

Towards Language Support for Component-Oriented Real-Time Programming (Position Paper)

Michael Franz
franz@uci.edu

Peter H. Fröhlich
phf@acm.org

Thomas Kistler*
kistler@ics.uci.edu

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA

Abstract

Component-oriented programming promises to finally make the vision of pervasive software components a reality. In the area of dependable real-time systems, the benefits of increased reuse, reliability, and efficiency make component-oriented programming especially attractive. As part of the Lagoon project, we are investigating how component-oriented programming can be supported at the level of programming languages. Recently, we have also become interested in supporting the construction of real-time systems within the Lagoon framework, and we have started exploring the language design space for this domain. We outline the design of the experimental programming language Lagoon and discuss two of its novel features in more detail. We also discuss the difficulties we see for integrating the ideas of component-oriented programming with the requirements of real-time systems. It is our hope that feedback from the workshop will enable us to more clearly define a research agenda in this area.

1 Introduction

The vision of pervasive software components has guided software engineering research for over three decades [8]. Currently, the most promising approach for making this vision a reality is the idea of component-oriented programming [16]. Component-oriented programming aims to replace traditional monolithic software systems with reusable *software components* and layered *component frameworks*. Components extend the capabilities of frameworks, while frameworks provide an execution environment for compo-

nents. Both can be developed by independent and mutually unaware vendors, and their composition into a running system be performed by a third party.

A component-based approach to software development promises many advantages, almost all of which result from the possibility of vendors specializing in a single domain of expertise. Development resources can thus be concentrated on a single component (or a single framework), hopefully increasing its reliability and efficiency beyond what would be possible otherwise. These qualities make component-oriented programming especially attractive for dependable real-time and embedded systems, where adherence to strict quality standards is of paramount importance.

However, the very nature of real-time systems complicates the composition of independently developed components. Apart from functional requirements, which have been expressed successfully in the past, certain non-functional requirements—most obviously related to execution time constraints—have to be taken into account. Besides the goal of making functional and non-functional requirements orthogonal as far as possible, there seems to be little agreement on how non-functional requirements should be modelled and enforced [7, 14].

We have been investigating how to support the component-oriented paradigm directly at the programming language level. The experimental programming language Lagoon¹ achieves this goal better than other currently popular languages such as Java [1] and C++ [15]. Recently, we have also become interested in supporting the construction of dependable real-time systems within the Lagoon framework, and we have started exploring the language design space for this domain.

*Current address: Transmeta Corporation, 3940 Freedom Circle, Santa Clara, CA 95054, USA, kistler@transmeta.com

¹An earlier version of Lagoon, from which the one described here has evolved over the last two years, was described in [5].

Definite results are still pending. We have identified several problems that seem to preclude a straightforward extension of the *Lagoona* model (and more generally component-oriented programming) to real-time systems. Nevertheless, some of *Lagoona*'s mechanisms can be used to support scheduling of *non-essential* computations within a real-time system. It is our hope that feedback from the workshop will help us in understanding this paradox better and maybe enable us to more clearly define a research agenda in this area.

The rest of this paper is organized as follows. Section 2 introduces the experimental programming language *Lagoona*, focusing mainly on its novel object model. Section 3 outlines the problems we see for integrating the ideas of component-oriented programming (and hence *Lagoona*) with the requirements of real-time systems and shows how *Lagoona*'s flexible message-send semantics could be exploited to schedule non-essential computations.

2 The Programming Language *Lagoona*

Lagoona is an imperative, modular, and object-oriented programming language loosely based on Oberon [13]. Note that we use the term "object-oriented" somewhat reluctantly because the better term "component-oriented" is not yet widely accepted. As will become clear below, *Lagoona* differs fundamentally from common object-oriented languages, for example in the treatment of messages and the absence of inheritance.

The basic compilation unit in *Lagoona* is the `MODULE` as opposed to the class or interface in conventional object-oriented languages. Modules contain declarations of constants, variables, types, procedures, messages, and methods. The core of *Lagoona* is a conventional imperative programming language. Basic data types include `INTEGER`, `REAL`, `BOOLEAN`, `CHAR`, and `SET`. Structured data types include `ARRAY` and `RECORD`. Apart from assignment commands, *Lagoona* supports the usual control structures such as `IF`, `CASE`, `WHILE`, `REPEAT`, `FOR`, and arithmetic expressions. Sequences of commands or arithmetic expressions can be abstracted using a standard `PROCEDURE` mechanism. However, unlike in Oberon, parameter passing modes are expressed abstractly as `IN`, `OUT`, or `INOUT`, similar to Ada [12] and the COM Interface Definition Language (IDL) [2, 4]. This allows describing the intended data-flow across a parameter explicitly, but without committing to a certain implementation of parameter passing.

While the imperative core of *Lagoona* is conservative, its object model differs considerably from conventional object-oriented languages. The fundamental concept "borrowed" from the object-oriented paradigm is a form of polymorphism: it must be possible to dynamically use different implementations (component instances) through one and the same static interface. In C++ or Java, this would be mod-

elled by inheritance from an abstract base class or through the interface mechanism. However, these approaches have fundamental limitations in a component-based setting. For example, interfaces and implementations must be statically bound to each other (by name) to ensure compatibility [3].

Lagoona distinguishes explicitly between *messages* that define protocols of interaction, and *methods* that define particular implementations of such a protocol. Messages are "stand-alone" in the sense that they are not subordinate to a specific language construct such as interfaces or classes in Java and C++. Instead, messages are declared at the module level. As we have discussed elsewhere [6], this is in part motivated by the observation that messages have to be "anchored" in some static, unchanging place to retain a unique identity. If they are subordinate to entities such as interfaces or classes, both of which are usually *extensible* through subtyping and subclassing, syntactic and semantic interface incompatibilities can arise.

To define more complex protocols of interaction, *Lagoona* provides *message set types* to group arbitrary sets of messages. For example, given messages A, B, C, and D, the type $S = \{A, B, C\}$ represents such a group. Note that a message set type only provides a convenient *name* for a set of messages, but messages are *not* subordinate to it. New message set types can be constructed from existing ones through "set-like" *union* and *difference* operations. For example, the type above could also be expressed as $S = \{A\} + \{B, C\} - \{D\}$. Compatibility between two message set types is defined using a subset relation between the sets of messages they denote. Given the type $T = \{A, B, C, D\}$ and variables $s : S$ and $t : T$, the assignment $s := t$ is legal, while the reverse assignment $t := s$ is not. The obvious consequence of this definition is that type compatibility between message set types is *structural*. Nevertheless, this structural compatibility is *safe* in *Lagoona* because individual messages have and retain unique identities, unlike in languages such as Java and C++. Furthermore, the combination of set-like operations and structural compatibility enables the expression of *subtyping* as well as *supertyping* in a straightforward way. Hence variables and formal parameters can be typed "minimally" to exactly specify the relevant interaction protocol. For example, a method X that only applies the D message to some object can be declared as `X(INOUT object : {D})`; instead of having to mandate an implementation of the complete T message set.

Message set types can be used to reference *objects* that implement the messages denoted by the type in the form of *methods*. In *Lagoona*, an *object type* is basically a record type with attached methods (procedures). This idea is fairly standard, except for the fact that object types do *not* explicitly declare which message sets they implement. Rather, the structural type compatibility mechanism introduced be-

tween message set types is extended to object types. A variable of object type Z can be assigned to a variable of message set type S if and only if Z implements (has methods for) all messages denoted by S . However, between two object types, name-based compatibility is used, since record fields are still subordinate to their type and thus are not unique.

Compatibility between message set types and object types is checked when they are actually used in assignments or as actual parameters. This supports the evolution of software components better than static declarations do, at the price of detecting certain type-errors “slightly” later. Note that this is also important when a component implements messages that might not even be known to the rest of the composed system. Furthermore, a method can implement multiple messages of compatible signature if the developer of the method decides that they are semantically compatible. Hence duplication of method bodies can be avoided and no additional forwarding methods have to be introduced.

Instead of inheritance (in the sense of reuse of field or method declarations in a subclass), *Lagoona* provides a *generic message forwarding* mechanism. Avoiding inheritance (and delegation) helps to guard against certain semantic problems caused by the self-referential nature of this mechanism [9]. It has been shown that restricting inheritance (and delegation) for safety purposes in a component-based setting essentially makes inheritance equivalent to forwarding [10, 17]. Therefore we rely on forwarding from the outset, which also provides for more uniform software architectures (see below).

The generic forwarding mechanism works as follows. When an object receives a message, and a corresponding method implementing this message exists in the object type, that method is executed. If no matching method is found, but the special method `DEFAULT` is implemented, it is executed instead. Inside a *default method* we can *resend* the message to other objects. However, resending is a generic operation in which the actual message remains opaque. Note that an empty default method can be used to ignore all messages an object does not explicitly implement. Furthermore, the forwarding mechanism can be used to emulate the semantics of both static and dynamic inheritance if the original receiver is passed as an explicit parameter. If neither a matching method nor a default method exists, execution is aborted with an exception.

For this flexible message forwarding approach to be safe, some restrictions on message-sends have to be imposed. In particular, for messages that return a result, we must be able to statically determine whether they are handled or not. Otherwise using the result of the message-send would be unsound. In *Lagoona*, we decided to offer two clear alternatives to the programmer. If a message is sent to a message set variable, it must statically exist in the set of messages

denoted by the corresponding message set type. Note that this rule applies to all messages, regardless whether they return a result. If a message is sent to an object type variable, and if the message returns a result, an implementation for it must statically exist in the object type. If it does not return a result, the message might be handled (possibly after cascaded forwarding), it might be ignored, or an exception might be raised. Hence developers can decide on the exact message-sending semantics they need at a fine-grained level.

From an architectural perspective, the availability of source-language support for fine-grained message forwarding (at the level of individual objects) makes it possible to structure software systems in a particularly straightforward manner. In most cases, the resulting systems are less complex than corresponding object-oriented systems because all control flow is explicit. In *Lagoona*, code-reuse that in object-oriented systems would be achieved by inheriting from a class that already has the desired behavior is instead realized by forwarding messages to an object that performs the operation on behalf of the original receiver. The implementation challenge is to remove the overhead of this (possibly cascaded) dispatch and make it no more costly than a virtual function call in an object-oriented language.

Programming in *Lagoona* has certain similarities with object-oriented programming, but also important differences. The central activity in object-oriented program development is typically that of incrementally modifying a given application framework through subclassing and method overriding. In contrast to this “incremental modification” paradigm, system-building in *Lagoona* is strictly compositional: a program is assembled using pre-existing objects and newly created objects by connecting them using appropriate “pipes.” Hence, *Lagoona* takes the overall system architecture already in widespread use at the macroscopic level in component-oriented systems and pushes it down to the microscopic level of individual objects, yielding a structural uniformity across all levels of abstraction. We contend that message forwarding is a “natural” paradigm for component-based systems since it uniformly applies to intra-component messaging as well as inter-component communication.

3 Towards a Real-Time Variant of Lagoona

The first obvious step for extending *Lagoona* with support for dependable real-time systems would be the addition of a tasking model and suitable synchronization primitives. We do not believe that this extension poses any particular problems, except maybe for the parametrization of tasks according to various relevant scheduling and behavioral policies. In what follows, we assume that some form of tasking model is already available.

The addition of real-time constraints seems to be a much more difficult problem. A central issue is which language constructs should be annotated with such constraints, and in what form the annotations should be expressed. Unfortunately, several previously feasible approaches to real-time systems are fundamentally incompatible with the component-oriented paradigm. This stems from the fact that the latter specify *functional* requirements (i.e., interfaces and their semantics), whereas the former additionally specify *non-functional* constraints on the *behavior* of applications (i.e., component interactions as well as their timing constraints).

As an example, individual methods can not sensibly be annotated with timing constraints such as an upper bound associated with their execution. This is because timing constraints are usually *global* properties of an application and not local to an individual component. One system might impose an end-to-end deadline for a sequence of message-sends, requiring a given method to perform its task in 10ms. However, in another system using different call chains and a different end-to-end deadline, the same method might be required to perform its task in only 5ms. Constraining components on a per method basis therefore inherently limits the reuse of a given component across different constraint systems. Annotating *messages* (and therefore interfaces) might seem to work better. However, if functionally equivalent messages specify different timebounds as parts of their semantics, the number of functionally equivalent but still incompatible interfaces would explode, again affecting the potential for reuse negatively.

Instead of specifying timing constraints on a per method or per message basis, the specification of real-time constraints could also be separated from the specification of the functional behavior (e.g., using a special real-time synchronizers language [11]). However, this scheme is also partially incompatible with the component-oriented paradigm. Not only does it not *scale* because the specification of constraints grows linearly with the size of applications, but specifying a real-time component also requires the full understanding of the component implementation and its interaction schemes, which contradicts the notion of black-box reuse. In light of these facts, it is still an open question how timing constraints can best be specified at the programming language level to facilitate component reuse across different systems and platforms.

However, the flexible message-send semantics of Lagoon could potentially be used to the benefit of a real-time system in the area of *non-essential* computations. One example of a non-essential computation can be found in algorithms that are iterative and produce successively refined results. In such cases, upper bounds for timing constraints are extremely difficult to find because they might intricately depend on dynamic input data. Nevertheless, since the com-

putation is iterative it might be feasible to terminate the computation early and return the last consistent result as soon as the timing constraints are approaching.

In Lagoon, such a problem can be solved by using a guaranteed message-send for essential computations while successive refinements can be obtained through an optional message-send. The idea of optional message-sends is that they can be skipped automatically as the deadline approaches. As described in Section 2 above, message-sends can be ignored if (a) the message does not return a result, (b) the message is sent to a variable of an object type, and (c) no implementation for the message is statically visible in the object type.² By consciously modeling the system in a way that (a–c) hold for all non-essential computations, we can give the scheduler more flexibility to achieve a deadline by ignoring these message sends completely.

Another example for non-essential computations are debugging and logging operations. These operations are often removed from a production version of the system in order to increase its efficiency. However, in the case of a failure it could still be useful to have as much information as possible logged.

Acknowledgements

We would like to thank Ziemowit Laski and Christian Stork for valuable comments on an earlier version of this paper. We are also grateful to the participants of the Lagoon mailing list. This work was partially supported by the National Science Foundation under grant EIA-9975053.

References

- [1] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, second edition, 1998.
- [2] D. Box. *Essential COM*. Addison-Wesley, 1998.
- [3] M. Büchi and W. Weck. Compound types for Java. In A. M. Berman, editor, *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'98)*, pages 362–373, Oct. 1998. Published as SIGPLAN Notices 33(10), October 1998.
- [4] M. Corporation. *The Component Object Model (Version 0.9)*, Oct. 1995. Available at <http://www.microsoft.com/COM/resources/COM1598C.ZIP>.
- [5] M. Franz. The programming language Lagoon: A fresh look at object-orientation. *Software — Concepts and Tools*, 18(1):14–26, Mar. 1997.
- [6] P. H. Fröhlich and M. Franz. Component-oriented programming in object-oriented languages. Technical Report 99-49, Department of Information and Computer Science, University of California, Irvine, Oct. 28, 1999. Revised Dec. 11, 1999. Available at <http://www.peter.froehlich.com/>.

²Alternatively a second, *non-binding* message-send operator could be introduced to make this behavior more explicit.

- [7] M. Lycett and R. J. Paul. Component-based development: Dealing with non-functional aspects of architecture. In Weck et al. [19].
- [8] M. D. McIlroy. Mass produced software components. In *Proceedings of the Nato Software Engineering Conference, Garmisch, Germany*, pages 138–155, Oct. 1969. Available at <http://www.cs.dartmouth.edu/~doug/components.txt>.
- [9] L. Mikhajlov and E. Sekerinski. The fragile base class problem and its solution. Technical Report 117, Turku Center for Computer Science, Lemminkäisenkatu 14, FIN-20520 Turku, Finland, June 1997.
- [10] L. Mikhajlov, E. Sekerinski, and L. Linas. Developing components in the presence of re-entrance. Technical Report 239, Turku Center for Computer Science, Lemminkäisenkatu 14, FIN-20520 Turku, Finland, Feb. 1999.
- [11] B. Nielsen and G. Agha. Towards re-usable real-time objects. Technical Report RS-98-44, BRICS, Department of Computer Science, University of Aarhus, Ny Munkegade, building 540, DK-8000 Aarhus C, Denmark, Dec. 1998. Available at <http://www.brics.dk/RS/98/44/index.html>.
- [12] I. S. Organization. *(ISO/IEC 8652:1995): Information Technology — Programming Languages — Ada*. 1995.
- [13] M. Reiser and N. Wirth. *Programming in Oberon: Steps beyond Pascal and Modula*. Addison-Wesley, 1992.
- [14] B. Robben, F. Matthijs, W. Joosen, B. Vanhaute, and P. Verbaeten. Components for non-functional requirements. In Weck et al. [19].
- [15] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 1997.
- [16] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.
- [17] W. Weck. Inheritance using contracts and object composition. In Weck et al. [18], pages 105–112.
- [18] W. Weck, J. Bosch, and C. Szyperski, editors. *Proceedings of the Second International Workshop on Component-Oriented Programming (WCOP'97)*, number 5 in TUCS General Publications, Turku Center for Computer Science, Lemminkäisenkatu 14, FIN-20520 Turku, Finland, Sept. 1997.
- [19] W. Weck, J. Bosch, and C. Szyperski, editors. *Proceedings of the Third International Workshop on Component-Oriented Programming (WCOP'98)*, number 10 in TUCS General Publications, Turku Center for Computer Science, Lemminkäisenkatu 14, FIN-20520 Turku, Finland, Oct. 1998.