0 /* <description of the interrupt> */". Programmers can easily find a piece of advice that corresponds to a specific interrupt that they want to install a handler for. Once a corresponding piece of advice is found, it is simply replaced by dragging and dropping a new piece of advice over the existing piece of advice as explained in the previous section. In Figure 10, STUART Transmit/Receive/Error and OS Timer match 0 interrupts are replaced by two pieces of advice from user and pxa255 features, respectively.

### 4.2.3 Building

Building an operating system usually incorporates crosscutting concerns in a build script such as the makefile. Figure 11 shows how the building mechanism is implemented in our OS. The core functions of the building mechanism are provided by the makefile in base. However, it cannot determine the name of the compiler and the linker script that are used during the building process. The list of source files to be compiled and the list of header files are also unknown to base. Such information can only be provided by other features such as arm, pxa255, thread and so on.

Feature base exposes four joinpoints: compiler_prefix, linker_script, source_files and header_files. The meanings of the pointcuts are self-evident. These pointcuts are advised by the other features. For example, arm advises compiler_prefix with piece of advice "arm-linux-", which is the prefix of the GCC cross compiler for ARM CPUs.

As another example, pointcut header_files is available at the location where the name of an extra header file is expected, i.e. between the two brackets in macro command #include "header.h". Then, if programmers want to include a header file in the top level header file (heart.h), they can advise the pointcut with the name of the header file as shown in Figure 11.

### 5. RELATED WORK

There have been many language extensions and mechanisms to support fine-grained joinpoints on existing AOP languages. Those include the statement annotation [10], the test-based joinpoints [11] and the loop continuations [12]. Among them, the statement annotation is close to ours; a joinpoint can be declared from our approach. Using the languages, it is possible to declare fine-grained joinpoints in existing AOP languages. There have been many language extensions and mechanisms to support fine-grained joinpoints over existing AOP languages. Those include the statement annotation [10], the test-based joinpoints [11] and the loop continuations [12]. Among them, the statement annotation is close to ours; a joinpoint can be declared from our approach. Using the languages, it is possible to declare fine-grained joinpoints in existing AOP languages.

Weave.Net [13] shares the same goal of the language independence with AOX. It uses XML tags to specify AOP constructs and relies on the common language infrastructure (CLI) of the .NET framework. However, its applicability is limited to the .NET languages, such as C# and lacks tool supports. XVCL [14] is another language independent technique comparable to AOX; the base programming language is considered as a simple text and the text is augmented using XML tags. Specifically, in XVCL, <break> and <insert> tags serve as a joinpoint and a piece of advice, respectively. However, it does not support the ordering of the <insert> tags. It also lacks the ability to modularize multiple concerns that are crosscutting each other, which is possible in AOX.

AspectJ and AspectC++ support aspect ordering in a different way from our approach. Using the languages, it is possible to declare partial orderings among aspects that will be collected to form a complete ordering. Nagy, et al. [4] and Tanter [5] proposed mechanisms that allow programmers to specify conditional and dynamic ordering among aspects. However, the existing approaches only support the ordering at the level of aspects and cannot control the order among individual pieces of advice.
There have been several GUI-based AOP tools that help programmers modularizing crosscutting concerns. In Stellation [15] and FEAT [16], this is done by explicitly associating a source code fragment with a concern and maintaining such associations in a database. They are similar to AOX in that programmers are not required to use the complicated pointcut designator of the conventional AOP languages. However, we have found that these approaches have a weak point as well. They do not work for an artifact that is augmented by other meta-language, such as a C source file decorated with the macro definitions. Also, they do not support selective enabling and disabling of concerns, and thus are difficult to be used for developing customizable software systems.

Our idea of making joinpoints as members of feature’s interface is inspired by OpenModule [8] and XPI [9]. These enforce strong encapsulation of a module’s implementation and prevent the well-known fragile pointcut problem [17]. The idea of weaving directory structures of features is inspired by AHEAD [18], in which crosscutting concerns are weaved by composing corresponding structural elements at the same level.

In line with our HEART OS, there have been several research projects on using AOP technologies in modularizing operating system concerns. Coady, et al. [19] successfully modularized some path-specific optimizations concerns in the FreeBSD kernel using AspectC with low overhead. Lohmann, et al. [20] proposed CIAO that is an embedded operating system having its features modularized using AspectC++. KLASY [21] and TOSKANA [22] are dynamic aspect weavers and they were used to add aspects dynamically to the Linux and the NetBSD kernels, respectively.

6. CONCLUSION AND FUTURE WORK

In this paper, we have presented our aspect-oriented approach to building a customizable embedded operating system in two steps. First, we have devised the AOX programming environment with XML annotations and timestamp-based advice arrangement mechanisms. These mechanisms allow AOX to be independent of the base programming language and to enable fine-grained joinpoints. We have implemented AOX as a plug-in to Eclipse IDE. Although AOX is designed for our customizable operating system, we are sure that it can serve as a general programming environment for most embedded system software.

Using AOX, we have built a customizable embedded operating system we call the HEART OS. We have shown that our aspect-oriented approach is effective in modularizing crosscutting features in the operating system while satisfying the aforementioned requirements. Our experience with AOX in building the HEART OS was very positive.

There are two directions along which our research can be extended. First, we are planning to enhance AOX. Specifically, it will be integrated with the variant management tools such as pure::variants [23] so that AOX can be used for developing software product lines [24]. Second, the HEART OS will be extended to support more features. The enhancements will include SMP support, distributed communication mechanism, dynamic deployment and loading. The results so far look promising.

7. REFERENCES


