On Aspect-Orientation in Distributed Real-time Dependable Systems *

Andreas Gal, Wolfgang Schröder-Preikschat, and Olaf Spinczyk
University of Magdeburg
Universitätsplatz 2
39106 Magdeburg, Germany

{gal,wosch,olaf}@ivs.cs.uni-magdeburg.de

November 1, 2001

Abstract

The design and implementation of distributed real-time dependable systems is often dominated by non-functional considerations like timeliness, object placement and fault tolerance. In this paper we illustrate how the of the *separation of concerns* approach of aspect-oriented programming can be used to address these non-functional aspects of such system separately from the functional requirements and what benefits result from this separation. Besides the theoretical reflection we also present a case study in which distribution, timeliness, and fault tolerance aspects are added to a simple logging component. The examples are implemented using the emerging AspectC++ language for aspect-oriented programming with C++. In the course of this paper we also evaluated how well the general-purpose aspect language AspectC++ is suited to address the aspects specific to this domain.

1 Introduction

Distribution, timeliness, and dependability are non-functional properties of program code. From an application programmer's point of view, it would be desirable to focus on the development of functional program code first, and than adding distribution, real-time and dependability aspects at a later point in time. This is especially true for modular component systems, where components are created, which should be equally useful in different application environments, for example ranging from applications with very lax timing constraints to hard real-time scenarios.

In contrast, in real-world Distributed Real-time Dependable Systems many components directly depend on some library, middleware layer or operating systems services. The variety and diversity of the programming interfaces requires to decide for a specific

*This work has been partly supported by the German Research Council (DFG), grant no. SCHR 603/1-1 and SCHR 603/2.

middleware or operating system very early in the development process. Once code has been written for a specific programming interface, switching over to a different interface is a very difficult and error-prone process, which has to be performed manually. Even worse, limitations or special features of the libraries often also affect the design process. This makes an efficient reuse of components unfeasible.

To circumvent this problem, it seems natural to address non-functional requirements like distribution, timeliness and dependability separately from the functional component code. This is exactly the problem aspect-oriented programming (AOP) [6] addresses. The aim of this paper is to show how AOP can be applied in the Distributed Real-time Dependable Systems domain and what the benefit are.

The rest of the paper is organized as follows. In section 2 we introduce the concept of aspect-oriented programming and demonstrate using simple examples how distribution, timeliness, and dependability aspects can be modularized and implemented separately from the functional component code. Related work is presented in section 3. Our conclusions and a road-map for our future work are contained in section 4.

2 Separation of Concerns

Following the principle of *separation of concerns*, the idea of AOP is to separate the so called component code from the aspect code. The aspect code can consist of several aspect programs, each of which implements a specific aspect in a problem-oriented language. Then an *aspect weaver* takes the component and the aspect code, interprets both, finds join points and weaves all together to form a single entity. This process is illustrated in figure 1.

Different approaches have been proposed when the weaving process should be performed. They all have in common that at runtime the functional and non-functional program code parts have to be woven together. This means that the machine-level instructions that stem from aspect code and those that stem from functional component code are mixed and executed in an order defined by the aspect programs and the aspect weaver. Both, runtime and compile-time mechanism are suitable to fulfill this requirement.

Runtime code weaving is often used in combination with virtual machine based or interpreted languages like Java and TCL [8]. For languages that produce machine-executable code by compilation like C or C++ often source code transformation is preferred, as there is no virtual machine, which could be instructed to inject aspect code at certain join points.

In this paper we will focus on source code transformation based aspect weaving at compile time. For our examples we chose C++ as component language and the aspect programs are implemented using AspectC++ [4]. AspectC++ is preprocessor-like compiler which supports a superset of the C++ language. This language superset contains constructs to identify join points in the component code and to specify advice in form of code fragments that should be executed at these join points. The output of the AspectC++ compiler is plain C++ code, which can be translated with standard C++ compilers to executable code.

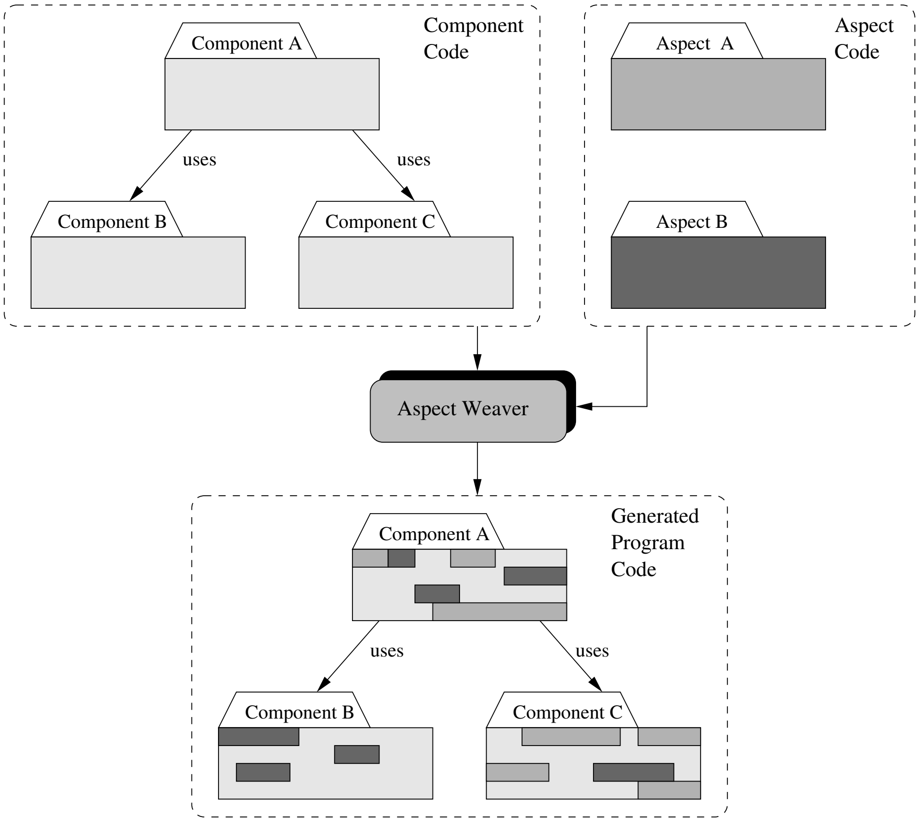

Figure 1: Weaving of aspect code into component code

2.1 Distribution aspect

When writing distributed applications, often one of the first steps is to select a distribution middleware. Beside the plain interface syntax the semantics are also often quite different. For example, CORBA enforces the use CORBA-specific data types and dynamic memory allocation in certain cases. This does not necessarily match with other middleware platforms. Thus, selecting a distribution middleware is often a decision for the whole life-time of an application.

Following the concept of aspect-oriented programming allows to separate the component code and the aspect code. In figure 2 the source code of a simple message logging class is shown. This example contains pure functional component code. The code was written without respect to any possible distribution scenarios.

In figure 3 the AspectC++ code for a client remote access aspect is shown. In the lines 1-2 a pointcut is defined. In AspectC++ a pointcut is a set of join points where aspects affect the component code. The pointcut in this example refers to a single join point, which is the *output()* method of the class *Log*. In lines 10-14 an advice for

```
class Log {
public:
  void output(const char* text) {
    cout << text << endl;
  }
};
```

Figure 2: Not distribution-aware component code

```
pointcut log_method(const char* arg) =
  executions("void Log::output(arg)");

aspect LogClient {
  CORBA::Object_ptr serverLog;

  void create_request(CORBA::Object_ptr obj, const char* msg) {
    CORBA::Request_ptr req = obj->_request("output");
    req->add_in_arg("msg") <<= msg;
    return req;
  }

  advice log_method(msg) : void around(const char* msg) {
    CORBA::Request_ptr req;
    req = create_request(serverLog, msg);
    req->invoke();
  }
};
```

Figure 3: Aspect code for the client side

this join point is declared. In this case the *around* advice replaces the original code at the join point with special code for the remote execution. In this example we use the CORBA middleware and the Dynamic Invocation Interface to relay the execution to the server. The attribute *serverLog* is assumed to hold a valid reference to an instance of *Log* on the server side.

The corresponding server side aspect code is shown in figure 4. The server code does not need any of the AspectC++ language extensions to the C++ language. The server side uses the CORBA Dynamic Skeleton Interface (DSI) to receive the requests from the clients. The virtual method *invoke()* is called by the CORBA middleware each time a request is received. In lines 3-13 the argument is unmarshalled before in line 15 the implementation of the *output()* method is invoked.

Naturally it is possible to implement much more complex scenarios with AspectC++ and the AOP approaches completely encapsulates all middleware-specific code into the aspect codes. This enables to support many different middlewares with the same component code by writing dedicated aspect programs for each middleware.

```
class LogServer: public virtual ServantBase, private Log {
  void invoke(CORBA::ServerRequest_ptr req) {
    if (strcmp(req->operation(), "output"))
      throw CORBA::BAD_OPERATION(0, CORBA::COMPLETED_NO);

    CORBA::NVList_ptr args;
    orb->create_list(0, args);
    CORBA::Any a;
    a.replace(CORBA::_tc_string, 0);
    args->add_value("", a, CORBA::ARG_IN);
    req->arguments(args);
    const char* msg;
    *(args->item(0)->value()) >>= msg;

    Log::output(msg);
  }
};
```

Figure 4: Aspect code for the server side

2.2 Real-time aspect

In real-time systems components do not only have to perform operations correctly, but also have to meet certain timing requirements. General purpose components like graphical user interface frameworks are often not design with a real-time scenario in mind and thus real-time programmers are many times forced to build large parts of their applications from scratch. Building components suitable for real-time applications is a difficult task, as besides the functional requirements attention has also to be paid to the non-functional timing requirements. This additional complexities makes building real-time components more expensive and error prone than general purpose components.

In this section we will show how aspect-oriented programming can be exploited to allow the component programmer to address the functional parts of the component separately from the non-functional timing behavior.

A simple way to monitor and regulate the execution of component code with unknown or unpredictable execution time boundaries is to apply execution time surveillance. A per-thread watchdog timer is used to ensure that the execution of a request the the component untrusted in terms of timing is processed within the given time bounds. The watchdog has to be started when the component code is entered and can be stopped if the component processed the request successfully and timely (figure 5). If the assigned time slot expires before the component code has terminated processing the request, an error condition is raised (figure 6).

Without aspect-oriented programming, the necessary watchdog code has to be inserted manually into the component code. This does not only tangle the source code of the component but also adds unnecessary overhead if the component is used outside the real-time domain.

The described problem can get even worse if the component is only subject to execution time restrictions when called from certain specific places in the application. To

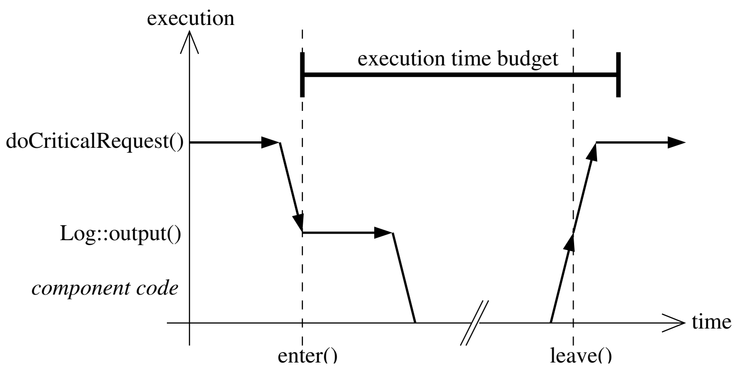

Figure 5: An monitored logging request is processed within the time bounds

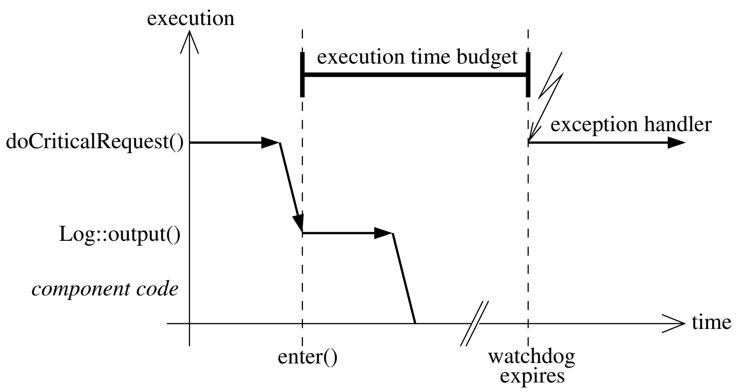

Figure 6: A logging request was not processed timely and is aborted

cope with this situation application specific logic would have to be added to the component code to decided from where the request was issued and what timing constraints apply.

Aspect-oriented programming can be used to cleanly separate between the functional component code and the non-functional real-time properties. In figure 7 the AspectC++ code of a simple execution time surveillance aspect is shown. In this example the *output()* method of the class *Log* is guaranteed to complete within 10ms or an error condition will be raised by the guard. The pointcut declaration in line 2 identifies the *output* methods within the class *Log*. The *cflow* pointcut designator selects only those join points which appear within the dynamic execution context of the method *doCriticalRequest()*. Thus, the execution time limits only apply if the request was issued from a control flow originated in *doCriticalRequest()*.

The around advice in line 4 specifies code to be executed before and after the component code at the join point. *proceed()* is replaced by AspectC++ with the original

```
aspect GuardedExecution {
  pointcut guarded() = executions("void Log::output(const char*)") &&
                       cflow(executions("doCriticalRequest"));
  advice guarded() : void around() {
    guard.enter(10); // 10ms
    proceed();
    guard.leave();
  }
};
```

Figure 7: Aspect code for execution time surveillance

code at the join point.

2.3 Fault-Tolerance aspect

In our final example we will show how aspect-oriented programming can be exploited to implement a fault-tolerance aspect as an extension of the distribution aspect. Consider a scenario where a mobile robot has to move through an area and needs to stay in contact with supervisor nodes. Three base stations are spread over the arena, but environmental conditions may prohibit communication with some nodes at certain points in the terrain (figure 8). For the application to work properly it has to be ensured that the log messages reach at least one of the supervisor nodes. Figure 9 shows the AspectC++ source code of the corresponding aspect program for the client side. The aspect is derived from the distribution aspect (line 5) as the fault-tolerance algorithm used in this scenario can be seen as an extension of the distribution aspect described earlier in this paper. The aspect code in lines 9-16 uses CORBA to send multiple request in parallel. In line 15 the execution blocks until at least one server has responded.

To use the described fault-tolerance aspect code in conjunction with the execution time surveillance aspect code, AspectC++ has to be informed to execute the execution time surveillance aspect code first. This can be achieved using a *dominates* statement (figure 10). When both aspect programs are applied on the component code, the resulting woven code will attempt to contact all base stations and transmit the message to them. In the case that no base station responds, the execution time on the client side is still bound as an error condition is raised when the assigned time window is used up.

2.4 Discussion

The general purpose aspect language AspectC++ is not necessarily equally well fitted to implement all aspects. In this section we will try to highlight the advantages and weaknesses of AspectC++ with respect to this specific application domain.

From the expressive power point of view AspectC++ has at least the same expressive power than AspectJ, which currently is the most popular general purpose aspect-language. Therefor our choice to using use AspectC++ represents the current state of the art in the aspect-oriented programming field.

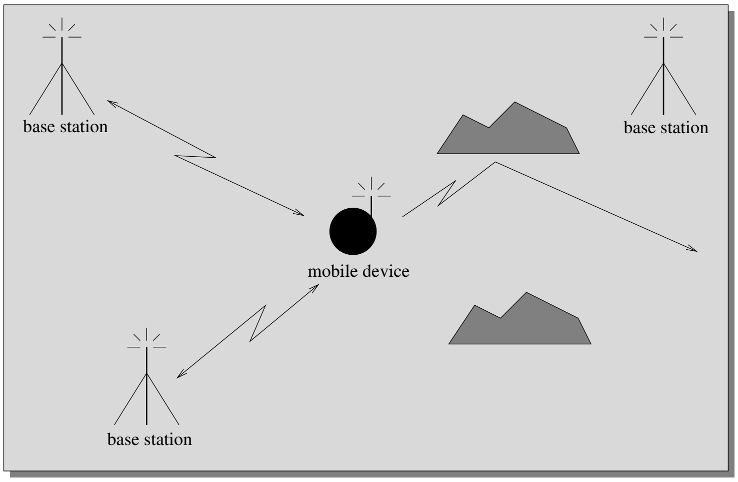

Figure 8: A mobile robot communicating with spread base stations

Section 2.1 gave an example how AspectC++ can be used to add an inner-component boundary. CORBA was used to connect both sides of this boundary, thus distribution of functions to different network nodes becomes possible. All the middleware specific code could be cleanly encapsulated. It was not necessary to do any CORBA specific calls in the component itself, thus CORBA can be easily replaced with some other middleware platform. Nevertheless the result is not completely satisfying. Normally designing a distribution scheme for an application is not done on the class or function level, but on the level of large scale clusters of components. AspectC++ currently does not provide any language element to go onto this abstraction level in the implementation of the distribution aspect. For each class with a network-wide accessible interface its own aspect has to be implemented and for each method a special advice must be given. The result is a massive replication of code that follows a common pattern. To solve this problem it would be necessary to extend AspectC++ with features like *aspect templates* or a *reflection mechanism*, which allows aspect code to explore the class structure, methods, and arguments of the component code in an imperative way.

The second example (section 2.2) showed how monitoring and execution time surveillance can be implemented with AspectC++. In this case the expressive power of the language was sufficient for the task. Especially the concept of control flow identification (*cflow*) promises interesting applications. In this case it allows to have conditional time constraints, which depend on the path in the call graph that was used before the join point was reached.

The example of a fault-tolerance aspect (section 2.3) was built on top of the distribution example. It shows that aspects can easily be extended by using aspect inheritance. At the same time a major weakness of AspectC++ can be observed in this example. AspectC++ does not offer any means to remove methods or attributes from classes.

```
pointcut log_method(const char* arg) =
  executions("void Log::output(arg)");

aspect FTLogClient dominates GuardedExecution :
public LogClient {
  CORBA::Object_ptr serverLog[3];

  advice log_method(msg) : void around(const char* msg) {
    CORBA::Request_ptr req[3];
    for (int i = 0; i < 3; i++)
      req[i] = create_request(serverLog[i], msg);
    CORBA::ORB::RequestSeq rseq(3, 3, req, CORBA_FALSE);
    orb->send_multiple_requests_deferred(rseq);
    CORBA::Request_ptr rreq;
    orbp->get_next_response(rreq); // wait for first response
    CORBA::release(rreq);
  }
};
```

Figure 9: Aspect code for fault-tolerant logging

```
aspect GuardedExecution dominates FTLogClient {
  ...
};
```

Figure 10: Establishing the execution order of aspects

The attribute *serverLog* of the distribution aspect has to be overloaded in the derived aspect. While it is shadowed by the new definition, it does still exist and consumes memory resources. The same applies to classes manipulated by aspects. Even if any access to an attribute of a certain class is removed by an aspect there is currently no way in AspectC++ to remove the class attribute itself.

3 Related Work

A number of other aspect languages have been proposed, including AspectJ [5] for Java. AspectJ is currently the most advanced and popular general purpose aspect language. As Java is still considered experimental in the real-time domain [1], we decided to use C/C++ as component language, which is predominant in this domain. Besides AspectC++, which can be used for C and C++ code, a very similar general purpose aspect language AspectC[3] has been proposed. AspectC is in many ways very similar to AspectC++, but does not support object orientation which we consider crucial in designing complex applications like distributed systems.

The idea of Metaobject Protocols is in some ways similar to aspect oriented programming. This is especially true for static ahead-of-time implementations like OpenC++[2]. OpenC++ is a parser and source code manipulator for C++. Meta programs can be written in C++ and can be used to define new syntax, new annotations, and new object behavior.

As aspect programs do not have the same imperative structure than usual C++ code, from our experience it is more comfortable to use special purpose languages for aspect-oriented programming. The AspectC++[4] language designed for this reason has been implemented using PUMA[9]. PUMA is as OpenC++ a C++ parser and source code manipulator, streamlined toward being used by aspect weavers.

Reflection is also known to be useful for adding non-functional aspects to applications[7]. In this paper the authors use reflection for incorporating fault-tolerance techniques into distributed applications. Their work differs from ours as we have focused more on analyzing the general helpfulness of the aspect-oriented technique than on specific fault-tolerance, or distribution algorithms.

4 Conclusions and future work

Distribution, real-time, and fault-tolerance are non-functional requirements on a software product. AOP allows to separate the implementation of these aspects from the core functionality, thus a development done by specialized teams becomes more realistic than it is today. The software components do not need to contain the code for distribution, real-time, or fault-tolerance. Instead they will be enriched with the missing features by the aspect weaver without needing to edit the component code manually, which is an error-prone process. The decision what middleware platform or library is to be used is left to the user. This gives the user more flexibility and reduces the work load for the component vendor at the same time.

It is important to understand that aspect orientation does neither make incorrect software correct nor does it allow to use algorithms with unbounded worst case execution times in hard real-time environments. Aspect-oriented programming can be no replacement for thorough design and measurements when implementing real-time systems. But it seems ideally suited to model and implement software components that should be used with different distribution, real-time, and fault-tolerance requirements in various projects. The growing complexity of applications demands exactly for this increased reusability of components.

Currently we see the main drawback of aspect-oriented programming in the lack of powerful tool support. Even though the AspectC++ language has been specified the implementation is still far from being complete. AspectJ is much more advanced, but whether it can be applied in this domain stands and falls with the success of Real-Time Java.

In the near future we will attempt to improve the AspectC++ compiler implementation to make it usable for large real-world projects. We will use the enhanced AspectC++ compiler to experiment more complex application scenarios. We are especially interested how aspect-oriented programming can be used to support distributed check-pointing.

References

[1] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull. The Real-Time Java Specification, 2000. `http://www.javaseries.com/rtj.pdf`.

[2] S. Chiba. Metaobject Protocol for C++. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 285–299, Oct. 1995.

[3] Y. Coady, G. Kiczales, M. Feeley, N. Hutchinson, J. S. Ong, and S. Gudmundson. Exploring and Aspect-Oriented Approach to OS Code. In *Proceeding of the 4th ECOOP Workshop on Object-Orientation and Operating Systems (ECOOP-OOSSWS'2001)*, pages 55 – 59. Universidad de Oviedo, June 2001. ISBN 84-699-5329-X.

[4] A. Gal, W. Schröder-Preikschat, and O. Spinczyk. AspectC++: Language Proposal and Prototype Implementation, Aug. 2001. Submitted to the AOP workshop at the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2001).

[5] G. Kiczales, E. Hilsdale, J. Hugonin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In J. L. Knudsen, editor, *ECOOP 2001 – Object-Oriented Programming*, volume 2072 of *LNCS*. Springer-Verlag, June 2001.

[6] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, June 1997.

[7] A. Nguyen-Tuong and A. S. Grimshaw. Using reflection for incorporating fault-tolerance techniques into distributed applications. *Parallel Processing Letters*, 9(2):291–301, 1999.

[8] R. Pawlak, L. Duchien, G. Florin, L. Martelli, and L. Seinturier. Distributed Separation of Concerns with Aspect Components. In *Proceedings of the 33rd International Conference on Technology of Object-Oriented Language (TOOLS 33)*. IEEE Computer Society, June 2000. ISBN 0-7695-0731-X.

[9] O. Spinczyk and M. Urban. The PUMA Project Homepage, 2001. `http://ivs.cs.uni-magdeburg.de/˜puma/`.