Using aspects for integrating a middleware for dynamic adaptation

Gaëtan Vaysse, Françoise André, Jérémy Buisson

Using Aspects for Integrating a Middleware for Dynamic Adaptation

Gaëtan Vaysse
IRISA/INRIA
Rennes, France
Gaetan.Vaysse@irisa.fr

Françoise André
IRISA/Université de Rennes 1
Rennes, France
Francoise.Andre@irisa.fr

Jérémy Buisson
IRISA/INSA
Rennes, France
Jeremy.Buisson@irisa.fr

ABSTRACT

Middlewares are designed to hide common concerns in software development. One of the challenges in middleware development is to conceive how they can be integrated into applications in an efficient manner and at an acceptable cost for developers. This article reports results of practical experiences of integrating Afpac, a middleware for dynamic adaptation, thanks to aspect-oriented programming. Using current techniques, places where applications and middlewares can interact are limited to boundaries of application entities. In the case of Afpac, this is not sufficient. This paper details the issues that arise when integrating Afpac and proposes a set of features an aspect weaver should have to be usable in that case. As usual weavers do not provide those features, a weaver has been developed in order to demonstrate the feasability of this approach.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; D.1 [**Programming Techniques**]: Miscellaneous; D.2.11 [**Software Engineering**]: Software Architectures

Keywords

dynamic adaptation, aspect-oriented programming, middleware integration

1. INTRODUCTION

In this paper, we consider that applications are assemblies of components built on top of middlewares. The word "component" abusively stands here for building block without any further detail. Integrating the use of middlewares into components requires a lot of code. Usually, this code is generic and disseminated. It explains why aspect oriented development is investigated in middleware development. Indeed, aspects provide means to introduce code at given places inside an application. In addition, the notion of aspect is very close to the one of concern. A concern can be seen as a set of behaviours responding to certain environment criteria. This notion takes an important place in middleware development as middlewares usually aim at extracting secondary concerns (such as security, distribution, and so on) from components. In that way, implementation details of those concerns are hidden to application developers. Aspect orientation can thus be used for the integration of middlewares with components. This allows developers to see middlewares as black boxes. They can then focus on the effective code instead of bothering with the integration code.

Afpac [5] is a middleware that captures the concern of dynamic adaptation. In order to use it, the developer has to introduce into his code some particular statements named adaptation points, which indicate where the middleware is allowed to make the component react to changes of its environments. The integration of Afpac requires the component to notify its middleware each time it enters control structures. In that way, Afpac knows the actual progress of the execution and can identify upcoming adaptation points.

The huge amount of statements to add makes manual integration of Afpac unpracticable. This paper focuses on how this integration can be hidden to developers. Section 2 describes how middleware integration is usually done. Section 3 details how Afpac must be integrated to a client code. It shows in what techniques described in section 2 are not sufficient in the case of Afpac. Section 4 depicts the features we propose for an aspect weaver to integrate Afpac. Section 5 shows how the integration of Afpac can be realized with such a weaver. A comparison with existing weaving techniques is given in section 6. Endly, section 7 gives some perspectives this case study opens to aspect oriented programming.

2. MIDDLEWARE INTEGRATION TOOLS

Middlewares aim at easing application development by providing high-level abstractions. Lying in the middle between applications and operating systems, low-level implementation details of those abstractions are completely hidden to the developers. For instance, usual abstractions provided by middlewares are communicating entities such as objects, components and services. Such middlewares mask as much as possible the implementation details of the communications between the entities.

Whatever the provided abstractions, middlewares have to be integrated into applications in an easier way than if the abstractions were reimplemented specifically for the application. Otherwise, middlewares are useless. Tools for easing middleware integration are thus of primary importance.

AOMD'05 November 20-December 2, Grenoble, France

2.1 Interceptors

An interceptor is an element of a chain of responsability that can modify a given message. If messages are considered as reified method invocations, interceptors allow to alter the semantic of method invocation. For example, an interceptors can be used to execute code before and after invocations. They can also be used to implement transparently remote invocation to the component.

2.1.1 Services integration

A service is a communicating software entity that performs some expected task. It interacts thanks to message exchange. In order to be effectively integrated with services, an application often needs to have glue code directly written into it. This glue code has the responsability of managing interactions with services for example by generating and parsing messages.

Several services may be expected to perform semantically the same task. However, those services may assume distinct schemas for messages. When such services are integrated, the glue code is required to be duplicated for each possible schema. Letting the developer writes many versions of the glue code by its own is not acceptable.

One solution to this problem consists in adding an adapter to the code of the client. This adapter can be seen as an interceptor inserted between the client and the service. It depends on the service that the client connects to. This approach separates the concern of integrating the service from the application itself. If this adapter is built automatically as it is done with the WSML [7] project, the developer would not have to consider creating code for each possible schema.

If in addition the adapter is inserted dynamically, this approach makes the client able to benefit from service discovery services to select the best service independently of the schema it expects. Dynamic aspect weaving permits such a dynamic insertion. As described in [14], WSML does so: it redirects client requests to the right service. These *redirection aspects*, described in [15], are plugged dynamically thanks to the JAsCo [6] dynamic aspect weaver.

Whereas WSML targets web services, the same ideas apply to other kinds of entities when several semantically equivalent entities export syntactically different interfaces.

2.1.2 Container

In many component middlewares such as CCM [1] or EJB, components are isolated from their outside world by containers. This entity is commonly used for handling life cycle and dependencies of the contained component like PicoContainer [2] does. Dependency concern can be solved using dependencies injection, i.e. outsourcing to an external entity the management of bindings. When a component explicitly describe its dependencies, its container can transparently resolve its bindings.

As interactions with the contained component are made exclusively through its container, the container is a natural placeholder for interceptors. In fact, the container itself may be seen as an interceptor. For example, CCM defines in its architecture portable interceptors to hook the control flow between components while it is within the middleware. Such a mechanism allows for example to implement the access control concern without any impact on the source code of the component.

2.2 Mixins

Alternatively to intercepting the interactions of the component, the middleware can be integrated by mixing the building blocks of the middleware with those of the component. For example, in an object-oriented environment, classes could be mixed. When two classes are mixed together, their functionalities are merged. This approach is in particular the one used in the implementation Julia of the Fractal [4] component model to integrate controllers to the component.

Concretely, a mixin class can be seen as a declaration of the differences between a class and its specialization. However, as the general class is not specified, the specialization is generic and can thus be applied to many classes. Projects such as Jam [3] provide tools to apply mixins to classes.

2.3 Aspects

Aspect oriented programming aims at providing solutions to integrate concerns in applications where classical approaches such as object or component oriented fail. Aspects are particularly well suited for concerns that literally crosscut applications, i.e. concerns that do not fit well with the natural decomposition. It can be observed that the concerns taken in charge by middlewares commonly crosscut applications. This makes aspect oriented programming a promising mean to integrate middlewares.

Techniques used to weave aspects are various. The language can be domain-specific as with AspectJ or JAsCo. Alternatively, as with Jac [12], aspects can be better integrated into the user code if expressed with the same language as user code. Aspects can be woven at compile-time (AspectJ) or load-time (Jac). Most aspect languages are designed for object-oriented environments. However, the concept of aspect can be applied to other paradigms. For example, Aspectual Caml [13] implements aspect weaving for functional languages.

2.4 Relations between tools

Interceptors are software elements placed between the client that invokes a method and the component targeted by the invocation. Installing an interceptor can thus be seen as weaving an aspect around the invocation and/or the method. In fact, the cited example of WSML uses aspect-oriented techniques to install its interceptors. Containers, as placeholders for interceptors, may be useful to implement the weaving process around those join points. Conversely, as containers intercept interactions with contained components, aspect-oriented programming can be used to generate containers. The former approach is similar to [8]; the latter is the approach of Jac that relies on aspect weaving to generate customized containers.

As for specialization in object-oriented design, a mixin class is allowed to redefine methods of the base class. In that case, mixin is one way to install an element in the responsability chain for the invocation of the method. Mixins thus seem to provide a superset of the interceptor features.

Mixins have been described as a declaration of generic specializations of classes. At a very low level of abstraction, an aspect can be seen as a declaration of a transformation of a software entity. In fact, specializing a class consists in introducing and redefining attributes and methods. This kind of transformation is usual in aspect-oriented programming. Mixins may thus be considered as a subset of aspects.

3. INTEGRATION OF AFPAC

Afpac is a middleware made to simplify the development of dynamically adaptable components. As its role is not related to interactions between components, it is integrated mostly inside components, instead of between them.

3.1 Requirements

As the core of its interactions with the component, Afpac introduces the notion of adaptation points, i.e. places in the control flow where the component behaviour can be modified to handle changes in the environment. For instance, in the context of grid computing, when new processors are added, the middleware may decide to make the component adapt itself so it takes the new nodes into account. In that case, adaptation points denote states at which data are not transient and can thus be redistributed.

To make its decision, the Afpac middleware needs to know precisely the state of the component. That is to say: for conditional instructions, which branch has been entered; for loops, what is the current iteration; and so on. These information are checked against a control flow graph that is given to Afpac. Thanks to this static model, the middleware can predict the next adaptation points in the execution. Those points are the one allowed to execute the adaptation.

3.1.1 Annotations

In order to allow the Afpac middleware to track the execution, the middleware has to be notifed by the component of its dynamic behavior upon each control structure. Figure 1 shows how the code of the component must be annotated to perform this notification.

```
1   int f(int i) {
2       FunctionEnter("int f(int)");
3       int result;
4
5       if (ConditionalEnter(i > 0)) {
6           g();
7           adaptationPoint();
8           result = h()+1;
9       } else
10          result = h();
11      ConditionalLeave();
12      FunctionLeave();
13      return result;
14  }
```

Figure 1: Adaptable code

Interestingly, the presence of annotations does not depend whether it is the first or second branch of the conditional instruction that is chosen, but if at least one of them contains an adaptation point. It does not depend either on any run-time context. Annotations can thus be added statically. Moreover, the manner the code has to be annotated remains constant and seems automatic.

In addition, any function that contains an adaptation point must be considered as an adaptation point itself. In the given example, any function calling "f" should thus be annotated in the same way. On the other side, control structures and functions that do not contain any adaptation point should not be modified as Afpac does not need to track the progress of their code. Not annotating those structures lowers the overhead. During the integration, the transitive closure of the call graph is required to determine precisely which control structures and functions should be modified.

3.1.2 Control flow graph

In addition to annotations for tracking the execution of the component, the Afpac middleware requires a static representation of the control flow graph of the component. Afpac expects this structure to be a tree view of the hierarchical task graph in order to preserve high level constructs such as loops. This structure can be obtained by modifying the source code of the component. Figure 2 shows the modified code of figure 1 for the control flow graph, where comments recall the original source code.

```
1   // int f(int i) {
2   Function* f = new Function("int f(int)");
3   //      if(i > 0)
4   Conditional* c = new Conditional;
5   //      {
6   //          adaptationPoint();
7   c->ifTrue()->append(new AdaptationPoint);
8   //      } else {
9   //      }
10  f->append(c);
11  // }
```

Figure 2: Control flow graph

Although the resulting source code differs from the one for annotations, the criteria for deciding whether a control structure or a function should be modified are the same in both cases.

3.2 Integration

As Afpac aims at capturing the concern of dynamic adaptation, developers of components should not bother about it. Figure 3 shows the highest level of intrusion that can be acceptable to use a middleware for dynamic adaptation: developers only have to put adaptation points. They should not have their view of the code polluted by additional statements.

```
1   int f(int i) {
2       if (i > 0) {
3           g();
4           adaptationPoint();
5           return h()+1;
6       } else
7           return h();
8   }
```

Figure 3: Almost original code

Tools described in section 2 are useless in the case of Afpac. Indeed, those tools focus on integrating functionalities between components, whereas dynamic adaptation is internal to each component.

Nevertheless, the two required modifications can be easily seen as transverse functionalities as they are disseminated through the whole code of component. Aspect thus seems a particularly well suited concept for expressing the integration of Afpac to components.

4. FINE GRAIN WEAVER

Our static weaver, called Taco, focuses on the features required to weave the use of Afpac to a component, while remaining usable in other contexts. It weaves C++ codes.

4.1 Join points

Join points weaved by Taco are the control structures. Internally to the weaver, they are organized as a tree that is close to the abstract syntax tree of the target source code. Join points are reified as triples containing:

- One type: loop; conditional instruction; call; unconditional branch instruction; and so on.
- Some parameters: for example a boolean expression for a conditional instruction; two instructions and a boolean expression in the case of a *for* loop.
- Some bodies: two for a conditional instruction; one for a loop; none for a unconditional branch instruction.

4.2 Advices

Aspects are declared with a domain-specific declarative language. They associate cuts, subsets of join points, to advices. However, manipulating control structures with the required expressiveness would require so much language constructs in a purely domain-specific language that it would be too complex. For this reason, in addition to quoting C++ code, advices are given the ability to perform static manipulations of the join points they are applied to. Those manipulations are expressed through commands within advices thanks to a general purpose language.

While it permits an easier integration of the elements of the reified join points inside the inserted code, commands can be used to generate code depending on the context of the join point. They can be used for instance to transform a *for* loop into a *while* loop without having to create specialized constructs. Figure 4 shows an advice that does that modification. Commands are used to access and paste the different elements of the *for* structures.

```
1  replaceAll() : is(for) {
2      $( echo(current():start()) $);
3      while ( $( echo(current():stop()) $) ){
4          $(
5              echo(current():body())
6              echo(current():step())
7          $)
8  }     }
```

Figure 4: Transforming a control structure, commands are in $(... $) braces

The weaver exposes to commands the reified form of join points. As commands are executed statically, no support for reflexivity in neither the target program nor the target language is required.

Commands can also be used to generate additional data that can be seperated from the woven source code. Such data may even be perssitent, and used during another weaving process. They can also be used to generate source code. For example, it could be used to generate new symbols while enforcing their uniqueness.

4.3 Pointcuts

The pointcuts of that weaver are patterns that comes in the form of boolean expressions. Some operators offered for these expressions can be used to access the environnement of the current join point, i.e., the preceding or following join points, and its parents or children. There is the possibility to check the environnement of the current node against certain conditions. A join point may be a cut depending on the result of such a check. As Taco weaves statically, pointcuts are evaluated statically. For this reason, tests available for writing pointcut expressions check static properties of the join points. For example, Taco provides test operators on the current point, (is it a *for*), its parent, or other indirect ancestors (is it within a *for*), and its inner points (does it contain a *for*).

Like the advice system, the pattern system allows to issue commands to the weaver. The aspect developer can then create custom check operators. He can also maintain internal state structures to use during the tests. Those structures can persist and be used through the whole weaving process.

4.4 Implementability

Weaving control structures means modifying recursive structures, because a control structure body can contain other control structures. Thus, a join point can contain other join points of the same kind. Working on a recursive view of the program is more difficult than working on non-recursive view. For example, it could prevent the weave operation to terminate if introduced code is allowed to be weaved. In addition, modifications made on a point may discard its inner points or modifications made on them. That can happen if the current join point is replaced by some other code without using parts of the former version. The parts that were not put back are then lost for the result of the weaving process, possibly preventing some cuts to be satisfied. It then raises the question whether those parts should be weaved.

Instead of answering these questions, it was decided that the weaving process would work bottom-up. A point is always weaved before its surrounding control structure. That way, if the surrounding control structure is only partially reintroduced, the inner points have already been weaved, whether they will be cut or not. That maximizes the numbers of weaved points. As a sequence of statements is always weaved from first to last, it creates a partial order for weaving join points.

As commands allow to pass data from a join point check or advice application to another one through global variables, aspect designers are given an internal state of the weaving process. Having a constant and known sequence of the weaving process helps developing aspects taking profit of it.

5. INTEGRATION OF AFPAC WITH TACO

Inserting the annotations shown in figure 1 can be done using the same construct than the example of figure 4. Calls to Afpac can be inserted at the requested points of the control structures. Thanks to commands, a unique identifier can be generated to create a variable to hold the parameter value of the return statements. Thanks to this, the return statement can be moved to the end of the function in order to ensure its uniqueness. Making global the variable used in commands for this identifier allows to modify consistently all return statements.

The criteria for deciding whether a control structure should be annotated or not can be implemented thanks to the environnement checking capabilities: inner join points can be checked against certain conditions, such as the presence of an adaptation point. Moreover, instead of computing for each call the transitive closure of the call graph, the internal state of the weaver offered by commands can be used to maintain a database of functions containing adaptation points.

Endly, the extraction of the control flow graph has been done thanks to the capabilities of the general-purpose language embedded in the aspect language. For simplicity reasons, the aspect does not actually modify the source code of the component; it generates the modified form to an alternate output stream. Using data transiting from one point to the other allows to generate links between the different nodes. The strong assumptions it makes on the weave order and its unusual behavior for aspects make this application of Taco on the borderline between aspects and program transformation.

6. DISCUSSION

Most code weavers are used in object oriented languages. They usually focus on the call graph or dataflow of the application allowing the user to modify the code where a function is called, or directly at the beginning or the end of that function. They also often allow to introduce new members or methods to classes. Their view of the program is more macroscopic than the one of Taco.

6.1 Differences in the weaving level

The reason why so few projects interest in fine grain weaving may be the lack of applications of this weaving level. In many applications built by making objects interacting with each others, the global behaviour of the application is obtained thanks to those interactions even if object methods retain particular functionality. Changing the behaviour is most easily done by changing the interactions. Modifying the implementation of the methods could be seen as being too close to the program working, whereas in some cases like Afpac, it is necessary.

Another reason for fine grain weaving to be neglected is that slicing the code offers similar possibilites. Instead of weaving a loop, the loop and its body can be separated from the surrounding code in new methods. Modifying the behaviour of the loop body can be done by weaving the corresponding method. The same applies for the loop itself. This concept can be brought further, by encapsulating control structures in objects. Control structures can then be manipulated directly without using a weaver, but it would make writing code harder and unnatural. For instance, writing a conditional would requires to creates an object with two methods, one for each branch. A pro of this method is that it would add a low level reflexivity to the control structures of the program, presenting them in an already reified form, like Taco does.

Nonetheless, some efforts are made to integrate some control structures in aspect oriented languages. University of Manchester proposes in [9] a solution to add a loop join point in AspectJ. Their approach differs from ours as the loop detection occurs after compilation. It also takes the iteration space into account. In this case, the detection has to be done on bytecode, which is more difficult than detecting it on the source code, as lesser information are available. The ability to detect loops is strongly dependent on optimizations made by the compiler. In some cases of nested loops, it may even be not possible to weave *around* the inner loops.

On the other hand, by working before compilation, Taco is insensitive to compiler optimizations. However, relying on high-level constructs of the language, Taco misses loops built manually by the developer thanks for example to the "goto" statements.

6.2 Command system

As the user has to work on a closer view of the modified code, the langage used to modify it becomes wider. There is more control structures and parameters for them in a language like C++ or Java, than there is constructs for specifying objects. Each of loops, jump tables or conditional instructions has its own set of parameters. On the other hand, constructs like interfaces, classes and operations, have lesser parameters. That would make a purely domain specific language too restrictive or too complex to be a good choice for aspects at the level of control structures. This is why commands have been integrated to the aspect language.

A command system has been integrated to allow more interactions with the control structure parameters, it can also be used to modify the aspect behaviour. The commands can be used to collect information on the weaved code, and to pass them from a pointcut pattern operation or an advice to an other one. Those information can also be stored, offering ways to use it after the weaving process finished. This present the user with new possibilities.

One of them is the possibilities to make aspect adapt to the context of its execution. Command variables scope can be larger than a particular join point. It can be the whole weaving process. It can be used to make the weaving process global to the source file, not weaving each join point independently.

But because of this approach, the current implementation of the model extraction aspect works mostly as a stack automaton that takes as inputs the constructs that have to be weaved. This is required to generate links between related nodes of the generated graph as they involve pairs of objects created from distinct join points. For example, the link indicating that a statement belongs to a loop requires the model of the statement created at the statement join point and the model of the loop created at the loop join point. If the weaver permitted the aspect to associate attributes to adviced join points, the aspect would be able to avoid any global variable, and thus any assumption over the weaving process. One counterpart of commands seems to be the temptation for aspect developers to forsake the declarative part of aspects and perform compile-time reflection.

7. OPEN ISSUES AND PERSPECTIVES

As it has been previously said, the aspect used to extract the control flow for Afpac may be nearer from program transformation programs than aspects. This result can be explained by the following points.

Firstly, despite the presence of statically interpreted commands in advices, Taco appeared unable to modify enough the structure of the weaved source code. This aspect would indeed require to be able to transform any construct of the C++ language into the creation of an instance. For example, it should be able to replace the declaration of a class

by the declaration of a variable. In that case, it could even be questioned whether the result of weaving an aspect has to be syntactically correct with regard to the language of the input source code. For example, the control flow graph aspect might have resulted in an XML document instead of a C++ program that instantiates the graph objects.

Secondly, extracting the control flow graph from a program can be questioned whether it belongs or not to the definition of aspect weaving. Surely this is a concern that crosscuts programs. In addition, in the case of the integration of Afpac, pointcuts are interestingly the same for choosing the structures to reify in the control flow graph and those to annotate. For this reason, it is strongly desirable to perform both tasks with the same tool.

Whereas other aspect weavers forsake control flow primitives as join points, it could be used in other context than Afpac. For example, diffenciating automatically a program requires to reverse the order of the instructions. Thus, it requires to modify the control flow. Currently this kind of modification is done by ad-hoc tools such as TAPENADE [10]. It would be valuable to express them as aspects, which would permit to separate the manipulation of program from the differenciation. Similarly, weaving aspects at the level of control structures may allow to declare loop parallelization as an aspect. Again, this would avoid the need for specialized tools.

8. REFERENCES

[1] *The Corba Component Model specification – http://www.omg.org/.*

[2] *PicoContainer related web site – http://picocontainer.org.*

[3] D. Ancona, G. Lagorio, and E. Zucca. Jam – a smooth extension of java with mixins. In *14th European Conference on Object-Oriented Programming*, June 2000.

[4] E. Bruneton, T. Coupaye, and J. Stefani. Recursive and dynamic software composition with sharing. In *Seventh International Workshop on Component-Oriented Programming (WCOP02)*, Malaga, Spain, June 2002. European Conference on Object-Oriented Programming.

[5] J. Buisson, F. André, and J.-L. Pazat. Enforcing consistency during the adaptation of a parallel component. In *The 4th International Symposium on Parallel and Distributed Computing*, July 2005.

[6] M. A. Cibran, M. D'Hondt, D. Suvee, W. Vanderperren, and V. Jonckers. Jasco for linking business rules to object-oriented software. In *International Conference on Computer Science, Software Engineering, Information Technology, e-Business, and Applications (CSITeA'03)*, pages 1–7, Rio De Janeiro, Brazil, June 2003.

[7] M. A. Cibran, B. Verheecke, D. Suvee, W. Vanderperren, and V. Jonckers. Automatic service discovery and integration using semantic descriptions in the web services management layer. In *Third Nordic Conference on Web Services*, Nov. 2004.

[8] Y. Eterovic, J. M. Murillo, and K. Palma. Managing components adaptation using aspect oriented techniques. In *First International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT04, held in conjunction with ECOOP 2004)*, June 2004.

[9] B. Harbulot and J. R. Gurd. A join point for loops in aspectj. In *Foundations Of Aspect oriented Languages workshop (FOAL 2005)*, Chicago, USA.

[10] L. Hascoët, V. Pascual, and R.-M. Greborio. The tapenade ad tool. In *AD Workshop*, Cranfield, June 2003.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In S.-V. LNCS, editor, *the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241, 1997.

[12] R. Pawlak, L. Seinturier, L. Duchien, G. Florin, F. Legond-Aubry, and L. Martelli. Jac : An aspect-based distributed dynamic framework. *Software Practise and Experience (SPE)*, 34(12):11191148, October 2004.

[13] H. Tatsuzawa, H. Masuhara, and A. Yonezawa. Aspectual caml: an aspect-oriented functional language. In *Workshop on Foundations of Aspect Oriented Languages*, page 3950, March 2005.

[14] B. Verheecke, M. A. Cibran, W. Vanderperren, D. Suvee, and V. Jonckers. Aop for dynamic configuration and management of web services in client-applications. *International Journal on Web Services Research*, 1(3), July 2004.

[15] B. Verheecke, M. Cibrn, and V. Jonckers. Aspect-oriented programming for dynamic web service monitoring and selection. In *The European Conference on Web Services 2004 (ECOWS'04)*, Erfurt, Germany, Sept. 2004.