1993

Building a Large Scale Distributed Object System for a Multilingual Programming Environment

Patrick A. Muckelbauer

Vincent F. Russo

Report Number:
93-022

BUILDING A LARGE-SCALE DISTRIBUTED OBJECT SYSTEM FOR A MULTILINGUAL PROGRAMMING ENVIRONMENT

Patrick A. Muckelbauer
Vincent F. Russo

CSD-TR-93-022
April 1993

Building a Large-Scale Distributed Object System for a Multilingual Programming Environment

Patrick A. Muckelbauer and Vincent F. Russo
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907 USA

Abstract

Object-oriented programming techniques are increasingly gaining attention as a solution to some of the software engineering problems plaguing the construction of large software projects. Unfortunately, object-oriented interfaces are usually only enforced and usable through language mechanisms, making it impossible for components in a large-scale distributed environment to interact. We are investigating the possibility of creating a runtime notion of an object's interface and allowing the dynamic querying of objects for their conformance to that interface. Our project also provides a mechanism to dynamically generate local references (proxies) to objects in different address spaces (domains) so that once this conformance is confirmed, operations can be be invoked on the object in a mechanism compatible with the client's programming language. This mechanism gives us seemingly local, native references to remote objects within a language.

1 Introduction

The use of object-oriented programming techniques is increasingly gaining attention as a potential solution to some of the software engineering problems plaguing the construction of large software projects [Mey87, HO87, JF88]. Object-oriented programming and design techniques stress modularity and data encapsulation through narrow and rigidly defined interfaces as a way of achieving low *coupling* between individual software components. Coupling measures the interdependencies between components. Low coupling is a desirable feature because it decreases the difficulty of separating, understanding, maintaining and reusing the individual components [GJM91]. It would seem, therefore, that object-oriented programming should be an ideal approach to use in the construction of large-scale distributed systems. Unfortunately, object-oriented interfaces are usually only accessible and enforced through specific languages or homogeneous systems. This severely limits the degree to which disjoint, unrelated components can interact in a wide-area, multilingual, distributed environment.

As part of the *Renaissance* project at Purdue University, we are focusing on overcoming these limitations. In particular, we are building a system which provides the ability for unrelated applications written in a variety of programming languages to obtain access to distributed objects implemented in a variety of manners. We are attempting to achieve our goals by creating a first class description of an object's interface which has a runtime realization and is independent of any programming language or system. We provide translators for this description into a variety of languages and systems.[1] These descriptions are used to generate interfaces to remote objects dynamically at runtime. They also allow flexible but complete interface specification providing both increased encapsulation and documentation.

[1] We hope to map non-object-oriented substrates into our systems.

2 Background

The main focus of object-oriented programming is to consider system components as interacting collections of data (state) that can only be accessed through a predefined set of operations. Throughout the rest of this paper we will use the term *object* to refer to such a collection, the term *method* to refer to one of the operations and the term *signature* to refer to an object's entire set of methods. Methods are *invoked* to alter or access the data of an object. In a true object-oriented system, method invocation is the only way in which an object's data is accessed. We define a *domain* to be a bounded collection of objects. We will refer to an object that invokes a method on a object in a different domain, termed a *remote* object, as being a *client* of that remote object. Furthermore, a domain is refered to as a *client* of a remote object if any one of its objects is a client of the remote object.

Fundamental to object-oriented programming is the notion of a *class* as a generator, or template, for object creation. A class specifies the interface and implementation of objects created from its template. Objects created from a given class are termed *instances* of that class. Most object-oriented languages provide a mechanism for classes to *inherit* portions of the signature and implementations of some of the methods from other classes called *parents*. This is how code sharing is implemented in class-based object-oriented languages. Classes which inherit from other classes are termed *subclasses* of those classes. A subclass' signature is a copy of the parent's signature plus any additional methods the subclass chooses to add. Likewise, a subclass can provide implementations for the new methods (or defer that decision to further subclasses) and provide new implementations of existing methods in the parent's signature. These new implementations only affect instances of the subclass. The parent's implementations are still used when the methods are invoked on instances of the parent classes. Inheritance is useful for code and interface sharing, for factoring code into common places, and for incremental development and documentation [HO87, Sny86, JF88, Mey87]. Inheritance is so useful in the context of class construction that it is viewed by many as an essential feature of object-oriented programming [Weg87].

Another feature distinguishing object-oriented programming is polymorphism achieved through late binding of method invocations to their implementations by target objects. This feature usually relies on some form of signature *conformance checking* to verify proper uses of objects. Signature S_x is said to *conform* to signature S_y (written $S_x > S_y$) if every method in S_y is also found in S_x (S_x may have additional methods as well), and for every method in S_y, each of the following conditions hold.

1. The corresponding method in S_x has the same number of parameters.
2. The parameters of S_y's version of the method conform to those of S_x's version.
3. The result of the method in S_x conforms to the result of the corresponding method in S_y.

With these rules, conformance can be viewed as a substitution rule for objects. If $S_x > S_y$ then an object with signature S_x can be used wherever one with signature S_y is expected.

For the purposes of this paper, we will distinguish between two uses of the term signature. As described above, every object has a signature defining all the methods the object makes available to its clients. This signature is usually specified by the class from which the object was instantiated. We term this the object's *concrete* signature. In addition, for each variable within a program there exist a signature that describes the interface of objects the variable can reference. We term such a signature an *abstract* signature.

Type checking object-oriented programs amounts to verifying whether the concrete signature of an object conforms to the abstract signature of variables to which it is assigned. In both statically and dynamically typed languages, a class specifies the concrete signature of its instances. Statically typed languages such as C++[Str86], Eiffel [Mey88] and Trellis/Owl [SCB+86], explicitly code abstract signatures in the program, allowing conformance to be checked at compile time. These abstract signatures are usually specified as classes; candidate objects are assumed to be instances of these classes or their subclasses. In dynamically typed languages like Smalltalk [GR83] and CLOS [DG87] variables have no declared types. Rather, conformance checking is deferred until method invocation time.

In our work we focus primarily on clients of objects distributed throughout different domains, and not on the way the objects themselves are implemented. For this reason, we feel that while classes are an invaluable object implementation tool, they are an inappropriate mechanism for describing distributed objects to clients. We feel

using a signature-only description mechanism over classes is essential to reduce the coupling between objects and clients and increase the scalability of the distributed object system. Rather than requiring the client code to know the concrete signature of an object in order to access the object, we allow programmers to specify an abstract signature they expect the object to have. We allow this specification in a language independent manner which is mapped by a translator into the target language.

As a simple motivation for our approach, consider reading files from a remote system providing file objects to clients. The abstract signature of a program using file objects would likely consist of methods for reading (**read**), writing (**write**) and positioning (**seek**) the files. However, when writing the class for file objects, the programmer would likely provide methods for various system implementation functions and system information function as well. These methods would not be of interest under normal circumstances to the average client of file objects but might be important for proper system operation. Examples of such methods would be those to access or update information about the position of the file on permanent storage, or to return information about update and modification times for the individual files. The desire to keep interfaces narrow to decrease coupling between components should make it clear that these additional methods are superfluous to the average client program. Requiring or allowing their existence to be made apparent to client programs only increases the coupling between the client program and the implementation of the file objects. Ideally, it should be possible for the client program writer to specify *exactly* the abstract signature he or she needs the remote file object to have and for the system to check the conformance to this signature at the time a reference to a remote file object is created. It is uninteresting to the programmer of the client code how the methods in these signatures are implemented and what additional methods are available.

3 Approach

Our approach is to create a first class description of a signature which has a runtime realization and is independent of any programming language or system. These descriptions are used to generate interfaces to remote objects dynamically at runtime. Our description supports signatures to describe objects and, in addition, supports a set of *primitive* types. Primitive types are simple data without methods. Types such as **integer**, **character**, **array of** <type>, and aggregates (structures) are examples. For the remainder of this paper, the term *object* unqualified will be used to refer to objects with methods and the term *data* to refer to instances of primitive types. Providing for simple data items is motivated by the success of languages like C++ at providing support for object-oriented programming while not penalizing traditional operations such as integer arithmetic, and by the observation that many method arguments are integers, characters or strings.

A client program using our description language must be written in a style which accesses all distributed objects through method invocations. Also, all objects are passed by referenced and not by value. We feel that passing objects by value is really an object mobility issue and not a conformance issue. Passing objects by value would involve moving class and implementation information between domains. This information is intentionally outside the scope of our interface descriptions. All data in our system will be passed by value, result, or value-result to methods.

Every object accessible to clients using our description scheme has a signature describing the name, return type, and argument types of every method the object will accept. For example a buffer object might have the following signature:

```
signature Buffer =
     size() : integer;
     characterAt( integer ) : character;
     putCharacterAt( character, integer ) : void;
end
```

Signatures can be used to specify the conformance required for arguments of other signatures as well. For example, the `Buffer` signature can be used as a description of the arguments to methods of a `File` signature.

```
signature File =
```

```
    read( Buffer ) : integer;
    write( Buffer ) : integer;
end
```

Because of the importance of determining the type of an object, all objects implicitly support the `signature` method. This method returns a reference to a `Signature` object describing the object's concrete signature. Signatures are themselves first class entities. Each signature is accessible through the signature `Signature` described below.

```
signature Signature =
    conformsTo( Signature ) : boolean;
    ....
end
```

The `conformsTo` method tests whether the signature conforms to another signature.

Our approach focuses on providing a first class, runtime representation of signatures that can be compared for conformance and integrated with language notions of objects and methods where present. Each object participating in our system is able to be queried for its signature object which provides a complete description of an object's interface. We provide the ability for objects representing programmer defined signatures to be created and checked for conformance with the signature objects provided by remote objects. Initially, we are integrating this representation into the C++[Str86] programming language so that the C++ type checking system can check statically for correct uses of remote objects[GR91]. However, we do not wish to limit ourselves solely to object-oriented languages. We will also integrate our notion in a procedural manner into traditional programming languages.

In a distributed object environment, accessing objects in different domains requires some sort of remote method invocation mechanism. *Proxies* [Sha86] are a well accepted solution to this problem. A proxy is a local representation of a remote object and maps the language notion of procedure call or method invocation transparently into a network transfer to the remote object and then back. We call the object represented by the proxy its *principal*. Proxies must transfer arguments to, and results back from, the node containing the principal. In our system, we generate proxies dynamically at the time object references (variables) are bound to actual objects. This generation is based on, and guided by, the signature of the object.

3.1 Object Attributes

Besides simply specifying the interface provided by objects with signatures, we allow objects in the system to carry *attributes*. The goal of supplying attributes is to provide additional information about *individual* objects not directly attainable from their concrete signatures in an attempt to improve efficiency. As attributes are discovered, they can be used to dynamically affect a proxy's implementation to take advantage of this new information. Attributes are similar to POOL's properties [Ame90], except we use attributes as an object discrimination scheme rather than a class or signature discrimination scheme. We anticipate attributes initially being as simple as (*name*, *value*) pairs, perhaps evolving into more complicated descriptions.

Attributes are used to create proxies that attempt to gain improved efficiency. In the absence of attributes, all proxies degenerate into simple stubs. A stub relays every call to the principal and is responsible for marshaling arguments to and from the principal. We use attribute information to both reduce network traffic and improve remote method invocation response time. Consider the following signature for files:

```
signature File =
    read( Buffer ) : integer;
    write( Buffer ) : integer;
    numberOfBytes() : integer;
```

No special caching information may be inferred from this signature. However, a per-object attribute for read only files could provide semantic information indicating that the return value for the method `numberOfBytes` is constant and could encode the value. For a read only file the proxy implementation of the method `numberOfBytes` can be resolved locally, reducing network traffic and improving method invocation response time.

3.2 Name Servers

Besides a mechanism for remote method invocation, a distributed system needs a naming mechanism to provide clients with references to remote objects in the first place. This is similar to the way that references to servers must be obtained in message-passing systems. One solution is to provide every client with a predefined reference to a *name server* object that maps symbolic names to references. Queries to this object return references to other objects which in turn provide specific services.

Name servers may impose static or dynamic typing on object references. In a dynamically typed object-oriented system, queries to a name server require only the name of the object and return a reference to the object if it exists no matter what the type of the object. Type checking is deferred until the object's methods are invoked. Statically typed systems do not allow method invocations to untyped references. The type of the object being referenced must be confirmed before method invocations are allowed. Support for static typing can be achieved in two ways. First, name servers can provide untyped references to objects, but the references are not usable until they are *narrowed* to a particular type. After narrowing, a new reference is constructed that may be used as the target of a method invocation with no further checks. If the object is not of the proper type, the narrow operation should fail. Second, untyped references can be prohibited. The type of the object being looked up can be included as an argument to the name server query operation. The name server will only return a reference if an object of the given name exists *and* it is of the type requested. In this way, once a reference is obtained, it can immediately be treated as a typed reference and method invocations can proceed normally.

Each client will be provided with a predefined set of references to name server objects. Because we provide runtime type information, our system can support both static typing and dynamic typing of object references. Dynamic typing decreases lookup time while increasing method invocation time, and conversely, static typing increases lookup time while decreasing method invocation time. Because lookup time is a one time cost whereas method invocation is not, we feel it is more important to optimize method invocation times. For this reason we chose to use static typing in our system.

Each name server is accessible through the signature `NameServer` described below.

```
signature NameServer =
    lookup( String, Signature ) : Object;
    ....
end
```

The `lookup` method returns a reference to an object if the given name exists and it is of the type requested.

3.3 Details

All remote method invocations occur by invoking a method on a proxy. The proxy hides all the details about the remote access to the principal. If the proxy has discovered attributes which allow the operation to be processes locally then it does so, else it must invoke the corresponding method on the principal, an operation refered to as a remote method invocation. A remote method invocation proceeds as follows. The client marshals data into a request message, sends it to the remote domain, and waits for a reply. The remote domain receives the request message, unmarshals the arguments, proxies any object references, invokes the method on the target object, marshals the results into a reply message, and sends it to the client. When the client receives the reply message, it unmarshals the results and generates proxies for any newly acquired objects.

A proxy relies on a transport protocol for providing the underlying mechanism for moving data between domains. For purposes of modularity, we assume the transport protocol is responsible for providing reliable communication between these domains, hiding all the details about lost, duplicate, or out-of-order messages. This design makes it easier to debug, modify, and/or replace the underlying transport mechanism. In addition, it is easy to support multiple transport mechanisms with this design. For example, if the domains are simply separate address spaces on a single node, the transport mechanism may be a simple shared memory scheme or a light-weight remote procedure call mechanism built on top of shared-memory. If the domains are on physically separate nodes, a true remote procedure call over a network is used [BN84].

Our system will include a machine-independent data representation for primitive types to support heterogeneous architectures; objects will be represented as handles. Handles provide a flat, global name space for objects in the system.

3.4 Current Status

Our ideas are implemented in the *Renaissance* system. *Renaissance* is an object-oriented[Rus91] multiprocessor operating system fully designed and constructed using object-oriented techniques. It is a reinvestigation of the ideas and algorithms learned in the *Choices* system from the University of Illinois at Urbana-Champaign[Rus91, CRJ87], and an extension of those ideas into a distributed object environment. *Renaissance* is intended to be a platform upon which to conduct distributed and multiprocessor operating system research. Extending the *Choices* goal of providing system objects to applications transparently, the goal of *Renaissance* is to provide transparent access to remote objects.

The first phase of this project was to create a signature-based interface to the *Renaissance* kernel. This phase involved making all application-accessible kernel objects support a method which returns the object's concrete signature. We also provided the ability for applications to create objects representing abstract signatures that were used in conformance checking.

In addition to these modifications, each domain (address space) in *Renaissance* including the kernel domain was provide with a dispatch routine which allowed dynamic invocations of its objects' methods.

Phase two of this project will be to add support for distributed objects. This support will be provided by a simple remote procedure call protocol to access the remote domain's dispatch routine. The protocol will include a machine independent representation for primitive types and support for the exchange of object attributes.

Parallel with the latter half of phase two, we will begin phase three, the integration of signatures into an object-oriented programming language. C++ will be used due to its availability and our previous work on integrating the signature concept into the language [GR91].

3.5 Evaluation

We will measure the success of our system by two criteria:

1. The level of transparency achieved in our system.

2. The efficiency of our model.

We fully expect to encounter numerous hurdles and unforeseen problems during the remainder of this project. One problem we do foresee is in the area of garbage collection and storage reclamation. Many object-oriented languages rely solely on a garbage collector for memory reclamation. Unfortunately, many of these garbage collectors were designed strictly for non-distributed languages and do not handle cross-references between different domains. Our desire to remain multilingual will serve only to aggravate the problem.

Security, another important aspect of a distributed system, allows the creator, or owner, of an object to limit access to the object to a set of authorized clients. Although we have primarily concentrating on designing a remote method invocation mechanism, we feel our system provides the flexibility necessary for building and testing security systems and plan on pursuing this area of research in the future.

4 Related Work

RPC-based Systems. Clients in remote procedure call (RPC) based systems [BN84] such as SUN RPC [Sun85] acquire system services by invoking local functions that transparently access remote services. Most RPC-based systems provide the notion of a program and a set of procedures to call within a program which is analogous to an object with methods. However, RPC-based systems do not support objects as a first class type, and therefore, they are not allowed to be passed as arguments or returned as results. For this reason RPC-based systems are not well suited for modeling object-oriented applications.

Interface Description Translators. A common approach to address some of the goals we address in our work is to build an interface description translator like Matchmaker [JR86]. These systems provide a object-based interface description language to traditional distributed systems like Mach. Matchmaker's proxies are generated statically at compile time from the source of the description. They provide a convenient mechanism for varying programming languages to incorporate remote objects. However, Matchmaker does not provide a runtime representation of the interfaces described, but rather, requires the client to know an object's signature to access it. This is achieved in MatchMaker but requiring the client and implementor of an object to share the interface descriptor for that object. This sharing is necessary to guarantee proper use of an object, but has the adverse effect, of increasing the coupling between the client and the object, limiting the overall system's scalability. In our system, this sharing is not required because the run-time type information is available to guarantee proper use of an object.

Traditional Distributed Systems. Clients in traditional distributed systems such as V [Che88, Che84], Mach [A+86], and Chorus [RAN88] acquire system services from servers by explicitly sending messages to ports or processes. While the servers in such systems can be considered objects and message sending analogous to invoking methods, such systems do not provide a runtime representation for signatures and therefore cannot perform type checking at the time objects are bound to references. All type checking must be done at message send time.

Distributed Object Languages and Systems Distributed object languages, such as Argus [Lis87], Smalltalk [Ben87], and Emerald [RTL+91], and distributed object systems, such as Clouds [DLA87] and Eden [LLA+81], not only provide a notion of objects and type conformance but are also provide features such as concurrency and atomicity, replication, persistence, fault tolerance, and migration. Unfortunately, the requirements placed on these systems to support these features makes it difficult for them to scale and interoperate with one another. In our systems, these features are consider object-specific and should be provided by the implementation of the object and hidden from the client behind the object's interface. Our work emphasizes the accessing of remote objects independent of their particular implementation and provides a framework in which these systems can interoperate.

5 Conclusion

Our work differs from existing research in the area in its use of runtime type information for achieving signature-based polymorphism. The use of signature-based polymorism is essential in achieving the low coupling between software components necessary for a scalable, maintainable distributed system. We feel the use of class-based polymorphism used by existing systems increases coupling and limits their effectiveness in a large-scale distributed environment.

To the extent that our research is successful, it will provide programmers of distributed systems an object-oriented framework within which to construct such systems. In addition, it will bring a language-independent notion of object and type (signature) closer to reality. Finally, providing the ability to separate implementation details from interface details in distributed systems will allow the construction of less coupled distributed software.

References

[A+86] Mike Accetta et al. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the USENIX Conference*, pages 93–111, June 1986.

[Ame90] Pierre America. A Parallel Object-Oriented Language with Inheritance and Subtyping. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 161–168, 1990.

[Ben87] John K. Bennett. The Design and Implementation of Distributed Smalltalk. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 318–330, 1987.

[BN84] Andrew Birrell and Bruce Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1), February 1984.

[Che84] David R. Cheriton. The V Kernel: A Software Base for Distributed Systems. *IEEE Software*, 1(2):19–42, April 1984.

[Che88] David R. Cheriton. The V Distributed System. *Communications of the ACM*, 31(3):314–333, March 1988.

[CRJ87] Roy Campbell, Vincent Russo, and Gary Johnston. The Design of a Multiprocessor Operating System. In *Proceedings of the USENIX C++ Workshop*, pages 109–123, 1987. Also Technical Report No. UIUCDCS–R–87–1388, Department of Computer Science, University of Illinois at Urbana-Champaign.

[DG87] L. G. DeMichel and R. P. Gabriel. The Common Lisp Object System. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '87)*, 1987.

[DLA87] Partha Dasgupta, Richard LeBlanc, and William Appelbe. The Clouds Distributed Operating System:Functional Description, Implementation Details, and Related Work. Technical Report GIT-ICS-87/42Functional Description, Implementation Details, and Related Work, Georgia Tech, 87.

[GJM91] Carlo Ghezze, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[GR83] Adele Goldberg and David Robison. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.

[GR91] Elana D. Granston and Vincent F. Russo. "Signature-Based Polymorphism for C++". In *Proceedings of the USENIX C++ Conference*, 1991.

[HO87] Daniel C. Halbert and Patrick D. O'Brien. Using Types and Inheritance in Object-Oriented Programming. *IEEE Software*, pages 71–79, September 1987.

[JF88] Ralph E. Johnson and Brian Foote. Designing Reusable Classes. *The Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[JR86] Michael B. Jones and Richard F. Rashid. Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 67–86, 1986.

[Lis87] Barbara Liskov. Distributed Programming in Argus. Technical Report Programming Methodology Group Memo 58, MIT, October 1987.

[LLA+81] E. Lazowska, H. Levy, G. Almes, M. Fischer, R. Fowler, and S. Vestal. The Architecture of the Eden System. In *Proceedings of the ACM Symposium on Operating System Principles*, pages 148–159, 1981.

[Mey87] Bertrand Meyer. Reusability: The Case for Object-Oriented Design. *IEEE Software*, pages 50–64, March 1987.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[RAN88] M. Rozier, V. Abrossimov, and W Neuhauser. CHORUS-V3 Kernel Specification and Interface, Draft. Technical Report CS/TN-87-25.10, CHORUS Systems, February 1988.

[RTL+91] Rajendra K. Raj, Ewan Tempero, Henry M. Levy, Andrew P. Black, Norman C. Hutchison, and Eric Jul. Emerald: A General-Purpose Programming Language. *Software – Practice and Experience*, 2(1):91–118, January 1991.

[Rus91] Vincent F. Russo. *An Object-Oriented Operating System.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1991.

[SCB+86] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An Introduction to Trellis/Owl. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 9–16, 1986.

[Sha86] Marc Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. In *Proceedings of the 6th. International Conference on Distributed Computer Systems*, May 1986.

[Sny86] Alan Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 38–45, 1986.

[Str86] Bjarne Stroustrup. *The C++Programming Language.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[Sun85] Sun Microsystems. *Networking on the SUN Workstation*, 1985.

[Weg87] Peter Wegner. Dimensions of Object-Based Language Design. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 168–182, 1987.