

# On Reconciling Objects, Components, and Efficiency in Programming Languages

Peter H. Fröhlich      Andreas Gal      Michael Franz  
phf@acm.org          gal@uci.edu          franz@uci.edu

Technical Report No. 02-12  
Department of Information and Computer Science  
University of California, Irvine  
Irvine, CA 92697-3425, USA

March 22, 2002  
Revised: March 27, 2002

## Abstract

The paradigm of component-oriented programming, which promises to improve upon object-oriented techniques in significant ways, is currently still difficult to apply in practice. One problem impeding further progress is that established strongly-checked, class-based, object-oriented languages do not take the requirements of component-oriented programming into account. The concepts of stand-alone messages and generic message forwarding, which have been developed in the programming language *Lagoona*, lead to language designs that suit the new paradigm better while still preserving an object-oriented programming style. We illustrate the utility of these concepts by analyzing several archetypal design and implementation problems and solving them in a coherent language framework. We also discuss how these concepts can be implemented efficiently and propose a suitable execution model based on high-level intermediate representations and dynamic recompilation.

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Lagoona</b>	<b>4</b>
2.1	Object Model . . . . .	5
2.2	Modules . . . . .	7
2.3	Messages . . . . .	8
2.4	Interface Types . . . . .	8
2.5	Implementation Types . . . . .	10
<b>3</b>	<b>Applications</b>	<b>10</b>
3.1	Interface Combination and Conformance . . . . .	12
3.2	Fragile Base Classes . . . . .	13
3.3	Component Reentrance . . . . .	13
3.4	Iterators . . . . .	15
<b>4</b>	<b>Implementation</b>	<b>16</b>
4.1	Static Compilation . . . . .	21
4.2	Dynamic Compilation . . . . .	23
<b>5</b>	<b>Related Work</b>	<b>25</b>
5.1	Language Mechanisms . . . . .	25
5.2	Component Models . . . . .	27
5.3	Generative Programming . . . . .	27
5.4	Dynamic Compilation . . . . .	28
<b>6</b>	<b>Future Work</b>	<b>28</b>
<b>7</b>	<b>Conclusions</b>	<b>29</b>

# 1 Introduction

Today’s software projects reach dimensions that would have been unthinkable in the days in which the idea of object-oriented programming started to emerge. The OpenOffice project [39], for example, is expected to cross the 10 million lines of code boundary within the next few months. In systems of such complexity, it is next to impossible for a single human being to understand the existing functionality well enough to identify potentials for reuse through mechanisms like inheritance. Worse than that, since projects of this size usually involve hundreds of independent developers, it is unlikely that all of them understand the system well enough to evolve it in a coherent fashion. Software development costs increase as well, making it uneconomical for a single vendor to develop all parts of a system from scratch. These developments explain the widely recognized need to improve upon the current practice of object-oriented programming, although how this can be done is much less obvious.

One recent proposal for such an improvement is the idea of *component-oriented programming* [41]. While various forms of “components” have been proposed in the past—starting in the late 1960s [27]—this latest incarnation focuses on the *distributed extensibility* of software. A software system is extensible in a distributed fashion if extensions can be (a) developed by anyone and (b) integrated at any time.<sup>1</sup> In contrast to the “standard” notion of extensibility, this rules out a central authority with global control over the extensions made to a system. Indeed, we can not even speak of “*the system*” anymore: we need to abandon the traditional “white box” view of software in which extensions are integrated by a global recompilation step. By necessity, component-oriented programming encourages clearly and rigorously defined boundaries between components (which provide extensions) and frameworks (which control their interaction). This makes component-oriented designs easier to comprehend than object-oriented ones and also improves the potential for reuse of frameworks and components.

While a variety of approaches to component-oriented programming have been suggested at this point, none of them has achieved a breakthrough yet. One of the problems impeding our progress is that conventional strongly-checked, class-based, object-oriented programming languages like Eiffel [28] or Java [18] do not fit the requirements of component-oriented programming well. The idea of distributed extensibility has profound implications for pro-

---

<sup>1</sup>These requirements are also known as (a) *independent extensibility* and (b) *dynamic extensibility* [41].

programming languages, but it was simply not considered when these languages were designed. Eiffel’s covariant argument types, for example, require a form of global analysis that can not be performed satisfactorily at either compile-time or load-time: At compile-time we do not know “the whole system,” and at load-time the user who “integrates the system” neither has the knowledge nor the means to fix a type-error. Another less obvious but just as pressing example are “name clashes” between frameworks and components.

In this paper, we report on the design and implementation of the experimental programming language *Lagoona*, which was developed with distributed extensibility in mind. We focus on two novel language mechanisms, namely *stand-alone messages* and *generic message forwarding*, and show how they allow us to eliminate or alleviate a number of (sometimes long-standing) design and implementation problems in object-oriented and component-oriented programming. Among others, we discuss the issues of *interface conflicts*, *fragile base classes*, and *component reentrance*. Since *Lagoona* differs from other strongly-checked, class-based, object-oriented languages to a significant degree, we also explore the implications of this design for language implementation at the compiler and runtime support level. In particular, we propose an efficient execution model for static compilation settings as well as an aggressively optimizing execution model based on dynamic recompilation.

The remainder of this paper is organized as follows. Section 2 introduces the basics of *Lagoona*, including the concepts of stand-alone messages and generic message forwarding. In Section 3 we discuss a number of well-known design and implementation problems in object-oriented and component-oriented programming and show how *Lagoona* helps to solve them. Section 4 describes our implementation and shows how we address the major challenges for efficient execution. In Section 5 we compare *Lagoona* to related work in various areas. Section 6 describes several ideas for future work and Section 7 summarizes our discussion and offers conclusions.

## 2 Lagoona

In this section we introduce the basics of *Lagoona* and give examples to acquaint the reader with its programming style. We focus on *Lagoona*’s novel aspects and do not describe the (fairly standard) imperative core language.

<b>Concern</b>	<b>Traditional</b>	<b>Lagoona</b>
Encapsulation	Class (modifiers)	Module
Specification	Class (abstract method)	Message
	Class (abstract)	Interface Type
Implementation	Class (concrete method)	Method
	Class (concrete)	Implementation Type
Modification	Class (inheritance)	Forwarding

Table 1: Design concerns and corresponding language constructs in traditional languages and in Lagoona.

## 2.1 Object Model

The foundation of Lagoona [12, 14, 15] is an object model that separates many of the roles traditionally played by classes, turning them into individual language constructs. Table 1 provides a compact comparison how different design concerns are mapped onto language constructs in traditional object-oriented languages and in Lagoona.

At the lowest level of Lagoona’s object model are *messages* and *methods*. Messages are *abstract operations* that describe *what* effect they achieve, while methods are *concrete operations* that describe *how* a certain effect is achieved. In other words, messages are specifications for methods, and methods are implementations of messages. At the next higher level, messages and methods are grouped into *interface types* and *implementation types*. An interface type is simply a set of messages, while an implementation type consists of a set of methods and associated storage definitions. Variables of these types are called *interface references* and *implementation references* respectively. Implementation types serve as generators for *instances*, which are first-class values that can be assigned to implementation or interface references. As with messages and methods, interface types and implementation types serve as specifications and implementations for each other. At the highest level of the object model are *modules* which encapsulate sets of messages, methods, interface types, and implementation types. Modules are *unique* in the sense that only a single copy of a certain module can exist in a given system.

So far, our description of Lagoona’s object model reads almost like the textbook definition of any object-oriented programming language. What sets Lagoona apart are the following additional relations between the concepts introduced above. Although messages are “grouped into” interface

types, they are not declared in the scope of a type but rather in the scope of a module. Since modules are unique, this implies that messages are unique as well. We use the term *stand-alone messages* to express this independence of messages from types. 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 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  $M_B$ :

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

We extend structural subtyping to implementation types, and if (1) and (2) hold,  $A \trianglelefteq C$  will hold as well. Furthermore, this enables a form of *inclusion polymorphism* that we like to call *implementation polymorphism*.

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.

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

This completes the definition of conformance, but the fourth case raises the question how implementation types can be reused or adapted. 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. The first operator  $\rightarrow$  is *strict* in the sense that the expression  $m \rightarrow r$  is valid if and only if  $m$  is an element of  $M_R$ :

$$m \rightarrow r \iff m \in M_R \tag{3}$$

In other words, this operator statically ensures that the message  $m$  will be “handled” by the instance bound to  $r$ . The second operator  $\Rightarrow$  is *blind* in the sense that the expression  $m \Rightarrow r$  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.<sup>2</sup> The blind message send operator is necessary to support reuse and adaptation by intercepting and rerouting messages. 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. We use the term *generic message forwarding* to express that the actual message remains opaque during this process. Obviously, the strict message send operator alone would not be sufficient to support this.

Lagoona’s object model can be viewed as another step towards eliminating the dominance of the class construct in object-oriented languages. Previous steps include the separation of interfaces and implementations [36] and the separation of modules and types [40], both of which are widely accepted by now. In the remainder of this section we explain each element of Lagoona’s object model in more detail. We also discuss how these elements are mapped into the actual programming language using several concrete examples.

## 2.2 Modules

Lagoona’s top-level language construct is the *module*, which serves a variety of purposes. Modules are compilation units and result in object files which in turn are the units of deployment [41]. Modules live in a flat, global namespace and cannot be nested. However, we employ a “hierarchical” naming convention based on Internet domain names, similar to the one originally proposed for Java packages [18]. Modules are *sealed* in the sense of [4]; only explicitly exported declarations are visible to clients, and no new declarations can be added from the outside. Modules can *import* other modules and then refer to their exported declarations. These references are fully qualified, but to avoid “excessive” qualifications we allow the introduction of local *aliases* for imported modules.

The module shown in Figure 1 exports all its declarations by marking them **public**. The module in Figure 2 imports the first one under the alias **S** and uses this alias to qualify further references, for example to the

---

<sup>2</sup>For sensible assignment semantics, it is also necessary to restrict  $\Rightarrow$  to messages that do not return a result.

```

module com.lagoona.papers.tr0212.stacks {
  public message void push(any obj);
  public message void pop();
  public message any top();
  public message boolean empty();
  public interface Stack {push, pop, top, empty}
}

```

Figure 1: The infamous stack abstraction in Lagoona. Messages are bound to modules, not types.

message `push`. However, several declarations inside the second module are not marked `public` and are therefore hidden from its clients.

### 2.3 Messages

One feature that sets Lagoona apart from established object-oriented programming languages is *stand-alone messages*. As shown in Figure 1, messages are bound to (declared in) modules instead of types. Since modules are unique within a given system, and since no two messages can have the same name within a given module, our approach makes messages unique as well. If messages were bound to types, the approach taken in most conventional object-oriented languages, we could not guarantee this property in general. Surprisingly, many of the applications described in Section 3 stem from this seemingly trivial difference. We usually associate a semi-formal specification with each message. The `push` message, for example, would be characterized with the precondition “`obj ≠ null`” and the postcondition “`¬empty`”. Finally, we assume that a message and its specification are *immutable* once published, which is similar to the assumption made about interfaces in COM [29] and related technologies.

### 2.4 Interface Types

Messages are the basis for *interface types* (`interface` in our concrete syntax) which represent references to objects that implement a certain set of messages. In Figure 1, the interface type `Stack` is declared as supporting the messages `push`, `pop`, `top`, and `empty`. If we declare a variable `s` of type `Stack`, we can only assign objects that implement at least these four operations to `s`. As explained in Section 2.1, conformance to interface types

```

module com.lagoona.papers.tr0212.simple_stacks {
  import S = com.lagoona.papers.tr0212.stacks;
  class Link {
    any object; Link next;
  }
  public class Stack {
    Link top;
    method void initialize() {
      this.top = null;
    }
    method void S.push(any obj) {
      Link x = new Link(); x.object = obj;
      x.next = this.top; this.top = x;
    }
    method void S.pop() {
      this.top = this.top.next;
    }
    method any S.top() {
      return this.top.object;
    }
    method boolean S.empty() {
      return this.top == null;
    }
  }
}

```

Figure 2: An implementation of the stack abstraction. Methods implementing messages are bound to 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 Lagoon are used to decouple independent components, similar to the use of interfaces in both COM [29] and to a certain extent Java [18].

## 2.5 Implementation Types

Implementation types (`class` in our concrete syntax) host methods and declarations for instance variables. Consider the implementation of the `Stack` abstraction shown in Figure 2. Each method implements exactly one message imported from the module `S`. The message `initialize` (and also `finalize`) has a special meaning in Lagoon: it is sent by the runtime system immediately after an instance has been created (or, in the case of `finalize`, right before it is garbage collected). The class `Link` is essentially used as a simple record type without any methods.

Figure 3 illustrates how message forwarding between instances is used to “extend” an existing implementation type. In this example, we want to extend the stack abstraction (and its implementation) with an operation that determines the number of elements currently on the stack. First we introduce a new message `elements` which does exactly that. Next we declare a class `Stack` that has an interface reference to another stack and an instance variable for the actual counter. The method `elements` simply returns the counter value. The methods `S.push` and `S.pop` update the counter and forward their messages to the “basic” stack instance. Although not directly related to the extension we want to produce, we also have to implement the messages `S.top` and `S.empty`. The reason is that both of these messages return a value and can therefore not be handled by the generic message forwarding mechanism implemented in the `default` method. However, implementing the `default` method as shown allows this extension to be composed with other, unrelated extensions.

## 3 Applications

In this section, we illustrate how stand-alone messages and generic message forwarding address a number of recurring design and implementation problems in both object-oriented programming and component-oriented programming.

```

module com.lagoona.papers.tr0212.counting_stacks {
  import S = com.lagoona.papers.tr0212.stacks;
  public message int elements();
  public class Stack {
    S.Stack stack;
    int count;
    method void initialize(S.Stack stack) {
      this.stack = stack; this.count = 0;
    }
    method int elements() {
      return this.count;
    }
    method void S.push(any obj) {
      this.count++; S.push(obj) -> this.stack;
    }
    method void S.pop() {
      this.count--; S.pop() -> this.stack;
    }
    method any S.top() {
      return S.top() -> this.stack;
    }
    method boolean S.empty() {
      return this.count == 0;
    }
    method void default() {
      current => this.stack;
    }
  }
}

```

Figure 3: Adding counting to the stack abstraction and its implementation.

### 3.1 Interface Combination and Conformance

Component-oriented programming often requires *combining* multiple interface types that were defined independently, for example if they are to be implemented by a single implementation type. Furthermore, since these combined interface types can again be defined independently, the *conformance* between interface types needs to fulfill certain requirements as well.

Interface combination itself is already a problem in conventional object-oriented programming languages, since it can lead to *syntactic* and *semantic* conflicts [14]. Often, these conflicts are referred to as “name clashes,” and the problem is considered to be solved by providing language mechanisms to work around it. For example, Java [18] supports overloading of method names, which can be used to avoid a subset of these conflicts. More general solutions are provided in Eiffel [28], which supports renaming of methods in descendant classes, and in C++ [10], which supports a form of explicit qualification of methods. However, these techniques fail to take distributed extensibility into account, because they are only applied to fix a “name clash” once it occurred. In Lagoon, both kinds of conflicts are ruled out *by design* since messages always have a unique identity. Interface combination in Lagoon thus has the following two properties: (a) any combination of interface types results in an interface type (no syntactic conflicts), and (b) any combination of interface types preserves all constituent messages (no semantic conflicts). Solving this (long-standing) problem in fact motivated the design of stand-alone messages to a certain degree.

While the problem of interface combination has been known for a long time, the related problem of interface conformance has received attention only recently. Consider two interface types **A** and **B** that have been defined independently by vendors *A* and *B*. Vendors *C* and *D* define—again independently—combinations of **A** and **B**, for example **C** = **A** + **B** and **D** = **B** + **A**. While both **C** and **D** support exactly the same messages, they do not necessarily conform to each other. Most object-oriented languages rely on a declared form of conformance, i.e. types are equivalent *by name* (or *by occurrence*) instead of *by structure* (or *by extent*). The usual objection to structural conformance is that it can lead to “accidental” conformance relationships, with the archetypal example being a **Cowboy** and a **Shape** both understanding a message **draw** with different semantics. Lagoon’s stand-alone messages provide a solution to this problem as well, as we can support structural conformance between interface types without the potential for accidental conformance. Recent proposals to extend Java with a form of structural conformance [3, 24] result in a more complicated and less flexible design.

## 3.2 Fragile Base Classes

The concept of *inheritance* was once hailed as the “golden way” towards extensible software systems. However, the mechanism is generally not suitable for achieving distributed extensibility. Assume a container class **A** that supports operations **Add** for adding an element, **Rem** for removing an element, as well as **MulRem** for removing *several* elements at once. We want to define a derived class **B** that also supports queries about the number of elements currently in the container. However, **B** cannot be implemented without knowing the implementation details of **A** as well: On the one hand, if the developer of **A** implements **MulRem** by calling **Rem** repeatedly, **Rem** (and only **Rem**!) must be overridden to maintain an accurate count. On the other hand, if **MulRem** does not call **Rem**, we have to override **MulRem** as well as **Rem**. This is known as the *fragile base class problem*, and it can be resolved by following an elaborate set of design conventions [30]. If we want to avoid it altogether, we have to restrict the use of inheritance or abolish the mechanism completely.

In *Lagoona*, generic message forwarding takes the place traditionally filled by inheritance. It is easy to see how to solve the example problem using this mechanism. Instead of deriving a new class **B**, we develop an implementation type **B** that has a reference to an instance of **A**. We implement the methods corresponding to the messages **Add**, **Rem**, and **MulRem** by first maintaining our count and then sending the message to the **A** instance. We also implement a default method to forward all other messages to **A**.

## 3.3 Component Reentrance

When we use messages and interface types to specify the functionality of certain instances, we often make the assumption that each operation executes *atomically*. However, for certain design patterns that rely on “callbacks” between instances this is not the case, leading to the *component reentrance* problem [31, 41].

Consider the *Observer* (or *Publish-Subscribe*) design pattern [17] for example, which is used to achieve loose coupling between objects by implicit invocation. A publisher encapsulates some kind of data that is of interest to subscribers. When this data changes, the publisher automatically notifies all its current subscribers. Figure 4 illustrates how this design pattern could be modeled in Java using two interfaces **Publisher** and **Subscriber**. Subscribers **attach** themselves to a publisher, and whenever **set** is invoked, the publisher in turn invokes **update** on all registered subscribers. Subscribers then use **get** to retrieve the current state of the publisher and update them-

```

package org.bloat.papers.tr0212.pubsub ;
public interface Publisher {
    public void attach(Subscriber me);
    public void detach(Subscriber me);
    public Object get();
    public void set(Object data);
}
public interface Subscriber {
    public void update(Publisher from);
}

```

Figure 4: Naive publishers and subscribers in Java.

```

module com.lagoona.papers.tr0212.pubsub {
    public interface Publisher {attach, detach, get, set}
    public message void attach(Subscriber me);
    public message void detach(Subscriber me);
    public message any get();
    public message void set(any data);
    public interface Subscriber {update}
    public message void update(interface {get} from);
}

```

Figure 5: Smarter publishers and subscribers in Lagoona, only `get` can be sent within `update`.

selves accordingly. While this sounds great, there are in fact several problems. For example, consider subscribers that send `attach` or `detach` to the publisher within their `update` method. Since the publisher is currently traversing some kind of data structure to `update` all subscribers, the effect of these operations becomes highly dependent on the implementation of this traversal. Even worse, subscribers might send `set` within their `update` method, resulting in infinite recursion.

The component reentrance problem can be solved by implementing publishers very defensively, e.g. by cloning the data structure before traversal and by protecting the `set` method using some kind of flag. However, the problem really boils down to what messages can be sent to the publisher from within the `update` method. If we restrict this set of messages, we can *statically* ensure that the reentrance problem does not occur. Figure 5 shows

how we would model the design pattern in Lagoon. Instead of typing the **from** parameter of **update** with **Publisher**, we introduce an anonymous interface type that only supports the **get** message. While subscribers can still send other messages if they have another reference to the publisher, or if they cast the **from** parameter accordingly, our description of the design pattern is still more accurate and elegant. To achieve the same result in Java, we would have to introduce an artificial base type, e.g. **PublisherJustGet**, that we derive **Publisher** from.

### 3.4 Iterators

Certain programming languages, for example CLU [26] and Sather [34], offer an *iterator* construct to traverse encapsulated data structures in a modular manner. In most object-oriented programming languages, iterators are “emulated” at the library level [17, 37]. Using Lagoon’s mechanism for generic message forwarding, we can implement iterators that are as powerful as library approaches, but often as convenient to use as language approaches.

Figure 6 shows how iterators are commonly used in Java and similar languages that follow the library approach. The container class provides some method, in this case **elements**, that returns an iterator object. This object encapsulates the necessary operations to perform the actual traversal, and offers an interface to check for more elements or to advance to the next element. However, the iteration loop itself must be implemented manually each time an iteration is required. In Lagoon, we can offer a more elegant solution that avoids this tedious repetition. Since the **default** method enables us to specify a strategy for forwarding messages in the imperative core language, we are by no means limited to just a single receiver. Instead, we can implement a generic *broadcast* mechanism for messages. Figure 7 shows how we can use this idea to implement iterators in Lagoon. The container **Array** implements a message **forward**, which returns an iterator instance. The iterator contains a reference to the elements to be traversed and fully encapsulates the iteration strategy. For this example, we have limited ourselves to forward iteration, but a **backward** message could easily be added, returning an iterator instance for backward iteration. The actual iteration is performed by simply sending a message to the iterator instance. The iterator itself does not implement any message but instead broadcasts all received messages to the elements in the container. The actual action to be performed on each element is located in a method of the container elements. Message parameters can be used to pass additional context information from the current control flow to this iterator method.

```

import java.util.Enumeration;

class JavaIterator {
    ...
    public void action() {
        Enumeration e = container.elements();
        while (e.hasMoreElements()) {
            Object item = e.nextElement();
            System.out.println(obj);
            ...
        }
    }
}

```

Figure 6: Using iterators in Java.

This approach to iterators offers a much cleaner separation between the iteration code and the application code than traditional iterator schemes. All code related to the iteration is located in the module containing the container and its iterator functionality.

## 4 Implementation

In this section we describe our prototype implementations for Lagoona. We focus on the different message dispatch strategies that we apply in our prototypes, as this is one of the key hurdles to achieve acceptable performance when executing code written in Lagoona.

To be able to experiment with Lagoona in the context of component-oriented programming, it is not sufficient to just implement a compiler. Instead, a complete Lagoona execution environment, which supports loading and linking at runtime, has to be created as well.

We currently have two independent implementations of Lagoona. The first prototype is an interpreter for Lagoona and is written in Python. The accepted language [16] resembles closely the original Lagoona syntax [12] based on Oberon-2 [33]. The goal of this interpreter is to explore future language features and various language dialects effectively. To this end, it supports multiple frontends. The frontend for the language with Oberon-based concrete syntax is complete, and a frontend for the Java-based concrete syntax used throughout this paper is under development. As shown

```

module com.lagoona.papers.tr0212.iterator {
  ...
  class ArrayForwardIterator {
    any[] data;
    method void default() {
      int j = 0;
      while (j < this.data.length) {
        current => this.data[j++];
      }
    }
  }
  class Array {
    any[] data;
    ...
    method ArrayForwardIterator forward() {
      ArrayForwardIterator i =
        new ArrayForwardIterator();
      i.data = this.data;
      return i;
    }
  }
  class LagoonIterator {
    Array array;
    ...
    method void action() {
      array.forward().print();
    }
  }
}

```

Figure 7: Implementing iterators in Lagoona by leveraging generic message forwarding for broadcasting.

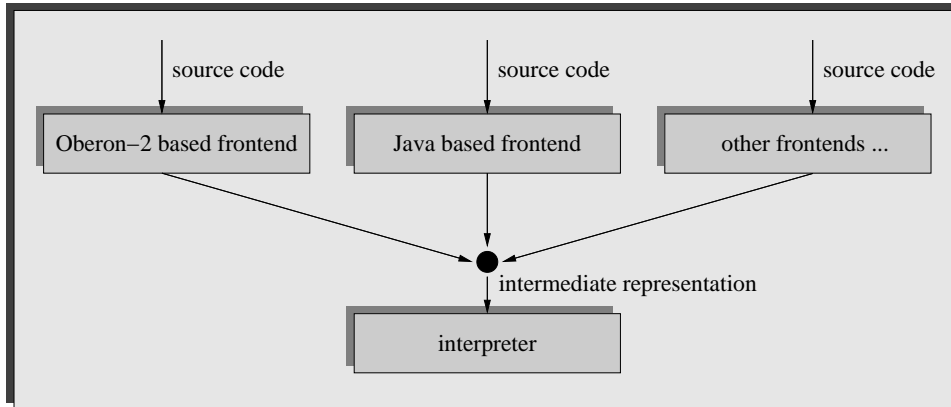


Figure 8: Architecture of the Lagoona interpreter prototype consisting of multiple frontends and a common interpreter.

in Figure 8 all frontends translate the Lagoona source code to a common intermediate representation, which is then executed by the interpreter.

The second prototype is a compiler and a dynamically optimizing runtime system for the Java-based syntax. The purpose of this project is to explore static and dynamic optimization techniques for Lagoona. The architecture of this project is shown in Figure 9. The compiler consists of a scanner and parser stage, followed by a semantic analysis and static type checking phase. The output of the compiler is an annotated syntax tree, which is our portable code representation. We have decided for a syntax tree representation instead of the byte code approach pursued by Java [25] or .NET [9] because it simplifies and speeds up the code verification step which has to be done by the runtime system before the code can be compiled. Generating optimized machine code from a syntax tree is also much simpler than from a “flat” byte code representation. In fact, most byte code compilers convert this “flat” representation back to some sort of control flow graph before performing things like register allocation. Converting an annotated syntax tree into a control flow graph is much simpler and faster than recovering the control flow graph from byte code.

Our research on efficient compilation of Lagoona code is mainly concerned with the key issues that separate Lagoona from traditional languages. Therefore, we spent only little effort to optimize the imperative language core. Instead we focused on the message dispatch problem, which is a key problem in compiling object-oriented languages in general and particularly

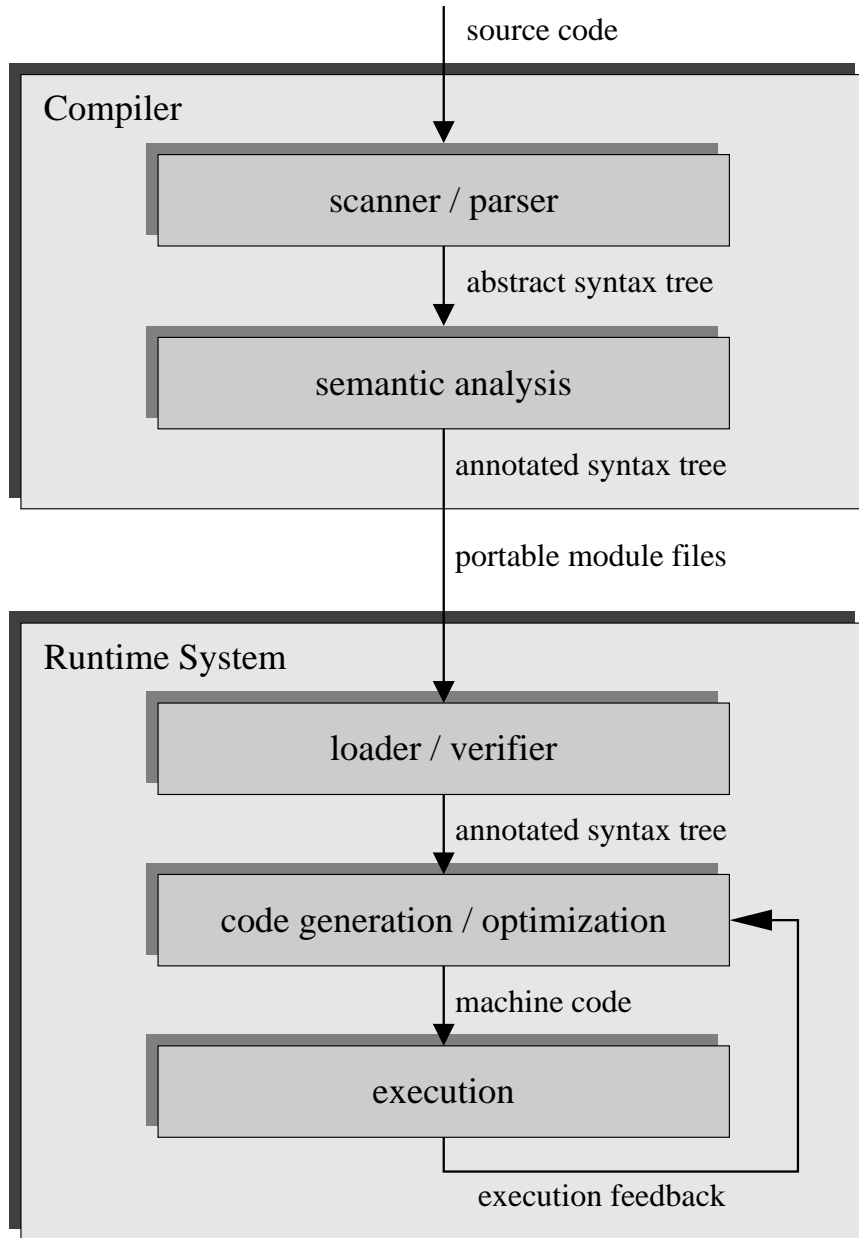


Figure 9: Architecture of the Lagoona compiler prototype consisting of a compiler frontend and a dynamic code generation backend.

languages based on stand-alone messages and generic message forwarding. Since there is no explicit type hierarchy, we cannot calculate offsets into method tables for the message dispatch at compile-time as we would in more static languages like C++ [10]. Also, since we need to support dynamic loading of modules for component-oriented programming, we can not perform a “global analysis” at compile-time to infer such a hierarchy either.

Using *dynamic compilation* allows us to circumvent this restriction (and others, see below) partially. However, for scenarios in which dynamic compilation does not pay off we also need an efficient approach for *static compilation*. We will detail our current implementation strategies for both, static compilation and dynamic compilation, in the remainder of this section.

Regardless of the compilation approach, descriptor tables for all interface and implementation types are required at runtime to perform the message dispatch. Each type descriptor table is built when the module defining the corresponding type is loaded. For each implementation type, the set of messages for which the type implements methods is stored in the descriptor table. For each message, a pointer to the corresponding implementation (method) is stored as well. For interface types, only the set of required messages is stored in the descriptor table, as interfaces cannot contain methods. It is also necessary to uniquely identify messages during execution. This can be achieved by assigning a unique numerical identifier to every message when the module that contains them is loaded.

Figure 10 shows the type descriptor tables for an interface type A and an implementation type B. To build these descriptor tables, the linker maintains a global list of all messages in the system. This list can be organized as a simple array of literal message names. The descriptor for implementation types additionally contains for each message a function pointer to the actual implementation. The globally unique numerical message identifier  $id(m)$  for a message  $m$  can be easily generated by using the address of the memory location where the literal message name is stored. The globally unique numerical representation for types is derived accordingly from the address of each type’s descriptor table. The size of each descriptor table can be determined at compile-time, but the linker is responsible to populate the descriptor tables with the appropriate message identifiers.

The global message array containing the literal names of all messages in the system can be quite large, but has to be kept in memory only as long more modules have to be loaded into the system. If an application reaches a stable state where it does not require the dynamic loading of further modules, the global message array can be purged. Even though the numerical message identifiers have been derived from address locations

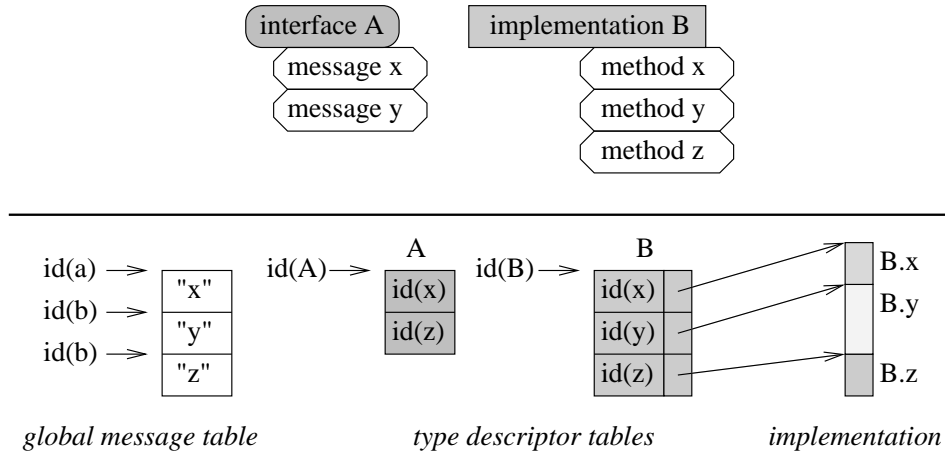


Figure 10: Layout of the type descriptor tables generated at load-time.

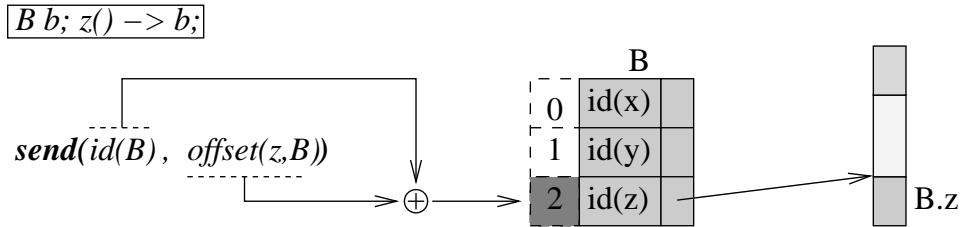


Figure 11: Resolving the message dispatch for implementation types at link-time.

within the global message array, they are never dereferenced as pointer. Thus it is safe to continue to use these message identifiers, even after the global message array has been freed.

#### 4.1 Static Compilation

In the static compilation scenario, a dispatch technique first described for Emerald [1] can be used. The basic idea is to treat interface types and implementation types independently during compilation, and to connect them with a customized mapping only on demand during execution.

The first step is to assign offsets to all messages in an interface type as well as to all methods in an implementation type at compile-time. These offsets can be calculated statically because the compiler can dictate the

order in which messages will appear in the type descriptor by ordering the message and methods in the module file accordingly. For the simplest case, if a message  $m$  is sent to an instance through an implementation reference of type  $T$ , the address of the target method can be obtained already at link-time by accessing the descriptor table of  $T$  using the compile-time calculated offset (Figure 11).

To dispatch messages sent through an interface reference, a *dispatch table* has to be generated. This dispatch table maps the message offsets of that particular interface to the actual methods to be executed on arrival of that message. The set of methods to be matched to the messages has to be selected according to the actual type of the object which has been assigned to that interface reference (Figure 12). Each interface reference needs to carry additionally to the instance pointer a pointer to the dispatch table (`dtp`) used to send messages to the object hidden behind the interface. Thus, on the machine level two words are required to represent an interface reference.

In certain situations it is desirable to explicitly widen the interface of an object reference. In the message set model this means that message are added to the set of messages the object behind a particular reference is assumed to implement. The programmer can safely perform such a cast operation if he is certain that based on the logical program structure the cast will always succeed. However, the compiler is not able to semantically verify the conformity at this point. A common application for such an explicit cast operation is to convert container element references, which are usually returned by some operation on a generic container, back to their actual implementation type.

The verification of explicit casts is performed at runtime using the type descriptors. The message set of the object addressed by the reference is compared to the message set of the type the reference has been cast to. If the conformity cannot be verified, the object must be incompatible to the interface and an exception is raised.

Pre-generating all possible dispatch tables is a waste of space, as there are  $n \times m$  possible combinations of  $n$  interface types and  $m$  implementation types. Instead, the mappings are created lazily at runtime whenever an instance is assigned to an interface reference.<sup>3</sup> Our current implementation uses a global dispatch table cache, allowing the most often requested dis-

---

<sup>3</sup>Note that this covers assignments that statically “look like” (interface type, interface type) pairs as well, since the second reference must refer to an instance and thus an implementation type.

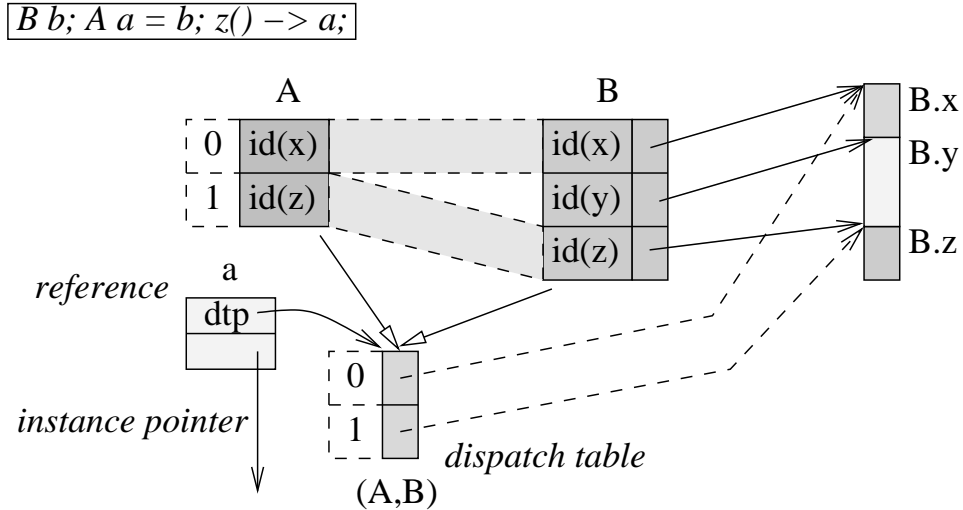


Figure 12: Dispatching a message sent through an interface.

patch tables to be reused quickly. Using an LRU scheme, dispatch tables are preempted from the cache if a certain fill rate is reached.

## 4.2 Dynamic Compilation

The dispatch scheme we described for static compilation only rarely allows us to apply common optimization techniques like *method inline expansion* [20], which are often applied when compiling object-oriented programming languages [7]. Inline expansion, or simply *inlining*, is an optimization technique which replaces frequently performed message send operations with the body of the method triggered by the reception of that message on the receiver end.

In *Lagoona*, it is particularly difficult for the compiler to perform inline expansion as the majority of message send operations are performed on interfaces and not on concrete implementation types. Because it is not visible at compile-time what actual implementation type is referenced, it is also not possible to determine the exact target method of such a message send operation. Additionally, the receiving method is often located in a different module and the compiler cannot embed code across modules without risking to render the whole application unusable if the other embedded module is exchanged without recompiling the embedding module (which is a common scenario in component-oriented programming).

To overcome these problems and limitations, and to allow for an aggressive optimization across component boundaries, we apply in our execution model for Lagoona runtime code generation [13]. In lieu of compiling the source code to direct machine code for some target architecture, an intermediate representation is generated, which is an annotated syntax tree in our case (Figure 9). The runtime system contains a code generator which translates this intermediate representation at load-time into machine code for execution. However, the translation process is not limited to load-time, as the majority of optimizations (like inline expansion) can benefit from execution feedback. The result is a *dynamic compilation* process which performs a *continuous program optimization* [22, 23].

One of the main sources causing Lagoona to rely so heavily on dynamic compilation is message forwarding, which is (amongst other things) used to express inheritance-like object relations. Figure 13 shows how a class B extends the class A by instantiation an object of type B. The message a is intercepted by the “derived class” B and some local action is performed before the message is forwarded on to the de-facto “base class” A. This scenario calls for inline expansion of the method A.a directly into the method body of B.b to prevent the costly message dispatch. However, the compiler cannot perform this action, as A and B are located in two separated modules which might change independently in a distributed programming environment. The runtime system in contrast can deal with this optimization problem easily, as it has a global view over all modules currently loaded into the system.

This particular inline expansion can be performed **ahead-of-time**, because the de-facto “base class” A is referenced through a reference with a concrete implementation type. If an interface type is used as forwarding destination, more effort has to be spent to identify possible candidates for inline expansion. At first the linker has to check whether more than one implementation type fulfills the interface. If only a single implementation type can be assigned to the interface reference (only one loaded implementation type is compatible to this interface), the inline expansion can be performed safely ahead-of-time. However, if an additional module containing a matching type is loaded, the inline expansion has to be reconciled. If more than one matching implementation type currently exists in the system, the assignment operations to the interface reference have to be checked carefully. If it is not possible to guarantee the actual type of all objects possibly assigned to the interface reference to be from one uniform type, the inline expansion has to be delayed up to the point of the actual reference initialization. This initialization usually takes place inside the *initialize* method. If an as-

signment operation to the reference is encountered, a *just-in-time* dynamic compilation process is started. As multiple instance of a class might assign different actual implementation types to their de-facto “base class” reference, the resulting machine code is bound to this particular object instance and does not simply replace the already existing method implementations. As the recompilation process has a significant cost itself, our prototype implementation uses a heuristic based on profiling information gathered on the fly to decide at runtime whether a detected possible optimization is actually probable to pay off at some point in the future, or not. Only optimization which have been predicted as profitable are actually performed.

## 5 Related Work

We have already discussed some related work in previous sections as part of our arguments. In this section, we discuss a broader range of related developments to place our work into a larger context.

### 5.1 Language Mechanisms

Stand-alone messages can be related to the concept of *multimethods* [2]. In a language supporting multimethods, such as Cecil [5], stand-alone messages could be “emulated” by introducing an additional dispatch parameter modeling the originating module. Despite recent progress regarding type-safety and modularity of multimethods [32], the concept is not yet supported in mainstream languages. Stand-alone messages are conceptually simpler than multimethods because they only rely on the established notion of modules and add no additional concerns for separate compilation. They also maintain the established object-oriented programming style.

Recent work on *units* and *mixins* [11] is related to Lagoona in a more interesting way. With Lagoona, we have argued that programming languages for component-oriented programming need to combine traits from modular languages with traits from object-oriented languages in a certain way. Namely, we have to distinguish explicitly between messages and methods and we have to separate messages from types, binding them to modules instead. Units and mixins also aim at the combination of modular and object-oriented language constructs. Units provide a module concept that is more flexible than ours: Instead of fixing the import relations of a set of modules once and for all, units allow the composition of modules through separate linking specifications. This has several important applications, for

```

module com.lagoona.papers.tr0212.MA {
  class A {
    method void a() {
      ...
    }
    method void b() {
      ...
    }
  }
}

module com.lagoona.papers.tr0212.MB {
  import MA = com.lagoona.papers.tr0212.MA;
  class B {
    MA.A base;
    method void initialize() {
      this.base = new MA.A();
    }
    method void MA.a() {
      ...
      MA.a() -> this.base;
    }
    method void default() {
      current => this.base;
    }
  }
}

```

Figure 13: Emulating inheritance in Lagoona using generic message forwarding.

example for the flexible creation of extended objects. Mixins provide a variation of inheritance (in the sense of subclassing) that allows derived classes to be parameterized by different base classes. However, Lagoona’s approach to forwarding and composition already subsumes mixins: while for mixins the base class relation is determined when units are linked, in Lagoona we can actually defer this relation until objects are instantiated. In summary, the units idea is very valuable, and we hope to explore the integration of a more flexible module system (with a distinct “units” flavor) into Lagoona in the future.

## 5.2 Component Models

Component models, such as COM [29], CORBA [35], and JavaBeans [38], are industry standards that claim to support component-oriented programming. However, the main emphasis of these models lies on defining interoperability and packaging conventions in the form of design patterns, rather than on providing comprehensive support. Many component models also address aspects that are essentially unrelated to component-oriented programming—such as distribution, concurrency, cross-platform portability, and cross-language integration—but that nevertheless increase their complexity significantly. Component models seem to be a temporary solution that will survive only until better, more comprehensive ways to practice component-oriented programming become available. We do not want to imply that component models are completely useless, but rather that they only serve a temporary purpose as far as the component-oriented paradigm is concerned.

## 5.3 Generative Programming

The paradigm of generative programming (GP) [6] is based on a number of ideas: domain specific languages, aspect-oriented programming (AOP), and generic programming. In GP, software systems are described in terms of domain specific languages that are used to encode domain knowledge on a high level. These descriptions are used to drive AOP [21] tools that integrate various reusable and basically unrelated “components” and aspects to produce customized applications automatically. The functional “components” are implemented using generic programming techniques (i.e. parametric polymorphism). While GP provides an interesting approach to source-level reuse and maintenance, its “components” are not components in the sense of component-oriented programming [41]. In GP (and AOP),

“components” are reusable and parameterized abstractions that only exist on the programming language level, but not in the deployed application. Thus, once an application has been produced using GP, the “components” it consists of can not be reused or updated separately from the application they were compiled into.

## 5.4 Dynamic Compilation

The term *dynamic compilation* has originally be coined in the context of Smalltalk-80 [8]. The Smalltalk-80 compiler emitted portable *v-code*, which in turn was translated to native machine code (*n-code*). Like Lagoon, Smalltalk-80 relied heavily on message send operations and used a similar caching scheme to speed up the invocation of often requested methods. Self [42] is a dynamically typed, object-oriented language, which in its later incarnations Self-91 and Self-93 [19] also used profiling and dynamic compilation to speed up performance critical parts of the program. The optimizations undertaken by *Self* differ considerably from dynamic compilation in Lagoon, as Lagoon is “more statically typed” than *Self*. This often allows to predict the real object type addressed through a reference ahead-of-time, which in turn simplifies the profitability prediction of dynamic optimizations. Our research also relates closely to work done by Kistler on continuous program optimization [22, 23].

## 6 Future Work

Just as we are pursuing two independent implementation approaches for Lagoon, our focus for future research in this area can be divided into two major directions.

First, we want to make further enhancements to the language design itself. We have recently promoted messages in Lagoon to first class entities and are now evaluating the resulting new possibilities. By adding reflective [43] capabilities to messages, we intend to significantly simplify the implementation of distribution object frameworks. Our aim is to create *generic proxy objects*, which forward all received messages to an object on the server side. Related to this approach is the idea to integrate concurrency into Lagoon. Our message forwarding capabilities would allow to express various approaches to message queuing very efficiently.

Another area in Lagoon we are currently attempting to improve is static type checking. For example, our current type system fails to detect “emulated” inheritance relations involving generic message forwarding. We plan

to add analytical capabilities similar to those used by the dynamic optimization process to our compiler framework, to allow the type checker to understand a limited subset of “simple” forwarding patterns. This would not only tighten static type-checking, but also increase the runtime efficiency as some dynamic cast operations can be saved.

The other major area we are focusing our research on is to make the execution model of *Lagoona* more efficient. The dispatch optimizations presented in Section 4 are only the first step in this direction. In the future we want to add more aggressive optimizations to reduce the language level overhead introduced by the high-level concepts in *Lagoona*. We are also investigating how the pay-off prediction for the dynamic recompilation process can be improved. In many cases, our current prototypes fail to predict less-frequently used message-send operations correctly and the dynamic compiler wastes a significant amount of time in optimizing code passages which are rarely used. As part of this research, we are working on producing meaningful benchmarks to measure the actual performance of our prototype implementations. We are exploring different approaches and possible benchmarks. While we are able to show the performance gain resulting from dynamic compilation in specifically designed test scenarios, we would like to apply our techniques to more general applications and real-world problems. Our progress in this area is slow, because we have to port benchmarks in question first to the *Lagoona* language, before we can conduct any measurements.

## 7 Conclusions

In this paper we presented *Lagoona*, an experimental programming language based on stand-alone messages and message forwarding. After motivating *Lagoona* in the context of component-oriented programming, we briefly explained the basic language features of *Lagoona* and showed how these language properties help to solve a number of well-known design and implementation problems in object-oriented and component-oriented programming. In the following implementation section we explained that the previously demonstrated advantages resulting from *Lagoona*’s language design do not come for free and make the message dispatch more complex. At the same time we also proposed the application of *dynamic compilation* to overcome these performance problems.

We believe that the main contribution of this work is an improved understanding of how modular and object-oriented concepts interact and how they can be combined to support component-oriented programming. Our

approach to separate messages from methods can be viewed as another step towards the separation of concepts subsumed by classes in traditional object-oriented languages. Previous results in this direction include the separation of interface types from implementation types [36] and the separation of modules from types [40], which both are widely accepted now. By reconciling the concept of stand-alone message with an efficient execution model we have removed a major obstacle which has hampered the application of stand-alone messages and generic message forwarding in the past. To allow others to use Lagoona as basis for further research on component-oriented programming languages, we plan to release the Lagoona framework, consisting of a compiler, runtime system, and runtime libraries, to the public in the near future.

## Acknowledgments

We would like to thank Fermin Reig for valuable comments on earlier versions of this paper. The TRANSPROSE research group is responsible for our view of dynamic compilation. We are also indebted to Wolfram Amme, Erik Ernst, Ziemowit Laski, Olaf Spinczyk, Christian Stork, Clemens Szyperski, and Mia Wallace for many fruitful discussions. This work was partially supported by the National Science Foundation under grants EIA-9975053 and CCR-0105710.

## References

- [1] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the Emerald system. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 78–86, Portland, OR, November 1986.
- [2] Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. CommonLoops: Merging Lisp and object-oriented programming. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 17–29, Portland, OR, November 1986.
- [3] Martin Büchi and Wolfgang Weck. Compound types for Java. In *Proceedings of the Conference on Object-Oriented Programming, Systems,*

- Languages, and Applications (OOPSLA)*, pages 362–373, Vancouver, British Columbia, October 1998.
- [4] Luca Cardelli. Typeful programming. SRC Research Report 45, Digital Systems Research Center, May 1989.
  - [5] Craig Chambers. The Cecil language: Specification and rationale. Technical report, Department of Computer Science and Engineering, University of Washington, Box 352350, Seattle, WA 98195-2350, USA, March 1997. <http://www.cs.washington.edu/research/projects/cecil/>.
  - [6] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
  - [7] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In Walter Olthoff, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 952 of *Lecture Notes in Computer Science*, pages 77–101, Århus, Denmark, August 1995. Springer Verlag.
  - [8] L. Peter Deutsch and Allan M. Schiffmann. Efficient implementation of the Smalltalk-80 System. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 297–302, Salt Lake City, UT, January 1984.
  - [9] ECMA. The .NET Common Language Infrastructure. Technical Report TR/84, ECMA, Geneva, Switzerland, June 2001.
  - [10] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.
  - [11] Robert Bruce Findler and Matthew Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of the International Conference on Functional Programming (ICFP)*, pages 94–104, Baltimore, MD, 1998.
  - [12] Michael Franz. The programming language Lagoon: A fresh look at object-orientation. *Software: Concepts and Tools*, 18(1):14–26, March 1997.
  - [13] Michael Franz. Toward an Execution Model for Component Software. In *Workshop Reader of the European Conference on Object-Oriented Programming (ECOOP)*, pages 144–149, March 1997.

- [14] Peter H. Fröhlich and Michael Franz. Stand-alone messages: A step towards component-oriented programming languages. In Jürg Gutknecht and Wolfgang Weck, editors, *Proceedings of the Joint Modular Languages Conference*, volume 1897 of *Lecture Notes in Computer Science*, pages 90–103, Zürich, Switzerland, September 2000. Springer-Verlag.
- [15] Peter H. Fröhlich and Michael Franz. On certain basic properties of component-oriented programming languages. In David H. Lorenz and Vugranam C. Sreedhar, editors, *Proceedings of the First OOPSLA Workshop on Language Mechanisms for Programming Software Components*, pages 15–18, Tampa Bay, FL, October 15 2001. Technical Report NU-CCS-01-06, College of Computer Science, Northeastern University, Boston, MA 02115.
- [16] Peter H. Fröhlich and Michael Franz. The programming language Lagoon. Technical report, Department of Information and Computer Science, University of California, Irvine, CA 92697-3425, USA, 2002.
- [17] Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [18] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, 2nd edition, 2000.
- [19] Urs Hölzle. *Reconciling High Performance with Exploratory Programming*. PhD thesis, Department of Computer Science, Stanford University, 1994.
- [20] Wen-Mei W. Hwu and Pohua P. Chang. Inline function expansion for compiling C programs. In *Proceedings of SIGPLAN 89 Conference on Programming Language Design and Implementation*, Portland, OR, 1989.
- [21] Gregor Kiczales, John Lamping, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, June 1997.
- [22] Thomas Kistler. *Continuous Program Optimization*. PhD thesis, Department of Information and Computer Science, University of California, Irvine, CA 92697-3425, USA, November 1999.

- [23] Thomas Kistler and Michael Franz. Continuous Program Optimization: Design and Evaluation. *IEEE Transactions on Computers*, 50(6):549–566, June 2001.
- [24] Konstantin Läufer, Gerald Baumgartner, and Vincent F. Russo. Safe structural conformance for Java. Technical Report OSU-CISRC-6/98-TR20, Department of Computer and Information Science, Ohio State University, Columbus, OH 43210-1277, June 1998.
- [25] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.
- [26] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. CLU Reference Manual. Technical Report MIT/LCS/TR-225, MIT Laboratory for Computer Science, October 1979.
- [27] M. Douglas McIlroy. Mass-produced software components. In *Proceedings of the NATO Software Engineering Conference*, pages 138–155, Brussels, Belgium, October 1968.
- [28] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.
- [29] Microsoft Corporation. *The Component Object Model (Version 0.9)*, October 1995. <http://www.microsoft.com/COM/resources/COM1598C.ZIP>.
- [30] Leonid Mikhajlov and Emil Sekerinski. A study of the fragile base class problem. In Eric Jul, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1445 of *Lecture Notes in Computer Science*, pages 355–382, Brussels, Belgium, July 1998. Springer-Verlag.
- [31] Leonid Mikhajlov, Emil Sekerinski, and Linas Laibinis. Developing components in presence of re-entrance. In *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems (FM)*, volume 1709 of *Lecture Notes in Computer Science*, pages 1301–1320, Toulouse, France, September 1999. Springer-Verlag.
- [32] Todd Millstein and Craig Chambers. Modular statically typed multi-methods. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1628 of *Lecture Notes in Computer Science*, pages 279–303. Springer-Verlag, June 1999.

- [33] Hanspeter Mössenböck and Niklaus Wirth. The programming language Oberon-2. *Structured Programming*, 12(4):179–195, 1991.
- [34] Stephan Murer, Stephen Omohundro, David Stoutamire, and Clemens Szyperski. Iteration abstraction in Sather. *Transactions on Programming Languages and Systems*, 18(1):1–15, January 1996.
- [35] Object Management Group. *The Common Object Request Broker: Architecture and Specification (Version 2.3.1)*, October 1999. <http://www.omg.org/cgi-bin/doc?formal/99-10-07>.
- [36] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 38–45, Portland, OR, November 1986.
- [37] Alexander A. Stepanov and Meng Lee. The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, Silicon Graphics Inc., 1994.
- [38] Sun Microsystems. *The JavaBeans Specification (Version 1.01)*, July 1997. <http://www.javasoft.com/beans/docs/beans.101.pdf>.
- [39] Sun Microsystems. *The OpenOffice Source Project Homepage*, 2002. <http://www.openoffice.org/>.
- [40] Clemens Szyperski. Import is not inheritance—why we need both: Modules and classes. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 615 of *Lecture Notes in Computer Science*, pages 19–32, Utrecht, The Netherlands, June 1992. Springer-Verlag.
- [41] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.
- [42] David Ungar and Randall B. Smith. SELF: The Power of Simplicity. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 227–242, Orlando, FL, December 1987.
- [43] Takuo Watanabe and Akinori Yonezawa. Reflection in an Object-Oriented Concurrent Language. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, September 1988.