

Linguistic Issues for Developing Aspect-Rich High-Performance Adaptable Systems

Shubhanan Bakre and Tzilla Elrad

Department of Computer Science

10 West 31st Street

Chicago, IL 60616

Illinois Institute of Technology

shubh@iit.edu, elrad@iit.edu

ABSTRACT

For a set of applications like operating systems, there is a well-known tradeoff regarding the complexity of the language used to program the system and the resultant performance. One of the consequences of these tradeoffs is that some developers choose to program their system using non-OO languages. AO features have similar tradeoffs yet for systems that require mostly aspectual properties to be adaptable – the tradeoffs balance may shift.

The issues this position paper challenges our workshop are: a) Do we want AO but not OO languages? b) Does it make sense for us to support ‘advanced separation of concerns’ on top of a language that does not support more basic separation of concerns? c) Is the uses of such languages apply only to temporary upgrade of legacy code – or is there intrinsic value for such languages? d) In case we chose to go AO but not OO what are the basic requirements from the underlying language?

We gained a modest insight to these issues by attempting to get the Minix OS go AO.

1. INTRODUCTION

The operating system is a vital component on any computer. It is responsible for managing all the resources and providing services in an efficient manner. When an application software needs modification, the complexity of the task depends upon the size of the application and the amount of code tangling present in the initial implementation. In case of an operating system, a lot of complexity is involved owing to the numerous concerns that are tangled inside the operating system code. So making modifications to the operating system code can be a very difficult – if not impossible – job.

Aspect oriented software development [1] provides techniques that enable us to remove this tangled code – caused by crosscutting concerns – and design and implement them in a modular way.

The objective of our research was to demonstrate that it is possible to support aspect orientation within an operating system that uses procedural as opposed to object-oriented strategies for achieving separation of concerns. In accordance with this aim, we chose the Minix operating system [8] and identified one concern/property of the system that crosscuts the whole operating system code, and modularized it using aspect oriented software

development techniques. For this purpose we chose exception handling within Minix.

One of the important reasons in choosing Minix for the purpose of this research was the presence of a well-defined structure in the operating system design. Advanced separation of concerns is practicable only in software that already has some basic separation of concerns. Minix is based on a client/server architecture. In addition to the basic structure provided by C, it has modular units in the form of various subsystems, which impart some semblance of structure to the operating system. Another reason for choosing Minix was that Minix is developed using C. Most of the present day operating systems are written in C. So working with Minix enables us to extend the scope of this research to other operating systems written in C.

The client/server architecture imparts a modular structure to the Minix operating system. In spite of this well-defined delineation of various services (memory manager, file system, network manager) Minix shows evidence of various crosscutting concerns like synchronization, security, scheduling, and exception handling.

Two kinds of crosscutting concerns are found in a client/server operating system like Minix. The first type is the intra-subsystem crosscutting concerns and the second type is the inter-subsystem crosscutting concerns.

A concern that crosscuts more than one subsystem within the system can be termed as an inter-subsystem crosscutting concern, whereas an intra-subsystem crosscutting concern is local to a particular subsystem. Exception handling is an inter-subsystem crosscutting concern and is scattered throughout the system code.

Our approach involves the development and use of an aspect weaver that takes as input an aspect specification and clean functional code and weaves them to produce the combined code in C. For this purpose we wrote an aspect-oriented extension to C in which the aspect specification can be written. The next section discusses the current implementation and the issues involved with it. The Aspect-Oriented Implementation section discusses our solution to the issues related to the current implementation. A brief discussion and analysis follows in the next two sections. We conclude with a summary of some related work and conclusion.

2. EXCEPTION HANDLING IN MINIX

Exception handling in Minix is implemented using the panic function. The basic task that is performed by this function consists of three steps. The first step is to print a message giving information about the cause of the exception. In the second step, any unsaved data is flushed to the disk. In the final step the system is aborted. Calls to this function are scattered all over the operating system code. In the Minix operating system code there are about 87 calls to the panic function spread out over three different subsystems and 21 files.

An exception can occur under various conditions like failure in synchronization, failure in communication, or failure to obtain free space in main memory. There are several cases under which the implementation of this concern can change. For example, the introduction of another condition under which the panic should be triggered. In this case the system programmer will have to go through the whole operating system code and find out in what places this condition might occur and add the panic function call at those places.

Another case under which the implementation of the panic concern might change is, if some extra information needs to be passed to the function. In this case, every location at which panic is called will have to be modified in order to add the extra argument.

Both these examples demonstrate the shortcomings of the current implementation and highlight the need for advanced separation of concerns. The above-mentioned examples not only introduce invasive changes in the operating system code but also are a major cause of bugs in any kind of software.

The reason behind all these problems is that the code related to exception handling is not implemented in a modular way. This is caused due to the lack of any capability of the current implementation to express the exception handling in a modular way keeping the current decomposition of the system intact. This results in a code that is highly sensitive to change and has a very low degree of reusability and extensibility.

These problems can be solved using aspect-oriented software development techniques. The approach that we used in our implementation uses an aspect language for expressing the exception handling in a modular manner, keeping the current decomposition of the system intact.

3. THE ASPECT-ORIENTED IMPLEMENTATION

The aspect-oriented implementation for this problem makes use of an aspect language that we developed for this purpose. The syntax and semantics of our language are similar to AspectJ [2]. The code related to exception handling is modularized within an aspect specification for exception handling. This specification serves as an input for the weaver, which statically weaves the advice specified in the aspect specification into the functional code of the operating system.

An around advice runs at each joinpoint to decide whether execution should continue or an exception should be raised. The

around advice makes this decision based on a value returned by an after advice that runs at all the points where the system might enter into an undesirable state (i.e., all the points where the original calls to panic were located).

Figure 1 illustrates the aspect specification for one condition and one function for which exception handling is required. In Minix there are several functions and several conditions for which exception handling is needed.

At the beginning of the aspect specification any header files to be included can be specified. The aspect specification is divided into three parts. The first part consists of the private declarations. In this section any variables or functions that are to be used within the advice can be defined. This section is optional. The next section is the pointcut declaration section. In this section, any pointcut specification that will be used along with an advice in the advice section is declared. The pointcut section must have at least one pointcut specification. The next section is the advice section. An advice must specify a pointcut from the pointcut list, so that the weaver knows where the advice is to be woven. For ease of implementation, a primitive pointcut is not allowed in the advice.

In Figure 1, the initial part declares all the include files which are being used. In the first section a few variable definitions are given. A function that does the actual exception handling is also defined. After this the pointcut section starts. In this section two pointcuts are defined. The first one captures all joinpoints where the 'clock_time' function is called. The second pointcut captures all points within the 'clock_time' function where the function 'sendrec' is called. The second pointcut also captures the return value from the function 'sendrec'. This value is available to the advice that runs at all the joinpoints within this pointcut. The next section is the advice section, which has an around advice and an after advice. The around advice is responsible for deciding whether exception handling is needed. The after advice helps in this decision-making.

4. AO BUT NOT OO?

Many design issues arise when it comes to introducing advanced separation of concerns within a system built on top of a procedural paradigm rather than an object-oriented paradigm. The problem occurs due to the difference in the level of abstraction between the two approaches.

Since the OO paradigm has better tools for representing modular units, more types of joinpoints/pointcuts are available as compared to the procedural paradigm.

In particular, problems occur when dealing with legacy code, which is not modular. For example, in the decision making involved during exception handling, a lot of conditions are evaluated some of which might consist of global or local variables. Capturing such joinpoints requires additional language features in order to prevent vestigial code.

4.1 C+: An Enhanced Non-OO Language

One possible solution might be to add capability to the language to enable reference to these points. An ideal case would be a language that maintains the performance edge of C and at the

same time adds features to address all joinpoints involving references to variables. Whether this is an acceptable solution is an open issue.

5. ANALYSIS

Our experience confirms the fact that it is not only possible to have aspect-orientation within a operating system developed using structured programming but also advantageous. Our implementation of exception handling in Minix is more adaptable and extensible than the current implementation.

5.1 Comprehensibility

Our implementation of the exception handling is more comprehensible than the current implementation. Since the code related to exception handling is localized, it is much easier to understand.

5.2 Adaptability

Since the crosscutting concerns related to exception handling are resolved and modularized it becomes easier to adapt the implementation to different requirements. Any unforeseen changes to the exception handling aspect of the system can be made without introducing invasive changes in the operating system code because the code related to exception handling is modularized.

5.3 Pluggability

The aspect-oriented implementation also supports (un)pluggability. A totally different implementation of exception handling can be added in place of the current implementation without making any invasive changes in the operating system code.

6. ISSUES

The purpose of this research was to have a proof of concept about having aspect-orientation in a traditional operating system. So the implementation of the aspect language was limited to the features that were necessary for this purpose. So some of the features are not yet supported in this implementation like passing function parameters to the advice. It would be a good idea to add more features to the existing implementation.

Another issue is the performance of the operating system. Performance is a very vital issue when it comes to operating system software. Compared to the current implementation of the exception handling aspect, the aspect-oriented implementation does not add any significant run time overheads to the system. This is because the weaving is done at compile time. Hence it can be safely said that the aspect-oriented implementation is at par with the current implementation in terms of performance.

Another issue with the aspect-oriented implementation is that of methods of implementation. In the current implementation of the operating system, size of the modules is very big. So for enabling execution of the advices at the proper locations, such modules have to be broken down into smaller parts. In other words some amount of code refactoring is needed. Alternatively, placeholder functions can be used in order to provide hooks (or joinpoints) for

the advices. In our implementation the latter approach was preferred because of the ease of implementation and the low chances of error. Another reason for choosing placeholder functions over code refactoring was to preserve the structure of the modules (dominant decomposition) based on their original functionality (and in order not to break the modules with respect to exception handling concern). It is an open issue as to which method is the better one.

As we mentioned in section 4, a problem with non-object-oriented languages is the difficulty in capturing joinpoints involving references to global or local variables. It would be disastrous to have joinpoints over each statement in the program otherwise the system would become too complicated to comprehend. Our solution to the problem was to introduce dummy functions that act as placeholders at these locations. Calls to these dummy functions serve as the joinpoints in order to reference the variable values at these locations.

This solution does not solve the complete problem of crosscutting concerns. Even though the code related to exception handling is modularized within the aspect, some vestigial code remains behind at the joinpoint locations. This code does not affect the system with regards to the functionality but might affect the comprehensibility of the system code.

This issue is related to the lack of expressibility of the language and as such must be addressed at that level.

Semantics of the aspect language must be well-defined in order to predict system behavior. Especially for system software it is essential to have language semantics that does not allow the programmer to make any mistakes thereby making the system unstable. A case in point is the 'proceed' function that is used for passing control to the function at the joinpoint. AspectJ semantics about 'proceed' specifies that calls to 'proceed' can be made any number of times and that it can execute any number of times within the 'around' advice. This gives a lot of control to the system programmer and can be useful in many situations. But it can also cause disaster if not used wisely. For example, if the 'proceed' refers to a synchronization function. In this case it would be a disaster if the function gets called more than once. Our approach to this issue was to allow only one execution of 'proceed' within the control flow of the advice. In this manner, the function at the joinpoint can execute at most once.

7. RELATED WORK

Path specific customizations [3] also addresses the issues related to crosscutting within a layered operating system that occurs due to path specific customizations. Their approach is a similar one of using an aspect weaver to introduce aspect-orientation within the operating system.

The Aspect Moderator Framework [6] also touches the issues related to a layered operating system. The framework makes use of the object-oriented technology in order to achieve better separation of concerns.

[5] demonstrates an aspect-oriented implementation of interrupt synchronization within the PURE operating system. It uses an aspect-oriented extension to C++.

8. CONCLUSION

For systems, such as operating systems, where most commonalities are horizontal it is beneficial to use an aspect oriented approach.

For existing operating systems that are written in a non-object-oriented language, an AO extension is doable and beneficial from the point of view of system adaptability. Regarding system performance – tradeoffs can be achieved by compile time weaving.

The current implementation of exception handling in Minix is spread across 21 files in three different subsystems including the kernel. The aspect-oriented implementation modularizes all this crosscutting code related to exception handling and puts it in one place. The implementation of the weaver was done using LEX, YACC and C and is almost 4000 line of code.

This work was done as a part of a master's thesis and we plan to continue working on it in order to experiment with other operating systems and other aspects. In particular we would like to modify the weaver in order to support more features.

The source code is available upon request from shubh@iit.edu.

```

/* Aspect Specification for exception handling in the procedure clock_time(). */

/*Include Section - Any predefined values and functions have to be included*/
#include "fs"
#include unistd
#include stdio
#include errno
aspect panic {
  /* Private Variables and Functions Declaration Section */
  int panicking;
  int error;
  void panic_func(char *format, int num)
  {
    if(panicking) return;
    panicking = TRUE;
    printf("File system panic: %s", format);
    if(num != NO_NUM) printf("%d", num);
    printf("\n");
    (void)do_sync();
    sys_abort(RBT_PANIC);
  }
  /* Pointcut Definition Section */
  pointcut fs_panic():
    calls(time_t clock_time());
  pointcut clk_time(int err) returning(int err):
    calls(int sendrec()) && withinmethod(time_t clock_time());
  /* Advice Definition Section */
  around(): fs_panic()
  {
    proceed;
    if(error == OK) {
      printf("Hello world\n");
    }
    else
    {
      printf("Synchronization panic\n");
      panic_func("clock_time:\n", error);
    }
  }
  after(int err): clk_time(err)
  {
    error = err;
  }
}

```

Figure 1: A Sample Aspect Specification for Exception Handling in Minix.

9. REFERENCES

- [1] Aspect-Oriented Software Development:
<http://www.aosd.net>
- [2] AspectJ: <http://www.aspectj.org>
- [3] Coady, Y., Kiczales, G., Feeley, M., and Smolyn, G.
 2001. Using AspectC to improve the modularity of

path-specific customization in operating system code.
 In *Proceedings of Joint ESEC and FSE-9*, pp88-98.Vienna, Austria.

- [4] Coady, Y., Kiczales, G., Feeley, M., Hutchinson, Ong, J. S. *Communications of the ACM*, vol 44, issue 10. pp 79-82, October 2001.

- [5] Mahrenholz, D., Spinczyk, O., Gal, A., Schroder-Preikschat, W. An Aspect-Oriented Implementation of Interrupt Synchronization in the PURE Operating System Family. *Proceedings of the 5th ECOOP Workshop on Object Orientation and Operating Systems*, np. Malaga, Spain.
- [6] Netinant, P. 2001. Supporting the Design of Adaptable Operating Systems Using Aspect-Oriented Frameworks. PhD. diss., Dept. of Computer Science, Illinois Institute of Technology.
- [7] Netinant, P., Elrad, T., Fayad, M. E., A Layered Approach to Building Open Aspect-Oriented Systems: A Framework for the Design of On-Demand System Demodularization. *Communications of the ACM*, vol 44, issue 10. pp 83-85, October 2001.
- [8] Tanenbaum, A.S. and Woodhull, A., S. eds. 1997 *Operating System Design And Implementation*. Upper Saddle River, NJ-07458: Prentice Hall.