Available online at www.sciencedirect.com
SCIENCE DIRECT®
Science of Computer Programming 56 (2005) 41–57
Science of Computer Programming
www.elsevier.com/locate/scico

Supporting software composition at the programming language level

Peter H. Fröhlich[a,*], Andreas Gal[b], Michael Franz[b]

[a] *Department of Computer Science and Engineering, University of California, Riverside, CA, United States*
[b] *School of Information and Computer Science, University of California, Irvine, CA, United States*

Received 31 October 2003; received in revised form 22 August 2004; accepted 6 September 2004
Available online 9 December 2004

Abstract

We are in the midst of a paradigm shift toward component-oriented software development, and significant progress has been made in understanding and harnessing this new paradigm. Oddly enough, however, the new paradigm does not currently extend to the level at which components themselves are constructed. While we have composition architectures and languages that describe how systems are put together out of atomic program parts, the parts themselves are still constructed on the basis of a previous paradigm: object-oriented programming. We argue that this mismatch impedes the progress of compositional software design: many of the assumptions that underlie object-oriented languages simply do not apply in the open and dynamic contexts of component software environments. What, then, would a programming language that supported component-oriented programming at the smallest granularity look like? Lag*oo*na, our project to develop such a language, tries to answer this question. This paper motivates the key concepts behind Lag*oo*na and briefly describes their realization (using Lag*oo*na itself as the implementation language) in the context of Microsoft's .NET environment.

Keywords: Component-oriented software development; Programming languages; Distributed extensibility; Language paradigms beyond object-oriented programming

* Corresponding author. Tel.: +1 951 827 2604; fax: +1 951 827 4643.
E-mail address: phf@acm.org (P.H. Fröhlich).

0167-6423/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2004.11.004

1. Introduction

While the idea of "software components" was proposed as far back as the 1960s [23], the arrival of the Internet has propelled us into an age where component-oriented programming (COP) is becoming simultaneously viable and necessary: viable, because an efficient component discovery and distribution mechanism is now available; necessary, because the complexity of Internet-enabled applications often exceeds the abstraction capabilities of existing programming paradigms. There is, of course, much confusion about what COP *actually* means [36,20]. This state of affairs is similar to the confusion surrounding object-oriented programming (OOP) in the 1980s. As with OOP, the "essence" of COP is not primarily found in technical details of programming language design and implementation (although we focus on its *implications* for these areas in the following). Instead, and again similar to the OOP case, the importance of the paradigm lies in its conceptual vision, i.e. the software architectures that it strives for, the software qualities that it emphasizes, and the software development processes that it mandates. The latter is also the most striking difference between COP and established paradigms, which are usually silent on issues of process.

We contend that the "essence" of COP is the notion of *distributed extensibility*, in contrast to the notion of *centralized reuse* which has been the focus of software components since they were first proposed. Centralized reuse means that software components are acquired by an application vendor who in turn sells a monolithic application to users. The application vendor alone has *complete* control over the integration process, deciding which components are delivered as part of the final application. Once deployed, the application cannot be "reintegrated" with newer or different components, keeping the application vendor in control. Also, without "privileged" access to the internals of the application, no party except the application vendor can develop extensions. In contrast, distributed extensibility (see Fig. 1) means that *any* interested party can develop extensions, which can be acquired and integrated by *anyone* at *any* time [12]. Monolithic "applications" disappear under distributed extensibility, to be replaced by *components* and *frameworks* (see Fig. 2). Components provide functional extensions for (domain-specific) frameworks, while frameworks provide (customized) execution environments for components.

The fundamental difference in process between centralized reuse and distributed extensibility also has profound implications for programming languages. It is current practice to *approximate* certain COP ideas using a variety of essentially OOP languages

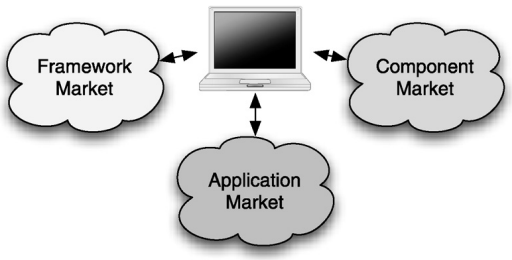

Fig. 1. Distributed extensibility enables anyone to independently develop, acquire, and integrate anything, anytime.

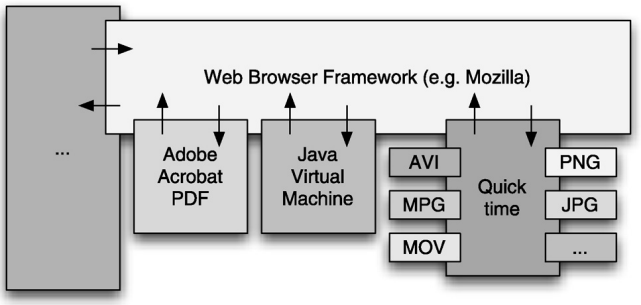

Fig. 2. Architecture of a Web browser in terms of frameworks and components, also illustrating hierarchical composition.

such as C++ [7], Eiffel [24], and Java [16]. However, these languages—while sometimes well designed—are still based on assumptions that contradict the notion of distributed extensibility. For example, Eiffel's covariant argument types require a form of global analysis that implies a "closed system" view of the world, whereas distributed extensibility mandates "open systems" that are never complete in the sense of monolithic applications. Similarly, evolving an existing class library can lead to syntactic conflicts ("name clashes") or semantic conflicts ("fragile base classes") in already developed extensions, causing them to fail. These are just examples that show the mismatch between current OOP practice and the ultimate requirements of COP. As a result of this misalignment, today's component systems fall short of their true potential. Rather than being truly composed from independently developed parts, they rely on the presence of large, shared underlying class libraries that themselves cannot be evolved easily.

Our research, on which we report here, has focused on developing an experimental programming language—Lag*oo*na—that supports COP *expressly*. Lag*oo*na retains much of the flavor and benefits of OOP languages but discards those elements that contradict the COP paradigm. In the following, we first introduce Lag*oo*na itself, focusing on two novel language mechanisms: *stand-alone messages* and *generic message forwarding* (Section 2). Next we show how these mechanisms allow us to eliminate or alleviate a number of (sometimes long-standing) design and implementation problems in OOP-based COP, issues such as *interface conflicts*, *fragile base classes*, and *component re-entrance* (Section 3). We then briefly describe our implementation and show how we address the challenges for efficient execution (Section 4). The final sections discuss related work, future work, and offer our conclusions.

2. Lag*oo*na

Lag*oo*na is designed around a standard imperative language core, a choice made more for reasons of familiarity than necessity. Lag*oo*na's object model, however, is different from those found in established OOP languages: it separates the many roles traditionally played by classes, turning them into individual language constructs [12,10,13]. Fig. 3

Concern	**Traditional**	**Lag*oo*na**
Encapsulation	access modifiers	Module
Specification	abstract method	Message
	abstract class	Interface
Implementation	concrete method	Method
	concrete class	Implementation
Modification	inheritance	Forwarding

Fig. 3. Design concerns and corresponding language constructs in traditional OOP languages and in Lag*oo*na.

provides a concise comparison of how design concerns are mapped onto the class construct in traditional OOP languages and onto separate constructs in Lag*oo*na.

Overview. At the lowest level of Lag*oo*na's object model are *messages* and *methods*, clearly separated into two language constructs. Messages are *abstract operations* that describe *what* effect they achieve, while 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. Although messages are a syntactic notion on the language level, we consider each message to denote a (formal) specification of its semantics as well. At the next higher level, messages and methods are grouped into *interface types* and *implementation types*—again, clearly separated into two language constructs. 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, and we consider each interface type to denote a (formal) specification of its semantics. At the highest level of the object model are *modules* that 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.

Types and conformance. 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, 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 [13]. 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:

$$\text{IntIntConf}\ \frac{\Gamma \vdash A = M_A \qquad B = M_B \qquad M_A \subseteq M_B}{\Gamma \vdash A \leq B} \tag{1}$$

In other words, we employ *structural conformance* or *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:

$$\text{IntImpConf}\ \frac{\Gamma \vdash B = M_B \qquad C = M_C \qquad M_B \subseteq M_C}{\Gamma \vdash B \leq C} \tag{2}$$

We thus extend structural conformance to implementation types, and if (1) and (2) hold, $A \leq C$ will hold as well. This enables a form of inclusion polymorphism [4] which we like to call *implementation polymorphism* to emphasize that only implementations can vary, and only "behind" an interface. Third, an interface type never conforms to an implementation type. Of course, Lag*oo*na allows interface types to be *cast* to implementation types, guarded by a dynamic check. However, for a cast to be statically valid, the reference that it is applied to must be explicitly declared *castable*. While the use of castable references is discouraged, it is nevertheless necessary, e.g. for useful container abstractions. Castable references can be assigned to non-castable ones, but non-castable references can *not* be assigned to castable ones. Finally, two implementation types only conform if they are the same type. In other words, we employ *occurrence equivalence* between implementation types: an object of implementation type A can *not* be assigned to a implementation reference of type B even if A and B implement identical message sets. This completes the definition of conformance, but the fourth case raises the question how implementation types can be reused or adapted.

Sending and forwarding. At runtime, Lag*oo*na's object model essentially reduces to a Web of independent instances that communicate through messages. Assume that 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:

$$\text{StrictSend}\ \frac{\Gamma \vdash R = M_R \qquad m \in M_R \qquad r : R}{\Gamma \vdash m \rightarrow r} \tag{3}$$

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 as long as m is a message and r is a reference:

$$\text{BlindSend}\ \frac{\Gamma \vdash m \in M \qquad r : R}{\Gamma \vdash m \Rightarrow r} \tag{4}$$

Of course, we have to guard the application of this operator by a dynamic check, similar to the one for casts mentioned above. The blind message send operator is necessary for supporting 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 the default method, the received message, including its arguments, can be *forwarded* to other instances. We use the term *generic message forwarding* to express that the actual message remains *opaque* in this process, being denoted by a predeclared polymorphic identifier. Since we lack static information about the *actual* message being forwarded, the strict message send operator alone would not be sufficient to support this. However, to ensure reasonable semantics for blind

```
module com.lagoona.stacks {
  public message void
    push(castable Any obj);
  public message void pop();
  public message castable Any top();
  public message boolean empty();
  public interface Stack {
    push, pop,
    top, empty
  }
}
```

Fig. 4. A stack abstraction in Lag*oo*na. Messages are bound to modules, not types.

message sends, various dynamic checks are performed. If no default method exists in the implementation type where a message without an explicit method is received, an exception is raised. Similarly, if the forwarded message returns a result but no actual result is returned from any method during the forwarding process, an exception is raised.

Modules and components. Lag*oo*na's top-level construct, the *module*, serves a variety of purposes. Following [36], we consider components to be collections of modules. Modules are compilation units resulting in object files, which in turn are the units of deployment. Following [3], and in contrast to Java packages or C++ namespaces, modules are *sealed*: only explicitly exported declarations are visible to clients, and no new declarations can be added from the outside. Modules live in a flat, global namespace and cannot be nested [14]. However, we employ a hierarchical naming convention based on Internet domain names, similar to the one originally proposed for Java. 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.

Stack example. As an example of the concrete syntax used by the Lag*oo*na compiler, we now discuss a simple stack abstraction and its implementation. The module shown in Fig. 4 exports the necessary messages as well as an interface type `Stack`. Note that `Stack` is only a convenient abbreviation; since conformance is defined by structure, we could repeatedly declare isomorphic interface types instead of using the name `Stack`. As mentioned above, we usually associate a semi-formal specification with each message. The `push` message, for example, would be characterized with the precondition "$obj \neq null$" and the postcondition "$\neg empty()$". The predeclared interface type `Any` represents the empty message set and is the top element in the resulting type lattice. Since we are defining a reusable container abstraction, we also need to ensure that clients can cast the result of `top` back to the type of object that they supplied in `push` by adding the `castable` qualifier. Note that we consider messages and their specifications *immutable* once published, similar to the assumption made about interfaces in COM [25]. Conceptually, interface types in Lag*oo*na are used to decouple independent components, again similar to the COM case and—to a certain extent—that for Java. The module shown in Fig. 5 exports an implementation type for stacks. It imports the earlier module under the alias `S` and uses this alias to qualify further references, for example to the message `push`. Implementation types (`class` in the concrete syntax) host methods and declarations of instance variables.

```
module com.lagoona.simple_stacks {
  import S = com.lagoona.stacks;
  class Link {
    castable Any object; Link next;
  }
  public class Stack {
    Link top;
    method initialize() {
      this.top = null;
    }
    method void S.push(castable 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 castable Any S.top() {
      return this.top.object;
    }
    method boolean S.empty() {
      return this.top == null;
    }
  }
}
```

Fig. 5. An implementation of the stack abstraction.

Except for `initialize`, each method in Fig. 5 implements exactly one message imported from the module `S`. The messages `initialize` (and `finalize`) have special meaning in Lag*oo*na, which is also why they have no return type: they are sent by the runtime system immediately after an instance has been created (and immediately before it is garbage collected). Note that several declarations in Fig. 5 remain hidden from clients, for example the class `Link` which is used locally as a simple record type without any methods. Fig. 6 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 `size` 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 `size` simply returns the counter value. The methods `S.push` and `S.pop` update the counter and forward their messages to the "basic" stack instance. For illustration, we also implement `S.empty` again to use the counter instead of checking the list pointer for `null`. All other messages, including `S.top`, will be handled by the special `default` method, which simply forwards them to an instance of the "basic" stack.

Common objections. At this point, it is useful to address a number of frequently voiced objections to Lag*oo*na's object model. First, it is often pointed out that the benefits of stand-alone messages can also be achieved by extending explicit qualification of messages to the interface types that define them. Instead of `S.push` we would then write `S.Stack.push`

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

Fig. 6. Adding counting to the stack abstraction and its implementation.

to refer to the `push` message defined as part of the type `Stack` in module `S`. However, aside from it being *notationally* more complex, there are also *conceptual* problems with this approach. On the one hand, we need to differentiate between *new* messages that are *introduced* as part of a type, and *existing* messages that are merely *reused* as part of the type's declaration. For example, we could consider qualified messages occurring in a type declaration as "reused" and non-qualified messages as "new". Now we are dealing with a *conceptually* more complex rule for interface types, a fact that becomes particularly obvious when anonymous interface types are considered: inside an anonymous type, we must *forbid* unqualified messages since there is no type name with which to qualify. On the other hand, there are situations where two interface types declared in a single module support one and the same message. If we declare messages inside of types, we have to choose one or the other as the "defining type" for the message, for no particular reason. Stand-alone messages enable us to avoid these complications, leading to a simpler language design. Another recurring objection is that forwarding is not powerful enough to serve as a *full* replacement for inheritance (or delegation) due to its lack of recursive self-binding. We agree with this observation in principle: there *are* certain call patterns—easily achieved using inheritance—which cannot be replicated in obvious ways using forwarding alone. However, those call patterns are also among the most complex and therefore most difficult to understand ones, casting some doubt on their long-term utility from a software maintenance perspective. Also, there is not much evidence that these complex call patterns are actually widely used. In contrast, a study of popular object-oriented design patterns [11]

shows that almost none of them are "lost" if we replace inheritance by forwarding. Finally, one might object to the semantics of blind message sends and therefore generic message forwarding. Indeed, using the "⇒" operator, we can send *any* message to *any* reference, without the compiler being able to perform static checking. However, removing blind message sends would restrict the degree of extensibility that we can achieve, particularly in the case of frameworks. Consider a framework for optimizing compilers for example [18]. In order to add new optimizations, we need to be able to add new analysis components (AC) as well as new transformation components (TC). A particular TC may depend on information from a particular AC, information that requires sending a message into the framework that was not defined when the framework was deployed. If we did not allow blind message sends in this context, we would rule out a whole class of interesting applications for COP. There is a trade-off between the degree of static safety that can be achieved and the degree of dynamic extensibility that is desirable. With blind message sends, we consciously take the side of extensibility over safety.

3. Applications

In this section, we illustrate how stand-alone messages and generic message forwarding address a number of recurring design and implementation problems that are pervasive when COP is practised using OOP languages.

Interface combination and conformance. COP often requires *combining* multiple interface types that were defined independently, for example if they are to be implemented by a single implementation type. 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 problematic in conventional OOP languages, since it can lead to *syntactic* and *semantic* conflicts [13]. 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 [33]. For example, Java supports overloading of method names, which can be used to avoid a subset of these conflicts. More general solutions are provided in Eiffel, which supports renaming of methods in descendant classes, and in C++, which supports a form of explicit qualification of methods. However, these techniques are only applied to fix a "name clash" once it has occurred, too late if we want to support distributed extensibility. In Lag*oo*na, both kinds of conflicts are ruled out *by design* since messages always have a unique identity. Interface combination in Lag*oo*na 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 were defined by independent vendors. Two other vendors now define—again independently—interface types that combine A and B, say C and D. That is, both C and D support *exactly* the union of message sets denoted by A and B. Obviously, we would like implementations of C to conform to D and vice versa. However, most OOP languages rely

on a declared form of conformance, i.e. types are equivalent *by occurrence* instead of *by structure*. Again, the established OOP approach fails to take distributed extensibility into account. 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. Lag*oo*na's stand-alone messages provide a solution to this problem: since messages have unique identities, we can support structural conformance *without* the potential for accidental conformance. Proposals to extend Java with a form of structural conformance [2,19] result in a more complicated and less flexible design.

Fragile base classes. The concept of *inheritance* was once hailed as the "silver bullet" for extensible software. However, inheritance is generally not suitable for achieving distributed extensibility as mandated by COP. Consider a stack abstraction similar to the one in Fig. 5, but also supporting the message `multi_pop` to take $n > 0$ elements off the stack at once. There are two obvious ways of implementing `multi_pop`, either by repeatedly sending `pop` to `this`, or by accessing the underlying list directly. Given one such implementation, we now want to extend this stack with a `size` message similar to Fig. 6. Using inheritance, we would declare a derived class and attempt to override the relevant methods to maintain a size counter. However, we cannot decide which methods to override without knowing the details of the `multi_pop` implementation: if `multi_pop` repeatedly sends `pop`, we must override `pop` (and only `pop`) to count correctly. If, however, `multi_pop` accesses the list directly, we must override both `pop` and `multi_pop` instead. This paradox is known as the *fragile base class problem*, and it can be resolved by using an elaborate set of programming conventions [26]. If we want to avoid the problem altogether, we have to restrict the use of inheritance or abolish the mechanism completely. In Lag*oo*na, generic message forwarding takes the place traditionally occupied by inheritance, and solving the problem posed above simply requires a `multi_pop` method in the implementation of counting stacks—regardless of the implementation choice made in basic stacks.

Component re-entrance. When using messages and interface types to specify the functionality of objects, we often make the assumption that each operation executes *atomically*. However, for certain design patterns that rely on "callbacks" between objects this is not the case, leading to the *component re-entrance* problem [36,27]. Consider the *Observer* (or *Publish–Subscribe*) design pattern [15] 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. Fig. 7 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 themselves accordingly. While this achieves the goal of loose coupling, 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 the traversal. Even worse, subscribers might send `set` within their `update` method, resulting in infinite

```
package com.lagoona.pubsub;
public interface Publisher {
  void attach(Subscriber me);
  void detach(Subscriber me);
  Object get();
  void set(Object data);
}
public interface Subscriber {
  void update(Publisher from);
}
```

Fig. 7. Naive publishers and subscribers in Java.

```
module com.lagoona.pubsub {
  interface Publisher {attach, detach, get, set}
  message void attach(Subscriber me);
  message void detach(Subscriber me);
  message castable Any get();
  message void set(castable Any data);

  interface Subscriber {update}
  message void update(interface {get} from);
}
```

Fig. 8. Smarter publishers and subscribers in Lag*oo*na; only `get` can be sent within `update`.

recursion. The component re-entrance 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 re-entrance problem does not occur. Fig. 8 shows how we would model the design pattern in Lag*oo*na. Instead of typing the `from` parameter of `update` with `Publisher`, we introduce an anonymous interface type that only supports the `get` message. Note that the parameter to `update` is not `castable`; as long as subscribers are not able to somehow obtain another, more permissive reference to their publisher, this solves the component re-entrance problem statically. In Java, we would have to introduce an artificial base type, e.g. `Gettable`, that we derive `Publisher` from. In general, we can use stand-alone messages and structural conformance to achieve *minimal interfaces* without the need for artificial supertypes. Better yet, we can practice *retroactive supertyping* and avoid the costly refactoring of type hierarchies, a technique particularly useful for software evolution.

Iterators. Certain programming languages, CLU [22] and Sather [31] for example, offer an *iterator* construct for traversing encapsulated data structures in a modular manner. Most OOP languages, however, "emulate" iterators as a design pattern [15], and it is frequently necessary to manually implement an external iteration loop to achieve full generality. Using generic message forwarding, we can offer an interesting trade-off for iterators. Our technique is not quite as general as the design pattern for external iteration, but it is almost as convenient to use as having a separate language construct for iterators. Lag*oo*na's `default` method allows us to specify a strategy for forwarding messages in

```
module com.lagoona.iterator {
  ...
  class ArrayForwardIterator {
    Any[] data;
    method 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 LagoonaIterator {
    Array array;
    ...
    method void action() {
      array.forward().print();
    }
  }
}
```

Fig. 9. Iterators in Lagoona, leveraging generic message forwarding for broadcasting.

the imperative core language, with no restriction on the number of receivers involved. We can therefore implement a generic *broadcast* mechanism for messages and apply it for iteration. Fig. 9 illustrates this technique for a basic `Array` abstraction. The container 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; adding additional strategies, such as backward iteration, would be straightforward. The actual iteration is performed simply by sending a message to the iterator instance, which in turn broadcasts that message to all elements in the specified order. Note that iterator itself does not implement *any* message explicitly, but broadcasts *all* received messages to the elements in the container. Implementing a method for the forwarded message is, of course, the responsibility of the elements; if the action to be applied to all elements requires additional context information, it can be passed along as a message parameter. This approach to iterators offers a much cleaner separation between iteration code and application code than traditional iterator schemes. All code related to the iteration is located in the module exporting the container and its iteration functionality.

4. Implementation

The traditional way of demonstrating the viability of a programming language is to implement its compiler and libraries using the new language itself, a route that we have followed for Lagoona as well. However, we wanted the Lagoona compiler, instead of emitting machine-specific native code, to generate portable, safe, and verifiable code for a virtual machine. Bootstrapping the compiler involved implementing a simplified Lagoona compiler in Java, using ANSI C as an intermediate target language. The bootstrap compiler was used only during the early stages of compiler development, and as soon the new compiler written in Lagoona was complete enough to translate itself, we abandoned the bootstrap compiler and Lagoona became self-hosting. In finding a good target architecture for the Lagoona compiler, we first considered using Sun's Java Virtual Machine (JVM) [21]. However, like many other virtual machines, the JVM is primarily intended to execute programs in one particular language and offers little support for mechanisms not available in Java. In contrast to the JVM, Microsoft's .NET framework

implements the ECMA Common Language Runtime (CLR) standard, targeting a wider range of source languages including Java, C++, Visual Basic, and C#. Thus, the .NET framework seemed to be a more suitable target for a novel language like Lag*oo*na. Unfortunately, while offering a great deal of flexibility as far as the instruction set is concerned, the .NET framework offers far less freedom when it comes to type systems. To allow interoperability between programs written in different languages, type-safe ("managed") code has to use the rigid *Common Type System* (CTS). Executing Lag*oo*na on the .NET framework required either abandoning verifiability, or superimposing our object model onto the CTS. The latter is possible with surprisingly little runtime overhead as we describe in the remainder of this section.

Each message *m* in Lag*oo*na is represented by a pair of types at the CTS level. The first type, $interface_m$, is a CTS interface type containing *m* as the single abstract method. The second type, $stub_m$, is a CTS class that implements $interface_m$ and contains marshaling code. Objects of class $stub_m$ are instantiated if a message *m* cannot be directly delivered to an object and has to be handled by the default method instead. Lag*oo*na types are represented as CTS types, with implementation types corresponding to regular CTS classes and interface types corresponding to abstract CTS classes. For every method *method*(*m*) that a Lag*oo*na type provides, the ability to directly receive the underlying message *m* is indicated at the CTS level by implementing the corresponding $interface_m$ interface. Thus, the generated code can use the *isinst* ("is instance") instruction to check whether a message can be delivered directly or has to be handled by default methods. Messages sent to implementation types are directly resolved to method invocations at the CTS level, and are thus not more expensive than in other languages. Delivering a message to an interface type requires an *isinst* check first: if no immediate delivery is possible, an object of the message stub type $stub_m$ is instantiated and passed on to the default method. The Lag*oo*na compiler performs aggressive type inference to resolve as many message send operations to interfaces to message send operations to implementation types as possible. Among others, the default method of each implementation type is analyzed. Often default methods contain very simple forwarding code, i.e. forwarding the message on to one other object only. In this case, analyzing the procedural forwarding code allows us to deduce static type information and message sends are optimized accordingly. However, for more complex forwarding code this analysis does not yield any useful results and the runtime checks remain in place. In contrast to the message dispatch mechanism, Lag*oo*na's modules can easily be mapped onto the *assembly* concept found in the .NET framework. Each Lag*oo*na module is compiled to a .NET assembly consisting of a single dynamic link library (DLL). The filename of the DLL is derived from the global name of the module and is used by the runtime system to locate an appropriate DLL for each import. The current Lag*oo*na implementation cannot tolerate renaming DLLs at the file level, but in the future, we plan to use .NET's *strong names* to uniquely identify modules. It is important to note that we intend to use strong names to identify module *names* only, while .NET uses strong names to identify specific versions of assemblies. In Lag*oo*na, different versions of the same module are interchangeable by design. Lag*oo*na's executables are regular Lag*oo*na modules. If the compiler detects a type which implements the special message *com.lagoona.runtime.action*, an entry-point stub is created. This stub will instantiate an object of the relevant type and send the initial message to start executing the application.

5. Related work

Language mechanisms. Stand-alone messages can be related to the concept of *multimethods* [1]. 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 [28], 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 OOP style.

Recent work on *units* and *mixins* [8] is related to Lag*oo*na in a more interesting way. With Lag*oo*na, we have argued that programming languages for COP need to combine traits from modular languages with traits from OOP languages in a certain way. That is, 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, e.g. for the flexible creation of extended objects. Mixins provide a variation of inheritance (in the sense of subclassing) that allows derived classes to be parametrized by different base classes. However, Lag*oo*na's approach to forwarding and composition already subsumes mixins: while for mixins the base class relation is determined when units are linked, in Lag*oo*na 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 Lag*oo*na in the future.

Component models. Component models, such as COM [25], CORBA [32], and JavaBeans [34], are industry standards that claim to support COP. 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 COP—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 practise COP 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 COP paradigm is concerned.

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 [17] 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 COP [36]. In GP (and AOP), "components" are reusable and parametrized 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" that it consists of cannot be reused or updated separately from the application that they were compiled into.

6. Conclusions

The paradigm shift toward COP is not yet reflected in programming languages. In the absence of dedicated COP languages, current COP practice often employs OOP languages developed before the notion of distributed extensibility was recognized as being important. This paradigm mismatch results in unnecessary design complexity and increased maintenance overhead. We have been investigating programming languages that expressly support COP. In this paper, we presented Lag*oo*na, an experimental COP language that provides several new constructs in direct support of distributed extensibility while attempting to appear "familiar" to OOP practitioners. Lag*oo*na's object model can be viewed as another step toward eliminating the dominance of the class construct in OOP languages. Previous steps include the separation of interfaces and implementations [33] and the separation of modules and types [35], both of which are widely accepted now. We were able to implement Lag*oo*na using Lag*oo*na itself, in the context of Microsoft's .NET framework.

Historical note. The version of Lag*oo*na described here is the latest incarnation of a set of ideas tracing back to Project Oberon [37,38], which incidentally contained many of the seeds that would sprout into today's notion of COP. The Oberon System popularized using *message objects* for achieving extensibility, a technique now more commonly known as *Requests*, part of the *Chain of Responsibility* design pattern [15]. The first language construct for messages appeared in Object Oberon [29], but was curiously absent from its successor Oberon-2 [30]. Messages reappear in another proposed extension of Oberon [9], but the focus of that work was a novel form of dynamic inheritance, not components. In the first version of Lag*oo*na [10], stand-alone messages finally appear in their current form, although embedded in quite a different object model. The major improvements in our current version of Lag*oo*na are (a) the introduction of strict and blind message send operators, (b) the use of structural conformance, (c) the distinction between castable and non-castable references, (d) the removal of inheritance between implementation types, and (e) more precise semantics for generic message forwarding.

Acknowledgments

We thank Dan Berger, Rebecca Harris, Fermin Reig, and the anonymous referees for valuable comments on earlier versions of this paper. We are also indebted to Wolfram Amme, Thomas Kistler, 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] D.G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, F. Zdybel, CommonLoops: Merging Lisp and object-oriented programming, in: N. Meyrowitz (Ed), Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA, Portland, OR, November 1986, pp. 17–29.

[2] M. Büchi, W. Weck, Compound types for Java, in: Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA, British Columbia, Vancouver, 1998, pp. 362–373.

[3] L. Cardelli, Typeful programming, SRC Research Report 45, Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, May 24, 1989.

[4] L. Cardelli, P. Wegner, On understanding types, data abstraction, and polymorphism, ACM Computing Surveys 17 (4) (1985) 471–522.

[5] C. 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, Available at http://www.cs.washington.edu/research/projects/cecil/.

[6] K. Czarnecki, U. Eisenecker, Generative Programming: Methods, Tools, and Applications, Addison-Wesley, 2000.

[7] M.A. Ellis, B. Stroustrup, The Annotated C++ Reference Manual, Addison-Wesley, 1990.

[8] R.B. Findler, M. Flatt, Modular object-oriented programming with units and mixins, in: Proceedings of the International Conference on Functional Programming, ICFP, Baltimore, MD, 1998, pp. 94–104.

[9] M. Franz, Protocol extension: a technique for structuring large extensible software-systems, Software: Concepts and Tools 16 (2) (1995) 14–26.

[10] M. Franz, The programming language Lag*oona*: a fresh look at object-orientation, Software: Concepts and Tools 18 (1) (1997) 14–26.

[11] P.H. Fröhlich, Inheritance decomposed, in: A. Black, E. Ernst, P. Grogono, M. Sakkinen (Eds.), Proceedings of The Inheritance Workshop, at ECOOP, Technical Report, Information Technology Research Institute (ITRI), University of Jyväskylä, Finland, June 14, 2002, Malaga, Spain, 2002.

[12] P.H. Fröhlich, Component-oriented programming languages: why, what, and how, Ph.D. Thesis, School of Information and Computer Science, University of California, Irvine, CA 92697-3425, USA, March 2003.

[13] P.H. Fröhlich, M. Franz, Stand-alone messages: a step towards component-oriented programming languages, in: J. Gutknecht, W. Weck (Eds.), Proceedings of the Joint Modular Languages Conference, Lecture Notes in Computer Science, vol. 1897, Springer-Verlag, Zürich, Switzerland, 2000, pp. 90–103.

[14] P.H. Fröhlich, M. Franz, On certain basic properties of component-oriented programming languages, in: D.H. Lorenz, V.C. Sreedhar (Eds.), Proceedings of the Workshop on Language Mechanisms for Programming Software Components, at OOPSLA, Technical Report NU-CCS-01-06, College of Computer Science, Northeastern University, Boston, MA 02115, Tampa Bay, FL, 2001, pp. 15–18, Available at http://www.ccs.neu.edu/home/lorenz/oopsla2001/.

[15] E. Gamma, J. Vlissides, R. Johnson, R. Helm, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

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

[17] G. Kiczales, J. Lamping, C. Maeda, C. Videira Lopes, J.-M. Loingtier, J. Irwin, Aspect-oriented programming, in: M. Aksit and S. Matsuoka (Eds.), Proceedings of the European Conference on Object-Oriented Programming, ECOOP, June 1997.

[18] T. Kistler, M. Franz, Continuous program optimization: a case study, ACM Transactions on Programming Languages and Systems 25 (4) (2003) 500–548.

[19] K. Läufer, G. Baumgartner, V.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.

[20] G.T. Leavens, M. Sitaraman (Eds.), Foundations of Component-Based Systems, Cambridge University Press, 2000.

[21] T. Lindholm, F. Yellin, The Java Virtual Machine Specification, 2nd edition, Addison-Wesley, 1999.

[22] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J.C. Schaffert, R. Scheifler, A. Snyder, CLU reference manual, Technical Report MIT/LCS/TR-225, MIT Laboratory for Computer Science, October 1979.

[23] M.D. McIlroy, Mass produced software components, in: P. Naur, B. Randell (Eds.), Software Engineering: Report on a Conference Sponsored by the NATO Science Committee, 7th–11th October 1968, Garmisch, Germany, Scientific Affairs Division, NATO, Brussels, Belgium, 1969, pp. 138–155.
[24] B. Meyer, Eiffel: The Language, Prentice-Hall, 1992.
[25] Microsoft Corporation, The Component Object Model (version 0.9), October, 1995, Available at http://www.microsoft.com/COM/resources/COM1598C.ZIP.
[26] L. Mikhajlov, E. Sekerinski, A study of the fragile base class problem, in: E. Jul (Ed.), Proceedings of the European Conference on Object-Oriented Programming, ECOOP, Lecture Notes in Computer Science, vol. 1445, Springer-Verlag, Brussels, Belgium, 1998, pp. 355–382.
[27] L. Mikhajlov, E. Sekerinski, L. Laibinis, Developing components in presence of re-entrance, in: Proceedings of the World Congress on Formal Methods in the Development of Computing Systems, FM, Lecture Notes in Computer Science, vol. 1709, Springer-Verlag, Toulouse, France, 1999, pp. 1301–1320.
[28] T. Millstein, C. Chambers, Modular statically typed multimethods, in: Proceedings of the European Conference on Object-Oriented Programming, ECOOP, Lecture Notes in Computer Science, vol. 1628, Springer-Verlag, 1999, pp. 279–303.
[29] H. Mössenböck, J. Templ, R. Griesemer, Object Oberon: an object-oriented extension of Oberon, Technical Report 109, Institute of Computer Systems, Swiss Federal Institute of Technology (ETH), Zürich, Switzerland, June 1989.
[30] H. Mössenböck, N. Wirth, The programming language Oberon-2, Structured Programming 12 (4) (1991) 179–195.
[31] S. Murer, S. Omohundro, D. Stoutamire, C. Szyperski, Iteration abstraction in Sather, ACM Transactions on Programming Languages and Systems 18 (1) (1996) 1–15.
[32] Object Management Group, The Common Object Request Broker: Architecture and Specification (version 2.3.1), October, 1999, Available at http://www.omg.org/cgi-bin/doc?formal/99-10-07.
[33] A. Snyder, Encapsulation and inheritance in object-oriented programming languages, in: N. Meyrowitz (Ed.), Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA, Portland, OR, 1986, pp. 38–45.
[34] Sun Microsystems, The JavaBeans Specification (version 1.01), July, 1997, Available at http://www.javasoft.com/beans/docs/beans.101.pdf.
[35] C. Szyperski, Import is not inheritance—why we need both: modules and classes, in: Proceedings of the European Conference on Object-Oriented Programming, ECOOP, Lecture Notes in Computer Science, vol. 615, Springer-Verlag, Utrecht, The Netherlands, 1992, pp. 19–32.
[36] C. Szyperski, D. Gruntz, S. Murer, Component Software: Beyond Object-Oriented Programming, 2nd edition, Addison-Wesley, ACM Press, 2002.
[37] N. Wirth, The programming language Oberon, Software: Practice and Experience 18 (7) (1988) 671–690.
[38] N. Wirth, J. Gutknecht, Project Oberon: The Design of an Operating System and Compiler, Addison-Wesley, ACM Press, 1992.