Learning from Components: Fitting AOP for System Software *

Andreas Gal, Michael Franz
Department of Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
{gal,franz}@uci.edu

Danilo Beuche
School of Computer Science
Otto-von-Guericke-University of Magdeburg
Magdeburg, D-39106, Germany
danilo@ivs.cs.uni-magdeburg.de

ABSTRACT

Aspect-oriented programming (AOP) and implementation of system software are fairly complex tasks on their own. Combining these two severe challenges seems to be not very inviting to operating system and system software builders. To make AOP more appealing to the system software community, we limit the sometimes pervasive nature of AOP by applying lessons learned from component-oriented programming to make AOP more manageable and easier to verify. The result is AspectLagoona, a featherweight aspect language with very simple semantics and easily understandable and well specified aspect and component code interaction.

1. MOTIVATION

At first, aspect-oriented programming [7] and component-oriented programming (COP) seem to be totally irreconcilable as far as their fundamental design principles are concerned.

Components are defined as a unit of composition with contractually specified interfaces and explicit context dependencies only [13]. A component can be understood as a black box for which it is only known how to connect to it to request a certain service (Figure 1). *How* the request is processed internally is hidden from the client. This property makes components interchangeable, as all dependencies are on the interface level only.

Aspects have a very different way of interconnecting with components. Instead of communicating with components through well defined interfaces, they have to reach directly inside the component to extract and modularize crosscutting concerns (which might very well even crosscut across component boundaries).

In general, commercial operating system builders tend to be much more reluctant to adapt new technologies than software writers in other domains. Operating system implementors are this conservative for a good reason: it is the responsibility of the OS to facilitate all I/O operations and to manage all hardware resources. While a faulty application might affect a certain single task performed by that particular application, a software error inside the operating system can easily interfere with all applications at once, causing damage to a much greater extent. Operating systems are in general also much more difficult to debug than standard applications as usually little control over the machine is left once the OS crashed.

Probably the most repulsive property of AOP for system software programmers is its crosscutting nature. While a line of component code tends to be connected to one particular action resulting in some predictable local effect, aspect code can sometimes have a subtle, sometimes even "ghostly", influence on the whole system by bypassing established communication protocols and interfaces between components and directly manipulating internal component state.

The interdependence between aspect code and component code is particularly hard to grasp as existing general purpose aspect languages like AspectJ [6] and AspectC++ [12] offer a wide variety of mechanisms for aspects to attach to component code.

To make AOP more attractive for system software developers, it is required to make aspect and component code interaction more obvious by simplifying the involved mechanisms and offering the programmer a mechanism to control where aspect code is allowed to interfere with component code and where not.

In this position paper, we present the language Lagoona [3] and its aspect-oriented extension AspectLagoona. Lagoona is an object-oriented language based on the idea of stand-alone messages and message forwarding. In Section 2 we will discuss the fundamentals of the Lagoona language. Section 3 highlights why stand-alone messages greatly simplify the understanding of aspect-component interaction while still preserving the similar level of expressibility as other general-purpose aspect languages. Section 4 discusses related work while Section 5 finally contains our conclusions and possible future extensions.

2. LAGOONA

The most obvious feature that sets Lagoona apart from established object-oriented programming languages is *stand-alone messages*. In Lagoona messages are bound to (declared in) modules instead of types, whereas most other object-oriented statically-typed languages subordinate messages to classes (Figure 2). Stand-alone messages prevent

*This work was partially supported by the National Science Foundation under grants EIA-9975053 and CCR-0105710.

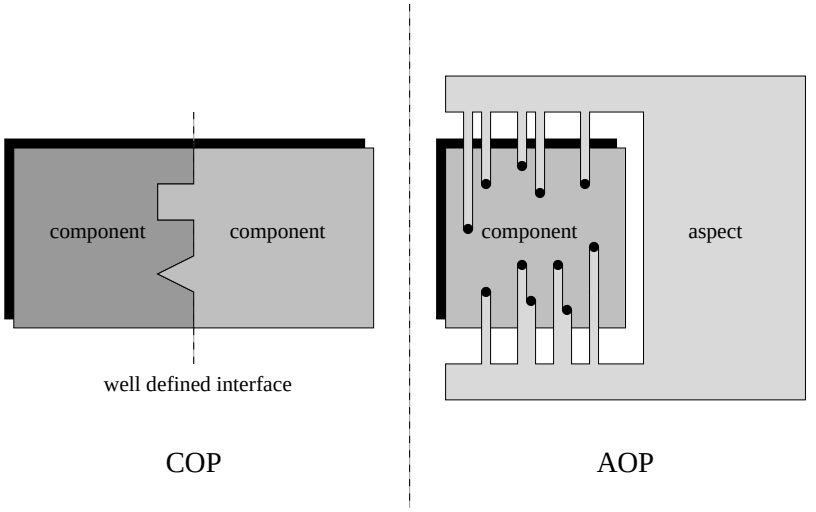

Figure 1: Interdependencies between components and aspects

"accidental" conformance relationships, where for example a `Cowboy` type and a `Shape` type both understanding a message `draw` with different semantics. Similar to the assumption made about interfaces in COM [10], it is assumed that messages and their specification are *immutable* once published and thus have the same meaning to any potential receiver object independently from its type.

Messages are the basis for *interface types*, (`interface` in our concrete syntax) which represent references to objects that implement a certain set of messages. Conformance to interface types is structural. The pervasive interface type `any` represents the empty message set and is the top element in the resulting type lattice. Note that the name we give to an interface type is only a convenient abbreviation; instead of using such a name, we could also declare isomorphic interface types repeatedly. Conceptually, interface types in Lagoona are used to decouple independent components [4], similar to the use of interfaces in both COM [10] and to a certain extent Java [5].

Implementation types (`class` in our concrete syntax) host methods and declarations for instance variables. While messages are *abstract operations* that describe *what* effect they achieve, methods are *concrete operations* that describe *how* an effect is achieved. In other words, messages are specifications for methods, and methods are implementations of messages. Each method implements exactly one message and is triggered when an object of the associated type receives that particular message.

While an interface type is simply a set of messages, an implementation type consists of a set of methods and associated storage definitions. In contrast to messages, methods *are* declared in the scope of an implementation type. This asymmetry is intentional, since we want to support multiple implementations of identical specifications on the level of messages and methods as well as on the level of interface types and implementation types to foster component-oriented programming. As with messages and methods, interface types and implementation types serve as specifications and implementations for each other.

To relate interface types and implementation types (including their instances), we need to define some notion of *conformance*.

First, an interface type B denoting a set of messages M_B conforms to an interface type A denoting a set of messages M_A if and only if M_B is a superset of M_A:

$$A \trianglelefteq B \iff M_A \subseteq M_B \tag{1}$$

In other words, we employ *structural subtyping* between interface types.

Second, an implementation type C with a set of methods implementing a set of messages M_C conforms to an interface type B denoting a set of messages M_B if and only if M_C is a superset of M_B:

$$B \trianglelefteq C \iff M_B \subseteq M_C \tag{2}$$

Third, an interface type never conforms to an implementation type. Of course, Lagoona allows interface types to be *cast* to implementation types, guarded by a dynamic check.

Finally, two implementation types only conform if they are the same type. In other words, we employ *occurrence equivalence* between implementation types.

As instance variables are internal to the associated object, at runtime, Lagoona's object model essentially reduces to a web of independent instances that communicate through messages.

Assume we are sending a message m to a receiver r, which can be an interface or an implementation reference, whose type R denotes a message set M_R. We distinguish two *message send operators* with different semantics.

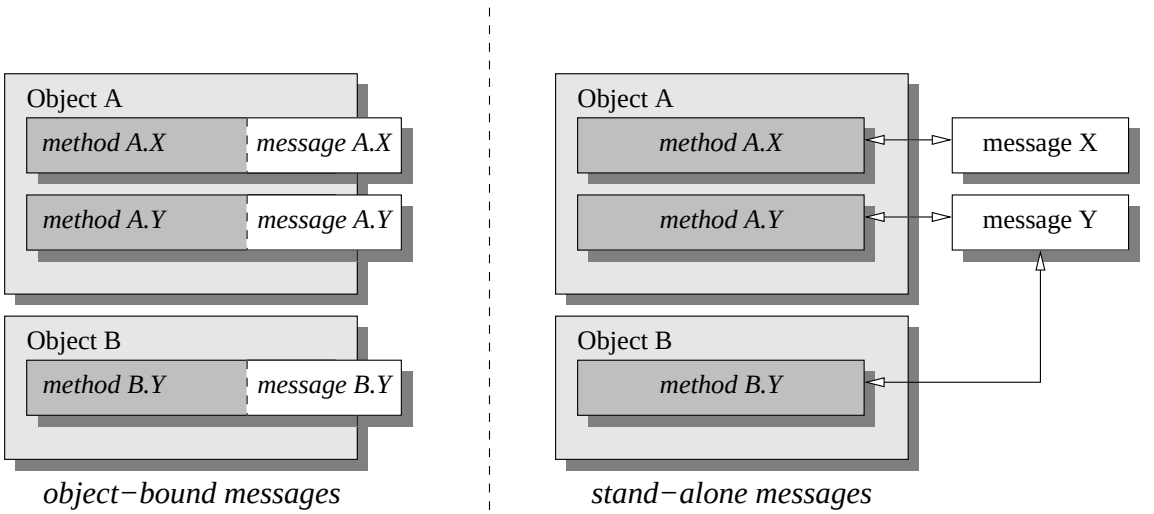

Figure 2: Messages in Lagoona and traditional object-oriented languages

The first operator . is *strict* in the sense that the expression $r.m$ is valid if and only if m is an element of M_R:

$$r.m \Longleftrightarrow m \in M_R \tag{3}$$

In other words, this operator statically ensures that the message m will be 'handled" by the instance bound to r.

The second operator ! is *blind* in the sense that the expression $r!m$ is *always* valid. Of course, we have to guard the application of this operator by a dynamic check, similar to the one for casts mentioned above.[1]

Implementation types can define a *default method* which is triggered for messages that do not have an explicit method associated with them. Inside this default method, messages can be *resent* or *forwarded* to other instances (Figure 3).

Lagoona does not contain any implicit fall back rules for the message dispatch such as inheritance. However, the programmer can easily emulate inheritance, both class-based as well as prototype-object based, using the default method and a simple forwarding statement. We use the term *generic message forwarding* to express that the actual message remains opaque during the forwarding process. The default method is implemented in a generic way and describes the forwarding action for all otherwise unhandled messages for the implementation type. Figure 4 shows a concrete code example where objects of type A forward all unhandled messages to an object of type B. Thus, basically A behaves as it would be derived from B in traditional object-oriented languages.

3. AOP WITH LAGOONA

Adding support for aspect-oriented programming to Lagoona is surprisingly simple. As we have mentioned earlier, instance variables are internal to the associated object in Lagoona and thus objects communicate exclusively using message send operations. This removes the need for *get/set* pointcut functions, which exist in other aspect languages to capture read and write actions to attributes.

[1]For sensible assignment semantics, it is also necessary to perform a dynamic check for the generation of return values in case of messages, which are expected to produce a return value.

```
module Example {
  ...
  class A {
    B b = new B();

    void X() {
      ...
    }

    void default(message m) {
      m.forward(b);
    }
  };

  class B {
    void Y() {
      ...
    }
  };
};
```

Figure 4: Emulating inheritance with stand-alone messages

There is also no need to distinguish between different types of invocations such as *call* and *execution* as all complex dispatch algorithms are explicit in Lagoona. If the programmer decides to use inheritance by emulating it with forwarding, that emulation code is embedded within the default method and can be directly accessed by aspect code.

In AspectLagoona advice code is bound to messages and executed every time that particular message is being send

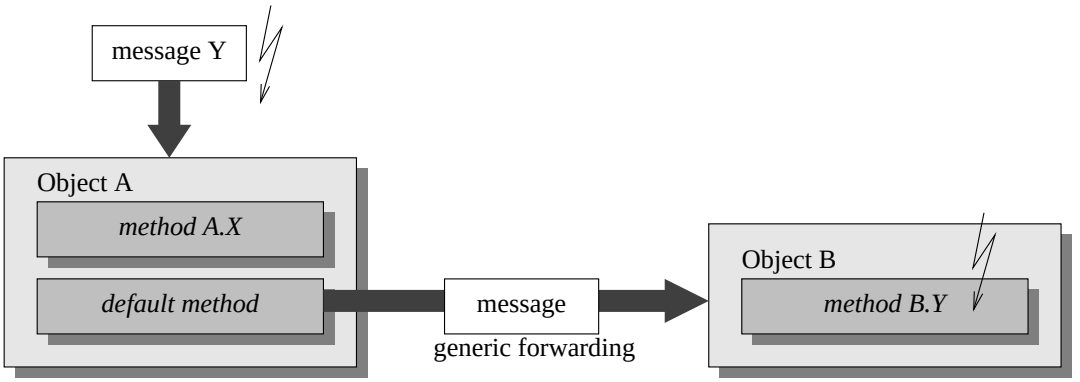

Figure 3: Using generic message forwarding to extend objects

to an object. As messages are unique in Lagoona, there is no need to introduce the concept of named pointcuts to describe and select parts of the class hierarchy. This has a subtle but significant impact on the code maintainability.

Languages like AspectJ and AspectC++ rely on explicit class names or name patterns to select the join points for which advice is being specified. To maintain consistency, these pointcut expression in the aspect code have to be updated every time a new class is added to the system which should receive advice from that particular aspect. Name patterns like "my*" can be used to ease this maintenance burden, allowing to capture all classes or methods having a name matching the pattern. However, this approach is error prone, if the name of a new class accidentally matches already existing name patterns in the program.

In AspectLagoona, aspects are bound to messages and not to concrete implementation types. If advice is specified for a certain method, it will apply to all methods implementing that particular message. When new implementation types are added to the system, the pointcut expression and the aspect definition do not have to be changed, because advice code is specified for messages instead of concrete methods. This feature is vital to apply aspect-oriented programming in a component-based environment, where new components can be added on the fly.

There is also no need to introduce a novel construct for pointcuts and pointcut expressions. A pointcut is a set of join points. In Lagoona the only meaningful join points are messages and Lagoona already supports the notion of a set of messages: *interfaces*. Figure 5 demonstrates how advice code is declared in Lagoona.

In contrast to other aspect languages where privileged aspects can inject code inside components, in Lagoona public interfaces exported by components serve as join points. This decouples the aspect code from the component code and allows the component to hide its internal structure to the degree it is necessary to provide interchangeability. Effectively, interfaces become not only the well-defined specification for inter-component communication, but also for the interaction between aspects and components (Figure 6).

```
module Application {
  interface NeedLocking {
    void print(char c);
    void flush();
  };
  class Screen {
    void print(char c) {
      // lock must be held
      ...
    }
    void flush() {
      // lock must be held
      ...
    }
  };
};

module Locking {
  advice Application.NeedLocking {
    void before() {
      // aquire lock
    }
    void after() {
      // release lock
    }
  }
};
```

Figure 5: Defining advice code in Lagoona

4. RELATED WORK

A number of powerful aspect-oriented languages exists today, including AspectJ [6], AspectC++ [12], and more recently AspectC# [8]. Unfortunately, none of them is geared to accommodate component-oriented programming and aspect-oriented programming at the same time.

AOP has been successfully applied in the operating systems domain by a number of projects. The PURE family of operating systems uses AspectC++ to implement interrupt synchronization for deeply embedded systems [9]. The a-kernel [2] project uses AspectC, a subset of AspectJ to modularize certain crosscutting concerns in the FreeBSD kernel.

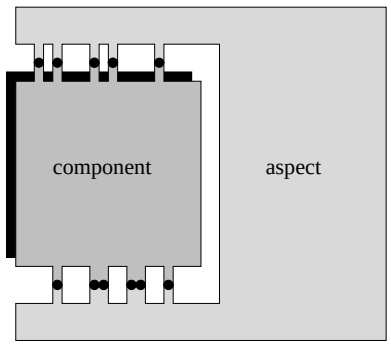

Figure 6: Well-defined aspect/component interface

Both projects operate on a monolithic operating system kernel and do not address component-based systems.

The Aspect-Modulator Framework [11] also permits to modularize C++ OS code with aspects. An aspect modulator is responsible to execute advice code where appropriate. The aspect modulator is invoked from join points generated manually through code insertion. Bossa [1] uses event-based AOP to modularize OS schedulers. Events are inserted manually into the source code and aspects can choose to subscribe to them. An interesting feature of the event-based AOP model is the possibility to dynamically enable and disable aspect code.

5. CONCLUSIONS AND FUTURE WORK

Implementors of system software are reluctant to adopt novel programming mechanisms and paradigms, unless the new technology is handed to them in manageable pieces. AspectLagoona aims exactly at these hard-to-convince users by offering a lightweight mechanism for aspect-oriented programming that still allows to deal effectively with all applications of aspects that the authors of this paper have encountered so far.

On the other hand AspectLagoona does not try to compete directly with AspectJ or AspectC++, as it is in contrast to the latter two aspect languages a complete new language design with a different object-oriented core language. This disqualifies AspectLagoona for the re-engineering of legacy operating systems, which seems to be currently the most pursued approach in pushing AOP into the commercial OS market.

The lesson we are trying to learn from AspectLagoona is what the smallest and least invasive approach to extend a languages with AOP capabilities would be. We are confident that such a minimalistic approach has a good chance of being accepted in certain areas of the system software domain.

Our current implementation of AspectLagoona includes a complete compiler and runtime system, all written in Lagoona. Recently we started to re-engineer parts of the compiler and especially the runtime system to make use of the aspect-oriented features of the language, which were initially not present.

As far as future work is concerned, we plan to rework the advice activation infrastructure, which is currently compile-time driven. Our goal is to fully integrate aspects into the component framework of Lagoona, allowing to compose systems from binary components *and* precompiled aspects at deployment time or even dynamically.

6. REFERENCES

[1] L. P. Barreto, R. Douence, G. Muller, and M. Südholt. Programming os schedulers with domain-specific languages and aspects: New approaches for os kernel engineering. In *Proceedings of the 1st AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, Apr. 2002.

[2] Y. Coady, G. Kiczales, M. Feeley, N. Hutchinson, and J. S. Ong. Structuring operating system aspects: using AOP to improve OS structure modularity, 2001.

[3] M. Franz. The Programming Language Lagoona: A fresh Look at Object-Orientation. *Software - Concepts and Tools*, 18(1):14–26, 1997.

[4] P. Fröhlich and M. Franz. On certain basic properties of component-oriented programming languages. In D. H. Lorenz and V. C. Sreedhar, editors, *Proceedings of the Workshop on Language Mechanisms for Programming Software Components (at OOPSLA)*, pages 15–18, Tampa Bay, FL, Oct. 15 2001. Technical Report NU-CCS-01-06, College of Computer Science, Northeastern University, Boston, MA 02115.

[5] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 2nd edition, 2000.

[6] 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.

[7] 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.

[8] H. Kim. AspectC#: An AOSD implementation for C#. Master's thesis, Department of Computer Science, Trinity College Dublin, Sept. 2002.

[9] D. Mahrenholz, O. Spinczyk, A. Gal, and W. Schrder-Preikschat. An aspect-orientied implementation of interrupt synchronization in the pure operating system family. In *Proceedings of the 5th ECOOP Workshop on Object-Orientation and Operating Systems*, Malaga, Spain, June 2002.

[10] Microsoft Corporation. *The Component Object Model (Version 0.9)*, Oct. 1995.

[11] P. Netinant, T. Elrad, and M. E. Fayad. A layered approach to building open aspect-oriented systems: a framework for the design of on-demand system demodularization. *Communications of the ACM*, 44(10):83–85, 2001.

[12] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An Aspect-Oriented Extension to the C++ Programming Language. In *Fortieth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific)*, volume 10 of *Conferences in Research and Practice in Information Technology*. ACS, 2002.

[13] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley / ACM, 1998.