Applying Aspects to a Real-Time Embedded Operating System

Francisco Afonso[1] Carlos Silva[1] Sergio Montenegro[2] Adriano Tavares[1]

[1] Department of Industrial Electronics
University of Minho
Campus de Azurém
4800-058 Guimarães - Portugal

{fafonso, csilva, atavares}@dei.uminho.pt

[2] Fraunhofer - Institute for Computer Architecture and Software Technology (FHG-FIRST)
Kekuléstraße 7
12489 Berlin - Germany

sergio.montenegro@first.fraunhofer.de

ABSTRACT

The application of aspect-oriented programming (AOP) to the embedded operating system domain is still a very controversial topic, as this area demands high performance and small memory footprint. However, recent studies quantifying aspects overheads in AspectC++ show that the resource cost is very low. Therefore, operating system development may benefit with the modularization of crosscutting concerns and system specialization offered by AOP.

This paper addresses our experience in applying aspects to synchronization (mutual exclusion) and logging in a real-time embedded operating system (BOSS). Furthermore, we present our ideas for future investigation in aspect-oriented implementations for fault tolerance, middleware customization and platform variability.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.4.7 [**Operating Systems**]: Organization and Design – real-time systems and embedded systems.

General Terms

Design, Reliability, Languages

Keywords

Aspect oriented programming, modularization, customization.

1. INTRODUCTION

Aspect-Oriented Programming (AOP) has opened a new field of research on the applicability of this software engineering technique to several application domains. Aspect-oriented language extensions, like AspectJ[5] and AspectC++[4] have been making easier the application of this technology to improve the modularization, separation of concerns, customizability and reusability of new and existing projects.

Workshop ACP4IS'07, March 12–13, 2007, Vancouver, British Columbia, Canada.

In this paper we describe our experiments using AspectC++ to implement concerns as mutual exclusion and logging in the BOSS embedded operating system. Besides, we present our future investigation goals in other areas, such as application level fault tolerance, operating system fault tolerance, middleware customization and platform variability.

This paper is organized as follows. Section 2 introduces the BOSS operating system and presents its philosophy and basic building blocks. Section 3 describes and discusses our experiments with mutual exclusion. Section 4 presents and exemplifies the use logging aspects. Section 5 describes our future investigation plans for others domains in operating systems development, as fault tolerance, middleware customization and portability. In Section 6 we make reference to the related work and finally in Section 7 we summarize this paper.

2. BOSS OPERATING SYSTEM

BOSS is a real-time operating system developed by FHG-FIRST, and had as first application the BIRD (Bi-Spectral Infrared Detection) satellite [18], launched in 2001. Since then, it has been applied in several other projects. Future BOSS utilization include CubeSat satellites [19] (TinyBOSS version) and robotics in space.

Simplicity is the main strategy for achieving dependability [6] in BOSS, as complexity is the cause of most development faults. The system was developed in C++ and has been ported to several platforms as PowerPC, x86 and Atmel AVR. It also runs on top of Linux, mainly for developing and testing purposes. Figure 1 exhibits a simplified class diagram of the core functionally in the kernel.

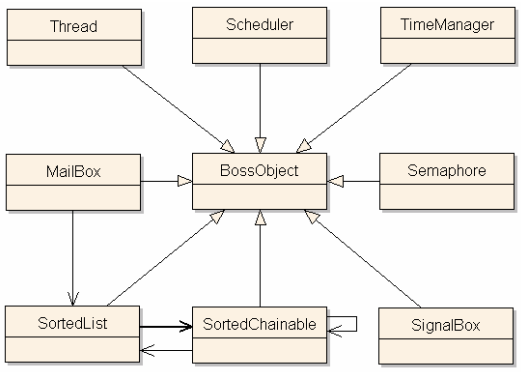

Figure 1. Core BOSS class diagram.

BOSS was designed to support fault tolerance in applications with hardware redundancy by including a middleware which carries out transparent communications between nodes, using the publisher-subscriber protocol.

3. ASPECTS FOR MUTUAL EXCLUSION

Operating systems use several mechanisms to achieve mutual exclusion between processes/threads and interrupt handling routines in executing critical sections of code, like disabling interrupts, disabling dispatch, using spin locks and semaphores. Generally the functions and methods that implement mutual exclusion are spread throughout the operating system code.

In the BOSS operating system, the most used mechanisms for mutual exclusion are disabling interrupts and disabling dispatch. These concepts are applied in building most operating system core functionality as signal boxes, mail boxes and even semaphores.

Our approach was to treat mutual exclusion as a separate concern and implement it using aspects. Figure 2 shows an example of the original implementation of the MailBox send method, and Figure 3 presents the created aspect code for disabling dispatch.

```
#define THREAD_ATOMAR \
                  scheduler.disableDispatch();
#define THREAD_ATOMAREND \
                  scheduler.enableDispatch();

void MailBox::send(SortedChainable* message) {
    THREAD_ATOMAR {
        messages.append(message);
        if(suspendedReceiver != 0)
           suspendedReceiver->resume();
        suspendedReceiver = 0;
    } THREAD_ATOMAREND;
}
```

Figure 2. MaiBox send implementation.

```
aspect ThreadAtomar {

  pointcut critical () =
    execution("% MailBox::send(...)") ||
    execution("% MailBox::receive_in(...)") ||
    execution(
        "% SortedList::getRemoveFirst(...)") ||
    execution ("% SortedList::find(...)") ||
    execution ("% SortedList::remove(...)") ||
    execution ("% SortedList::insert(...)") ||
    execution ("% SortedList::append(...)") ||
    execution ("% Semaphore::enter_in(...)") ||
    execution ("% Semaphore::leave_in(...)") ||
    execution ("% SignalBox::get_in(...)") ;

  advice critical () : before() {
     scheduler.disableDispatch();
  }

  advice critical () : after() {
     scheduler.enableDispatch();
  }

  advice execution("% SignalBox::get_in(...)"):
      order ("ThreadAtomar","InterruptAtomar");
};
```

Figure 3. ThreadAtomar aspect.

The disable dispatch mechanism only prevents concurrent access by threads. As shown in Figure 3, for the method get_in of the SignalBox class, a second aspect is applied to avoid concurrent access between threads and interrupt handling routines. The order advice declaration defines that the ThreadAtomar aspect has precedence over the InterruptAtomar aspect, not shown in this figure.

Besides disabling interrupts and dispatches, BOSS uses semaphores (mutexes) to protect critical code in other kernel classes, as the NameServer and Message classes. In this case, the separation of this semaphore code to an aspect is similar to the presented so far, but it includes a data member introduction of a private semaphore for each class it protects.

The application of the aspect-oriented version for mutual exclusion gives as advantage the separation of this concern from the basic system functionality. It makes easier for the programmer to understand where each mechanism is applied and to modify its application. The bigger the system is, the greater the benefits are. A minor disadvantage is that sometimes a refactoring in the operating system code is needed, usually by creating a new method exposing the critical section.

The performance and memory footprint of the implementations described in this section were evaluated using AspectC++ version 1.0pre3 and gcc version 3.4.2 in a Pentium computer. AOP and non-AOP versions were compiled under various degrees of optimization, and the final executable code was analyzed and compared. For optimization levels –O2 and –O3, the performance is supposed to be the same, as an identical code is generated. Regarding memory consumption, for optimization levels –O2 and –O3 the AOP version presented the same code size and an increase of 4 bytes in the .bss section for each aspect. Based on these results, we conclude that no significant cost is introduced with the AOP implementation.

4. ASPECTS FOR LOGGING

Logging is an essential mechanism for code validation and debugging, both at the operating system and at the application level. In its simplest form, it may consist of applying printfs in selected points of the code, guarded by ifdefs or if clauses for selective activation. More elaborated logging mechanisms rely on memory or file system storage.

The application of aspect-oriented programming to the logging domain seems to be both intuitive and practical. Several logging aspects can be created, covering distinct objectives, and they can be composed at compile time accordingly. The clear advantage is separating the logging code from the operating system code, and so each developer may own a proper set of logging aspects that do not interfere with other developer's code and that can be easily adapted to a new operating system release.

As an example, a logging aspect for tracing thread activities is shown in Figures 4 and 5. The LogManager class defines procedures for writing data to an internal memory buffer in ASCII format, using similar input format as printf. The printIt method sends the collected data to the display or to a serial interface.

The ThreadLogging aspect inherits from the LogManager class. As each aspect in AspectC++ will be implemented by a singleton object, only one instance of the memory buffer will be used by all threads. This aspect uses function execution joinpoints and so it

will be weaved into the operating system code. Usually the OS code is compiled into a library and this library is linked to the application code. In this example, the only modification required in the application code is to call the printIt method of the ThreadLogging aspect at the end of the logging session, by executing: ThreadLogging::aspect_of()->printIt(). As an option, the application code can also use the logging functionality calling the write or writeTimeEvent functions.

```
class LogManager {
  static const int BUFFERSIZE = 10000;
  char logBuffer[BUFFERSIZE];
  char * bufferPtr;
public:
  LogManager() { init(); }
  void init();
  void write(const char * fmt, ...);
  void writeTimeEvent(const char * fmt, ...);
  void printIt(void);
};
```

Figure 4. LogManager class.

```
aspect ThreadLogging: public LogManager {

  pointcut log1 () =
   execution ("% Thread::restart(...)") ||
   execution ("% Thread::exit(...)") ||
   execution ("% Thread::resume(...)");

  advice log1 () : before()  {
     writeTimeEvent("before %s target=%s",
                    JoinPoint::signature(),
                    tjp->target()->myName);
  }

  pointcut log2 (Time tm) = args(tm) &&
    execution ("% Thread::suspendUntil(...)");

  advice log2 (tm) : before(Time tm)  {
    writeTimeEvent("before %s until=%ld
                   target=%s",
                   JoinPoint::signature(), tm,
                   tjp->target()->myName);
   }
};
```

Figure 5. ThreadLogging aspect.

The utilization of aspects for logging can take advantage of the big variety of pointcut functions provided by AspectC++, specially the cflow, within and target functions. By using aspects, selective logging is easy to implement, allowing activation only in specific contexts and scopes.

A limitation in aspect-oriented logging is the inability to place logs in specific points inside a method, or to log internal variables. To overcome these limitations some refactoring may be needed, as the one discussed in Section 3.

The logging aspects were not submitted to performance comparisons to non-AOP versions, because both types of logging will influence the performance and memory footprint of the operating system, and are not supposed to remain in the final version.

5. FUTURE INVESTIGATION

Besides the application of aspects in the separation of concerns in basic operating system functionality, as presented for mutual exclusion and logging, we can envision its use for a broader specialization, namely in the domain of fault tolerance, middleware customization and platform variability.

The following sections discuss our proposal for future investigation of aspect application in these domains, using the BOSS operating system as test bed.

5.1 Application Fault Tolerance

The main objective of our research is providing support for application level fault tolerance by the operating system and middleware. Fault tolerance is usually achieved by redundancy and diversity. Hardware redundancy and software diversity are the most common techniques for increasing system reliability, but several other techniques may be applied, as time redundancy (task re-execution), information redundancy (correction codes) and data diversity (data re-expression).

Several FT strategies have been proposed and applied in the last 30 years. Some are effective only against transient faults, like hardware transient faults caused by electromagnetic radiation; others can deal with permanent software faults like Recovery Blocks (RB) [20], Distributed Recovery Blocks (DRB) [13] and N-Version Programming (NVP) [7].

Our approach is to provide a framework at the operating system/middleware level that makes possible the implementation of a wide variety of fault tolerant strategies, at the application level, with maximum transparency. For this purpose, a FT framework was developed and integrated to the BOSS operating system [1][2]. Figure 6 contains a simplified class diagram showing its basic classes.

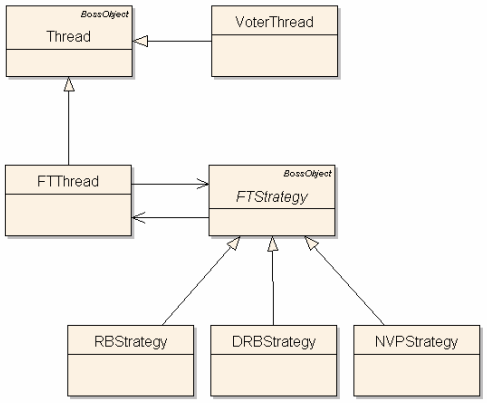

Figure 6. Fault tolerant framework class diagram.

A fault tolerant thread must inherit from FTThread and define an FTStrategy object which will implement the fault tolerant functionality. Some strategies, like DRB, involve message exchanges and coordination between multiples nodes, for defining roles, initializing global state and communicating results. All this work is performed by the middleware, but some specific procedures must be defined by the application, as for instance, the acceptance test in RB and DRB. The degree of transparency depends on the strategy selection and configuration. The VoterThread class is used in NVP to select the correct response among the NVP threads.

Our goal is to develop an aspect-oriented version of the framework and compare it with the current implementation, with regard to modularization, maintainability, performance and memory footprint.

5.2 Operating System Fault Tolerance

Another objective of our research is to provide fault tolerance to the operating system kernel itself. As already pointed out in the previous section, fault tolerance always requires some form of redundancy. This usually reflects in resource costs, as memory size or run-time overhead. Therefore, the application of fault tolerance to the operating system is normally avoided, as performance and memory footprint are of vital importance to the embedded system. However, for systems demanding high levels of dependability, as space or safe-critical applications, the implementation of mechanisms for fault tolerance in the operating system can be of great importance.

Our approach to provide fault tolerance to the operating system is the application of fault containment wrappers, as described in [21], where reflection techniques were used. As this study suggests, predicates, or invariants, can be established for each functional class in the operating system.

As a simple example, Figure 7 shows an aspect for checking the semaphore correct operation. For each execution of the semaphore primitives enter or leave a separate counter is incremented. With this additional information and the original value of the counter attribute of the Semaphore class, which represents the number of resources available, the advice code can verify if there is a discrepancy in the actual value of the counter variable.

```
aspect SemaphoreErrorDetection{

  advice "Semaphore": int initialCounter;
  advice "Semaphore": int enterCounter;
  advice "Semaphore": int leaveCounter;

  advice construction("Semaphore"): after(){
    tjp->target()->initialCounter =
                        tjp->target()->counter;
    tjp->target()->enterCounter = 0;
    tjp->target()->leaveCounter = 0;
  }

  advice execution("% Semaphore::enter(...)") :
            before()  {
     int calculatedCounter =
               tjp->target()->initialCounter
             - tjp->target()->enterCounter
             + tjp->target()->leaveCounter;
     if(tjp->target()->counter !=
        calculatedCounter){
       // doesErrorHandling();
     }
     tjp->target()->enterCounter += 1;
  }

  advice execution("% Semaphore::leave(...)") :
            before()  {
     // same as above for the enter primitive
     // ...
     tjp->target()->leaveCounter += 1;
  }
};
```

Figure 7. Aspect for semaphore error detection.

5.3 Middleware Customization

The BOSS operating system includes a middleware layer to allow local and external message communication transparently, using a publish-subscriber protocol. The main classes involved with this functionality are presented in Figure 8.

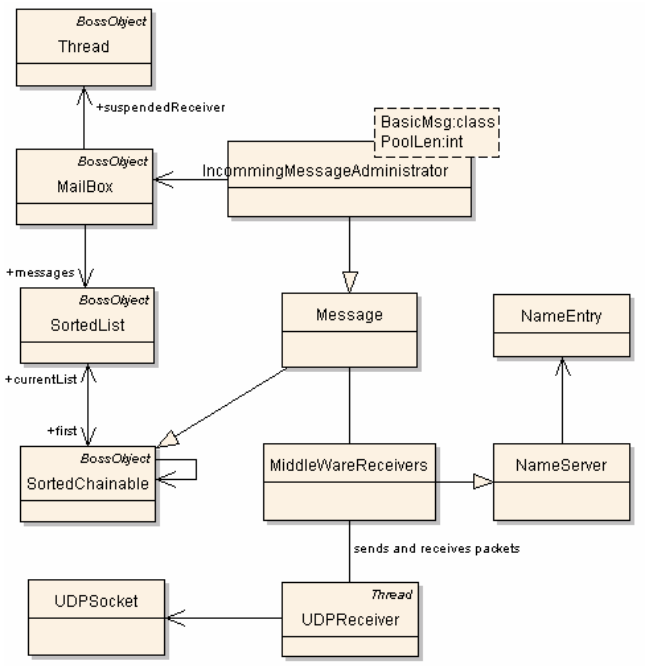

Figure 8. Middleware class diagram.

The Message class is the basic means of communication. It can be used directly or through other support classes like the Incomming-MessageAdministrator class. The MiddleWareReceiver class implements the basic procedures for sending and receiving messages based on its NameServer functionalities, registering destination messages and its respective subjects. The real communication is implemented by a support thread, represented in Figure 8 by the UDPReceiver thread.

We intend to explore aspect-oriented solutions to the middleware customization, such as network selection (e.g. Ethernet or, CAN), model selection (e.g. point-to-point or broadcast), marshalling configuration, and even some fault tolerance techniques at the communication level (e.g. message duplication, fault detection and retransmission).

The application of AOP to middleware customization seems to be promising, because of the high degree of configurability involved. In this domain, the aspect composition may be clearer and less prone to errors than common solutions based on ifdefs or hooks.

5.4 Platform Variability

Extending the application of aspect-oriented development in operating systems, we propose the study of AOP techniques for facilitating the portability for different hardware environments.

As mentioned in Section 2, the BOSS operating system was ported to several platforms and we plan to experiment if the use of AOP for weaving hardware dependent functionality would

improve the system maintainability without compromising its performance.

6. RELATED WORK

Aspect-oriented implementations in interrupt synchronization concerns have been well described for PURE [16], CIAO[14] and eCos[15] operating systems. The work on eCos[15] concluded that the application of AspectC++ for interrupt synchronization and kernel instrumentation did not incur in a significant increase in code size (0.9%), and even improved the run-time performance (1%).

The advantages of using AOP for program instrumentation, including debugging, profiling and run-time monitoring, were discussed in [17].

The work on [9] proposed the use of aspect-orientation to distribution, timeliness and dependability domains, giving some examples based on a CORBA application. Aspects were also proposed for improving the performance of existing fault-tolerance systems, like FT-CORBA [22], but very few works used aspects to implement the FT functionality, as described in [3]. In fact some researchers are skeptical about the use of AOP for distributed computing, as concurrency control and failure management [12]. Our future investigation in that domain aims to give some contribution to that debate, but focusing in the field of embedded systems.

Middleware specialization and customization with AOP is being applied with good results in large scale middleware as CORBA [10] [8] and ACE [11].

7. SUMMARY AND FUTURE WORK

This paper describes our work with the implementation of aspect-oriented modules for supporting mutual exclusion and logging to the BOSS operating system. The main advantages and weaknesses of this approach were presented for each case, as for example, the need of refactoring in the base code.

Our future work, already described with some detail in Section 5, include the use of AOP for fault tolerance support at the application level and for the operating system itself, and for middleware and platform specialization.

8. ACKNOWLEDGMENTS

This work has been supported by the Portuguese Foundation for Science and Technology.

9. REFERENCES

[1] Afonso, F., Silva C., Montenegro S. and Tavares A. Middleware Fault Tolerance Support for the BOSS Embedded Operating System. *In Proceedings of the International Workshop on Intelligent Solutions in Embedded Systems- WISES*, Vienna, Austria, 2006.

[2] Afonso, F., Silva C., Montenegro S. and Tavares A. Implementation of Middleware Fault Tolerant Support for Real-Time Embedded Applications, *In Proceedings Work-in-progress Session of the 18th Euromicro Conference on Real-Time Systems- ECRTS*, Dresden, Germany, 2006.

[3] Alexandersson, R., Ohman, P., and Ivarson, M. Aspect Oriented Software Implemented Node Level Fault Tolerance. *In Proceedings of the 9th IASTED International Conference on Software Engineering and Applications -SEA*, Phoenix, Arizona, USA, 2005.

[4] AspectC++ project homepage: http://www.aspectc.org

[5] AspectJ project homepage: http://eclipse.org/aspectj/

[6] Avizienis, A., Laprie, J.-C., and Randell, B. Fundamental Concepts of Dependability. *Technical Report 739*, Department of Computing Science, University of Newcastle upon Tyne, 2001.

[7] Chen, L., and Avizienis, A. N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation. *In Proceedings of FTCS-8*, pp. 3-9, Toulouse, France, 1978.

[8] Colyer A., and Clement A. Large-scale AOSD for Middleware. *In Proceedings of the 3rd International Conference on Aspect-oriented Software Development*, Lancaster, UK, pp. 56-65, 2004.

[9] Gal, A., Spinczyk, O., and. S-Preikschat, W. On Aspect-Orientation in Distributed Real-time Dependable Systems. *In Proceedings of the Seventh International Workshop on Object-Oriented Real-Time Dependable Systems - WORDS*, pp. 261-267, 2002.

[10] Hunleth, F., Cytron R., and Gril C. Building Customizable Middleware Using Aspect Oriented Programming. *In Proceedings of Workshop on Advances in Separation of Concerns on Object-Oriented Systems - OOPSLA*, Tampa Bay, Florida, USA, 2001.

[11] Kaul, D., and Gokhale, A. Middleware Specialization Using Aspect Oriented Programming. *In Proceedings of the 44th Annual Southeast Regional Conference*, Melbourne, Florida, USA, pp. 319-324 2006.

[12] Kienzle J., and Guerraoui, R. AOP: Does it Make Sense? The Case of Concurrency and Failures. *In Proceedings of the 16th European Conference on Object Oriented Programming*, pp. 37-61, 2002.

[13] Kim, K., and Welch, O. Distributed Execution of Recovery Blocks: An Approach for Uniform Treatment of Hardware and Software Faults in Real-Time Applications. *IEEE Transactions on Computers*, vol. 38, Nº 5, pp. 626-636, 1989.

[14] Lohmann, D., Spinczyk, O., and S-Preikschat, W. On the Configuration of Non-functional Properties in Operating System Product Lines. *In Proceedings of 4th AOSD Workshop on Aspects, Components and Patterns for Infrastructure Software- ACP4IS*, pp. 19-25, 2005.

[15] Lohmann D. et al. A Quantitative Analysis of Aspects in the eCos Kernel. *In Proceedings of EusoSys2006*, Leuven, Belgium, 2006.

[16] Mahrenholz, D., Spinczyk, O., Gal, A., and S-Preikschat, W. An Aspect-Oriented Implementation of Interrupt Synchronization in the PURE Operating System Family. *In Proceedings of 5th Workshop on Object Orientation and Operating Systems - ECOOP*, pp. 49-54, Malaga, Spain, 2002.

[17] Mahrenholz, D., Spinczyk, O., and S-Preikschat, W. Program Instrumentation for Debugging and Monitoring with AspectC++. *In Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing - ISORC*, pp. 249-256, Washington, DC, USA, 2002.

[18] Montenegro, S., and Zolzky, F. BOSS /EVERCONTROL OS/Middleware Target Ultra High Dependability. *In Proceedings of Data Systems on Aerospace -DASIA, Edinburgh*, Scotland, 2005.

[19] Montenegro, S., Briess, K. and Kayal H. Dependable Software (BOSS) for the BEESat Pico Satellite. *In Proceedings of Data Systems on Aerospace - DASIA*, Berlin, Germany, 2006.

[20] Randell B. System Structure for Software Fault Tolerance. *IEEE Trans. Software Engineering*, vol. SE-1, pp. 220-232, June 1995.

[21] Salles, F. et al. MetaKernels and Fault Containment Wrappers. *In Proceedings of the 29th Annual Symposium on Fault-Tolerant Computing*, pp. 22-29, Madison, WI, USA, 1999.

[22] Szentiványi, D., and Nadjm-Tehrani, S. Aspects for Improvement of Performance in Fault-Tolerant Software. *In Proceedings of the 10th IEEE Pacific Rim International Symposium on Dependable Computing*, pp. 283-291, 2004.