OROS: Toward a Type Model for Software Development Environments

William R. Rosenblatt[†]
Jack C. Wileden[†]
Alexander L. Wolf[‡]

COINS Technical Report 89–67
July 1989

[†]*Software Development Laboratory*
Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

[‡]AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, New Jersey 07974

This report to appear in
ACM Object–Oriented Programming Systems, Languages and Applications Conference
New Orleans, LA., October 1989

This work was supported by grant CCR–8704478 from the National Science Foundation with cooperation from the Defense Advanced Research Projects Agency (ARPA Order No.6104).

OROS: Toward a Type Model for Software Development Environments

William R. Rosenblatt[†]
Jack C. Wileden[†]
Alexander L. Wolf[‡]

[†]*Software Development Laboratory*
Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

[‡]AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, New Jersey 07974

Abstract

Three important goals of next generation software development environments (SDEs) are extensibility, integration and broad scope. Our work on OROS is predicated on the hypothesis that a type model, incorporated into an environment's object manager, can contribute to achieving those goals. This paper reports on an attempt at applying object-oriented typing concepts in the domain of software development environments. We believe that the result is a type model that has properties of interest both to software environment builders and also to builders and users of object-oriented systems in general.

1 Introduction

This paper describes OROS [19], an object-oriented type model[1] that we have developed as part of our research on object management for next generation software development environments (SDEs). This research is being done as part of the Arcadia project [25], a collaborative software environment research program encompassing groups at several universities and industrial organizations. The objective of Arcadia is to develop advanced software environment technology and to demonstrate this technology through prototype environments.

Three important goals of next generation software development environments, such as those envisioned by Arcadia, are extensibility, integration and broad scope. In particular, Arcadia environments are intended to be *extensible* in order to support experimental investigation of software process models and evaluation of novel tools in the context of a complete environment. At the same time, Arcadia environments must remain *integrated*, both externally, to aid users of the extended functionality, and internally, to facilitate tool cooperation, environment maintenance and further extension. Arcadia environments are also intended to be *broad in scope*, i.e., to support a wide variety of development activities, not merely monolingual program development and execution, and therefore to include many different kinds of tools and objects.

These goals require that Arcadia environments facilitate the addition, modification and replacement of any and all kinds of *environment components*, which are such things as tools, management data or process descriptions. This in turn requires the availability and control of detailed, manipulable information concerning

[1] In this paper, we distinguish between *type system*, a specific collection of types developed for use in some application (such as a particular software environment), and *type model*, a framework or mechanism for defining type systems.

the properties of *software product components*, such as specifications, designs, code, test data, or documentation.

Our work on OROS springs from the belief that environment extensibility and integration will be enhanced by treating both environment components and software product components as instances of types. In particular, a type system provides a classification scheme that can explain the roles of various kinds of components and their relationships to other kinds of components. If the classification scheme can be enforced, by checking that all components are proper instances of known types and that their uses are always in accordance with the properties associated with their types, then it can greatly facilitate reliability and modifiability of an environment. Naturally, it must be possible for environment and tool builders to define their own type systems if this approach is to be a help rather than an impediment. Similar observations, as applied to programs rather than environments, have motivated the increasingly widespread use of abstract data typing and object orientation in modern programming languages and database systems.

This paper reports on an attempt at applying object-oriented typing concepts in the domain of software development environments. Most of the other efforts toward environment type models with which we are familiar have essentially adopted or adapted some existing type model, either from a programming language or from a database schema language. In contrast, the OROS model has been developed through a top-down requirements definition effort, and has been specifically tailored to support environment object typing needs. We believe that the result is a type model that has properties of interest both to software environment builders and also to builders and users of object-oriented systems in general.

The next section of the paper outlines the OROS type model and provides brief examples of its major features. Due to space limitations, we refer readers interested in detailed examples to technical reports [19, 27]. Subsequent sections compare OROS with related work and describe our experiences with experimentation and implementation. We end with a summary and statement of future research directions.

2 The OROS Type Model

In developing a type model, one must settle upon:

- a set of *primitive types*, from which all others are constructed; and
- a *type definition mechanism*, which allows the type system to be built up inductively from the primitive types.

As we indicated in the introduction, OROS is motivated by our view of significant concerns for software engineers building and experimenting with environments. In the remainder of this section we elaborate on the observations that we have made concerning typing requirements for environment object management and the primitive types and type definition mechanism of OROS that have resulted from those observations.

2.1 OROS Primitive Types

In selecting a set of primitive types for use in environments, we began by asking ourselves what kinds of "things" are of primary interest to environment builders. Our observation was that there are three fundamental kinds of "things" that compose environments. Using the term *entity* to denote the most general, all-encompassing category for "things" (i.e., everything is viewed as an entity), we identified these three fundamental kinds of entities:

objects — essentially passive pieces of information about a software product or environment, such as management data, design documents or source-code modules;

relationships — conceptual connections among entities, such as the connection among all source-code modules belonging to some software product; and

operations — manipulations that can be performed on entities, such as compiling a source-code module.

The implication of this is that, in the OROS type model, everything in an environment is an instance of type `entity`. Further, everything is also an instance either of type `object` or of type `relationship` or of type `operation`. Thus, these three types are *subtypes* of type `entity`, according to the notion of subtyping described in Section 2.3.

A specific software environment's type system would include, in addition to the OROS primitive types, a set of specialized subtypes derived from them. For example, something that is an instance of type `object` might also be an instance of the subtypes `text_object`, `source_code` and `Ada_source_code`, each of these subtypes being more specialized and restrictive than its predecessor. In fact, we envision the definition, maintenance, modification and refinement of a type system to be a significant part of the work of an environment builder or experimenter who is using Arcadia environment technology.

Our second, related observation concerning the primitive types for an environment type model was that all three of the types introduced above are of equal significance. That is, we do not believe that objects are of greater importance than relationships, nor any other such prioritization, from the perspective of environment builders and experimenters. This provides the rationale for the name OROS, which derives from Object, Relationship and Operation System[2].

The inclusion of operation and relationship types in OROS, as well as its co-equal treatment of all three primitive types, sets OROS apart from most other object-oriented type models, as we explain in Section 3.

2.2 Type Definition Mechanism

Given our view of the appropriate primitive types and their co-equal status, we next considered properties needed in a type definition mechanism. This consideration was also driven by some observations.

The first observation was that, although abstract data types have traditionally been defined in terms of the operations that can be applied to instances of them, relationships are equally important determiners of a type's properties — particularly in the SDE domain. For example, the fact that an object-code module is related to some source-code module by the `compiled-from` relationship may be at least as important a property of type `object-code` as the fact that the `link` operation can be applied to instances of that type.

It is also interesting to note that relationships can subsume the notions of *attributes* and *components* often used in object-oriented type models. In fact, many such schemes use "attribute" or "property" as a kind of syntactic shorthand for binary, unidirectional relationships between objects [2, 10, 30]. For example, a type `compiler` may have a component of type `parser`; OROS could be used to model this via a `parser-of` relationship. OROS supports relationships among arbitrarily many entities of arbitrary types; attribute- or property-like relationships are merely a particular case.

A second observation along these lines was that the set of applicable operations and relationships should not only serve to define object types, but is equally significant in defining relationship and operation types. That is, it should be possible to define relationship types in terms of the relationships in which they can participate and the operations that can be applied to them, and similarly for operation types. Thus, the upshot of the preceding observations is that a type, be it an object type, a relationship type or an operation type, is defined (at least in part) by the set of all operations and the set of all relationships applicable to entities of that type.

[2] "Oros" (Όρος) is also Greek for "mountain". Oros is one of the prominent features of the Arcadia landscape.

The following is a simple example of an OROS type definition; the reader is referred to [19] for more detailed examples. This example shows how a typical object type from the SDE domain can be defined in terms of associated operations and relationships. Although syntax is not a major concern at the level of generality of OROS, this and subsequent examples employ an *ad hoc* type definition syntax.

```
type source_code is
    parents:
        object
    operations:
        edit, compile
    relationships:
        source_to_object_code,
        design_to_source_code
end
```

The "parents" field of this definition specifies that type `source_code` inherits operations and relationships from type `object`, i.e., that `source_code` is an object type. Type `object` might have operations such as `create` and `delete`; these operations also apply to instances of type `source_code`. In addition, source code objects can be edited and compiled; also, they participate in relationships with object-code modules and with design documents.

We have also observed that operation and relationship type definitions should include interface descriptions: operations in terms of their parameter lists, relationships in terms of the types of their tuple elements. We refer to both of these as *signatures*; both are essentially lists of name/type pairs. Thus, for example, the `compile` operation on the above `source_code` type might have the following type definition:

```
type compile is
    parents:
        operation
    signature:
        ( src  : in source_code;
          obj  : out object_code )
end
```

Notice that, for signatures of operation types, we use parameter modes (in, out, inout) in a manner similar to Ada. The definition of the relationship type `design_to_source_code` might look like this:

```
type design_to_source_code is
```

```
        parents:
            relationship
        signature:
            ( des :  design_document;
              src :  source_code )
    end
```

The final observation we have made about type definition relates to our goal of providing environment extensibility. As new capabilities are added to an environment based on a type model, new types must also be added; these types should integrate readily with the rest of the environment. We have found that environments often contain entities that are aggregations of other entities, e.g., stacks, arrays, lists, etc. Extending an environment, then, often involves modifying the types that describe such aggregations so that they can be used with new types. The type model should make it easy to perform such modifications.

One way to accomplish this is via *parametric types*, e.g., `stack[integer]`. Several object-oriented systems, such as Vbase [2], Napier88 [10], Galileo [1], and Trellis/Owl [22] have included similar mechanisms. In OROS, type definitions can optionally include arbitrarily many *formal parameters*, which are declared at the start of the definition and then can be used throughout the rest of the definition. *Parameterized types*, then, are instantiations of parametric types with existing types as *actual parameters*. Formal parameters of parametric types can also include restrictions on what kinds of types can be used as actual parameters.

For example, the following definition of type `relation` can be used to create relational-database-like objects:

```
    type relation[R : relationship] is
        parents:
            object
        operations:
            insert[R], delete[R],
            select[R], ...
    end
```

`Relation` can be parameterized by any relationship type (i.e., type `relationship` or any subtype thereof). This definition implies the necessity of parametric operation types such as the following:

```
    type insert[R : relationship] is
        parents:
            operation
        signature:
            ( rel  : inout relation[R];
              item : in R )
    end
```

Using the relationship type definition above, we can create a parameterized type definition for relations of `design_to_source_code` tuples:

```
    type design_to_source_code_rel is
        relation[design_to_source_code]
    end
```

Creating this definition would imply the creation of appropriately parameterized versions of `insert[]`, `delete[]`, etc. Note, of course, that this mechanism can be used to easily define "relations" over newly-added relationship types.

2.3 Subtyping and Semantic Intertype Relationships

We have considered several kinds of relationships among types (or *intertype relationships*) throughout the design of OROS[3]. We separate intertype relationships into two basic categories: *semantic* intertype relationships, which are relationshps among types' behaviors (such as subtyping), and *structural* intertype relationships, which are relationships among types' definitions (such as inheritance). A characteristic of OROS that differentiates it from other object-oriented type models is that it includes both kinds of intertype relationships and also separates them carefully.

Subtyping is included in such systems as Emerald [5], Galileo and Trellis/Owl. The basic idea is that, given two types A and B, A is a subtype of B if an instance of A can be used wherever an instance of B is required, but not necessarily vice versa. In the SDE domain, for example, one might like to specify that a `design_document` can be treated as a `text_object` (e.g., it can be printed or put through a text formatter).

Subtyping among object types has been treated formally by various authors, notably Black et al. [4], whose notion of types includes operations with arguments and results (i.e., with signatures). However, it does not include relationships associated with object types, nor does it include operation and relationship types as such. Thus, their results could not be directly applied to the full OROS model. Because these features of OROS are mostly novel, there is little or no precedent for the notion of subtyping or "subtyping-like" semantic intertype relationships that take them into account. In fact, we conjecture that one fixed set of rules may not be flexible enough to cover all such intertype relationships that we

[3]Note that the concept of intertype relationship should not be confused with the type `relationship`, introduced in Section 2.1, or a relationship associated with a type definition, introduced in Section 2.2.

feel are interesting in the SDE context. A major current focus of our research is to discover the nature of these intertype relationships and what sorts of rules are necessary to express them.

2.4 Inheritance and Structural Intertype Relationships

Like the designers of other object-oriented systems, we believe that inheritance provides a useful shorthand for type definition as well as a convenient way to represent a type system's organization. However, we have observed that inheritance is just one of an entire family of interesting relationships among type definitions. As mentioned above, we refer to these as *structural* intertype relationships. Space constraints permit only a brief synopsis of our work on them here.

Structural intertype relationships provide a way to describe and/or control differences among type definitions that is more precise than existing inheritance mechanisms. As an example: Smalltalk [13] provides single inheritance along with the ability for a subclass to override inherited operations (methods) as well as to add new methods. In our terms, this corresponds to three kinds of structural intertype relationships:

inherit — the superclass' definition is copied, and there is a guarantee that any subsequent changes in the superclass' definition are reflected in the subclass

add — the subclass can add a method

delete — the subclass can delete a method and replace it with a new one by *add*ing it[4].

In OROS, the *add* and *delete* relationships can apply to a type definition's operations, relationships, or signature elements.

We are experimenting with creating a small, yet complete, set of structural intertype relationships for OROS and its various implementations. For example, the presence of subtyping in a type model (as described above) suggests the need for the structural intertype relationships *specialize* (i.e., replace with a subtype) and *generalize* (i.e., replace with a supertype). We expect to report further on this in the future; for now, we note the similarity of this to our previous work on describing precise interface relationships between software modules [28, 29].

[4]Note that it is not possible to simply *delete* a method from a Smalltalk subclass.

3 Related Work

Because the OROS type model is intended specifically for the specification of object management systems for software development environments, we consider it in light of previous work in this application area. As with all such work, of course, we acknowledge such influences as traditional database systems (e.g., [8, 20]), object-oriented systems (e.g., [13, 15, 24]), type theory (e.g., [6, 9]) and integrated SDEs (e.g., [7, 12, 18]).

Requirements for SDE object managers have been suggested by Bernstein [3] and Katz [14], among others. Such requirements have been combined with results in the above areas to produce object management systems and type models that, like OROS, have been intended specifically for SDE applications. Thus it is not surprising that, among the type models of which we are aware, those that most closely resemble OROS have all been developed with similar applications in mind. These systems/models include Encore [30], Napier88, Vbase and Gaia [26].

Like all of these, OROS conforms to the definition of "object-oriented" (e.g., [9] or [16]), because it includes notions of abstract data types and inheritance. OROS' primary additions to the "ADTs + inheritance" formula are:

- inclusion of operation and relationship *types*,
- co-equal treatment of objects, relationships and operations, and
- separation of structural (e.g., inheritance) and semantic (e.g., subtyping) intertype relationships

Of the above systems, Napier88, Vbase and Gaia include notions corresponding to operation types (called "procedures" in Napier88). Operations in Gaia (i.e., instances of its "operation" class) encompass Ada procedures and functions, whose argument lists are as general as OROS operation type signatures. However, Vbase operations and Napier88 procedures only allow one value to be returned, which is not as general. This generality of OROS signatures is useful, for example, for modelling software tools that produce more than one output (such as compilers that produce cross-reference listings in addition to object code). Furthermore, none of these other systems explicitly address subtyping among operation types, while OROS does.

OROS supports relationships as parts of all type definitions. Encore supports binary, unidirectional relationships as "properties" of object types, while Napier88 supports them as record fields. Furthermore, OROS includes relationship types; like Rumbaugh [21], we feel that relationships should be supported as separate entities. For example, relationship instances (or *tuples*) are ideal for representing dependencies among software objects in the manner of the MAKE tool of the Unix[5] operating system [11]. Such relationships might involve instances of several different types; therefore, attribute or record-field schemes cannot model them directly. Integrating this kind of information with an environment's underlying type system, and not relying on separate tools like MAKE, has been shown to promote tool integration, reliability and modifiabilty [7].

Like Encore and Napier88, Vbase and Gaia support relationships as parts of type definitions, but the latter two also support relationship types. However, Vbase's are limited to binary (one-to-one) relationships, while Gaia's can be one-to-one, one-to-many or many-to-one, where "many" means "a set of instances of the same type." OROS supports all of these plus relationships among arbitrarily many entities of arbitrary types (necessary for representing MAKE-like information, as stated above). OROS also supports subtyping among relationship types; again, this is not explicitly addressed by any of these other systems.

All of the other systems treat object types as primary except Vbase. Vbase's taxonomy of primitive types includes several types that have co-equal status; three of these correspond to OROS' object, relationship and operation types.

Finally, OROS includes a careful separation of the notions of inheritance-like "structural" and subtype-like "semantic" intertype relationships. Moss and Wolf [17] advocate this approach and suggest that most existing object-oriented systems intertwine the two concepts; Snyder [23] also discusses how the two concepts affect each other. Galileo includes the separate concepts of *type* (intensional specification) and *class* (extensional specification); it uses inheritance in the context of classes (i.e., subclass/superclass) and subtyping in the context of types. In contrast, OROS is purely a *type* model, but it incorporates both kinds of intertype relationships and treats them separately.

4 Experience and Implementation

We have gained some experience with the OROS type model by taking two very different approaches to experimentation with and implementation of it.

4.1 Prototype User Interfaces

The first of these approaches involves experimentation with support tools that define user interfaces to OROS.

[5]Unix is a registered trademark of AT&T.

We believe that a powerful user interface can dramatically increase the utility of a type model to an SDE builder or maintainer. To date, we have built two interactive user interfaces (*browsers*) for OROS, one in Prolog and the other in Lisp. Both of these maintain type systems by allowing type definitions to be created, modified and deleted; both also perform full consistency checking.

We feel that such browsers are particularly useful in the SDE domain. As stated in the introduction, extensibility and integration are major goals of the Arcadia project. In an environment based on OROS, a major activity related to these goals is the addition of new types and the modification of the type system to accommodate them. Because such a type system may well have hundreds of types, an appropriate user interface (e.g., with powerful graphics) can allow an environment maintainer to concentrate on large parts of the type system without having to wade through the details of individual type definitions.

Just as importantly, we feel that a browser should perform moderately "intelligent" tasks that assist in environment type system maintenance. For example, when new functionality is added to an environment, the browser could determine where newly-defined types fit best in the existing type system by finding types that are most similar to them. This information, in turn, can suggest which existing environment components can be reused to support the new functionality and what new capabilities need to be developed. The inferences that must be performed to make such determinations would make use of the semantic and structural intertype relationships described in Sections 2.3 and 2.4.

4.2 Interoperability Support

The major goals of Arcadia environments imply that they must support *interoperability*, which we define as the ability of two or more tools to communicate or work together despite having been written in different languages. In contrast to the Unix operating system, for example, in which interoperability is hampered by the need to do *ad hoc* translation of structured data into byte streams (and vice versa), we have found through experimentation that a sufficiently rich type model such as OROS can go a long way toward providing interoperability in an SDE.

We have been using a subset of OROS in an experiment on near-term practical use of type model features to support interoperability. This subset, described in detail in [27], was chosen specifically to support interoperability of tools written in Ada and C, the predominant languages used to build Arcadia prototype software. It includes all the basic features of OROS, uses simplified sets of rules for inheritance and subtyping, includes "simple" object types (such as integer) that correspond closely to built-in types of the above languages, and embodies a small set of primitive type implementation semantics.

This OROS subset has been used as a basis for an experiment, also detailed in [27], on the interoperability of data between the domains of two Ada-based tools that employ somewhat different type systems. One of these tools manages persistent graph structures, while the other manages relations in the manner of relational databases. In the experiment, OROS-subset type definitions were developed and used to guide the hand-coding of Ada packages that enable graph structures to be expressed in terms of relations and vice versa. The result is an implemented example of interoperability, produced with the aid of the OROS subset.

In addition, we have implemented a tool that automates this process: the tool accepts type definitions and code for implementations of operations, and outputs Ada packages that are interfaces to the defined types. We plan to extend this tool so that it can output C as well as Ada code. True interoperability, as the definition above suggests, implies the ability to communicate data between applications written in different languages; therefore, an important future direction for our work in this area is the development of tools that allow the use of an OROS-based type model to achieve interoperability among several different languages.

5 Summary and Future Directions

We have described OROS, an object-oriented type model that we have designed to meet the specific needs of object management systems for software development environments. The properties of OROS result from observations we have made about the requirements of such a type model. We have also discussed our experimentation and implementation experiences.

Experiments such as that described in Section 4.2 have been our most important source of feedback about the efficacy of basic OROS concepts. We expect OROS to be "fine-tuned" as we continue such experimentation, particularly as we make further progress towards automated type implementation support. We also plan to work towards integration of this "low-level" support with the "high-level" user interface ideas we described in Section 4.1.

The other major future direction in our research is the further investigation of theoretical concerns brought up by some of OROS's novel type modeling concepts. The most important of these concerns are subtyping-like intertype relationships among operation and relationship types (Section 2.3) and structural intertype relationships among type definitions (Section 2.4).

Acknowledgements

We appreciate the contributions made by Michel Bosco, Lori Clarke, Eliot Moss and Peri Tarr to the work described here. We also appreciate the comments and suggestions provided by our other colleagues in the Arcadia consortium.

At the University of Massachusetts, this work was supported in part by the National Science Foundation (CCR-87-04478) with cooperation from the Defense Advanced Research Projects Agency (ARPA order 6104).

References

[1] Antonio Albano, Luca Cardelli, and Renzo Orsini. Galileo: A Strongly-Typed, Interactive Conceptual Language. *ACM Transactions on Database Systems*, 10(2):230–260, 1985.

[2] Timothy Andrews and Craig Harris. Combining Language and Database Advances in an Object-Oriented Development Environment. In *OOPSLA Conference Proceedings*, pages 430–440, October 1987. Published as *ACM SIGPLAN Notices*, vol. 22, no. 12, December 1987.

[3] Philip A. Bernstein. Database System Support for Software Engineering—An Extended Abstract. In *Proc. 9th International Conference on Software Engineering*, pages 166–179, April 1987.

[4] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object Structure in the Emerald System. Technical Report 86-04-03, University of Washington, Department of Computer Science, April 1986.

[5] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. Distribution and Abstract Types in Emerald. *IEEE Transactions on Software Engineering*, SE-13(1):65–76, January 1987.

[6] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[7] Geoffrey Clemm and Leon Osterweil. A Mechanism for Environment Integration. Technical Report CU-CS-323-86, Department of Computer Science, University of Colorado, April 1986.

[8] E.F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[9] Scott Danforth and Chris Tomlinson. Type Theories and Object-Oriented Programming. *ACM Computing Surveys*, 20(1):29–72, March 1988.

[10] Alan Dearle, Richard Connor, Fred Brown, and Ron Morrison. Napier88—A Database Programming Language? In *Proc. 2nd International Workshop on Database Programming Languages*, pages 213–229, June 1989.

[11] Stuart I. Feldman. Make—A Program for Maintaining Computer Programs. *Software—Practice and Experience*, 9(4):255–265, 1979.

[12] Ferdinando Gallo, Regis Minot, and Ian Thomas. The Object Mangement System of PCTE as a Software Engineering Database Management System. In *Proc. 2nd ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, pages 12–15, December 1986. Published as *ACM SIGPLAN Notices*, vol. 22, no. 1, January 1987.

[13] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[14] Randy H. Katz. *Information Management for Engineering Design.* Springer-Verlag, 1984.

[15] David Maier, Jacob Stein, Allen Otis, and Alan Purdy. Development of an Object-Oriented DBMS. In *OOPSLA Conference Proceedings*, pages 472–482, November 1986. Published as *ACM SIGPLAN Notices*, vol. 21, no. 11, November 1986.

[16] R. Morrison, A.L. Brown, R. Carrick, R. Connor, and A. Dearle. *On the Integration of Object-Oriented and Process-Oriented Computation in Persistent Environments.* Persistent Programming Report 57, Department of Computational Science, University of St. Andrews, St. Andrews, Scotland, January 1988.

[17] J. Eliot B. Moss and Alexander L. Wolf. Toward Principles of Inheritance and Subtyping in Programming Languages. Technical Report 88–95, Dept. of Computer and Information Science, Univ. of Massachusetts, November 1988.

[18] Patricia A. Oberndorf. The Common Ada Programming Support Environment (APSE) Interface Set (CAIS). *IEEE Transactions on Software Engineering*, SE-14(6):742–748, June 1988.

[19] William R. Rosenblatt, Jack C. Wileden, and Alexander L. Wolf. Preliminary Report on the OROS Type Model. Technical Report 88–70, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts, August 1988.

[20] Lawrence A. Rowe and Michael R. Stonebraker. The POSTGRES Data Model. In *Proc. 13th International Conference on Very Large Data Bases*, pages 83–96, September 1987.

[21] James Rumbaugh. Relations as Semantic Constructs in an Object-Oriented Language. In *OOPSLA Conference Proceedings*. ACM, October 1987. Published as *ACM SIGPLAN Notices*, vol. 22, no. 12, December 1987.

[22] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An Introduction to Trellis/Owl. In *OOPSLA Conference Proceedings*, pages 9–16, September 1986. Published as *ACM SIGPLAN Notices*, vol. 21, no. 11, November 1986.

[23] Alan Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. In *OOPSLA Conference Proceedings*, pages 38–45, November 1986. Published as *ACM SIGPLAN Notices*, vol. 21, no. 11, November 1986.

[24] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.

[25] R. N. Taylor, F. C. Belz, L. A. Clarke, L. J. Osterweil, R. W. Selby, J. C. Wileden, A. L. Wolf, and M. Young. Foundations for the Arcadia Environment Architecture. In *Proc. 3rd ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, pages 1–13, December 1988. Published as *ACM SIGPLAN Notices*, vol. 24, no. 2, February 1989.

[26] Don Vines and Tim King. Gaia: An Object-Oriented Framework for an Ada Environment. In *Proc. 3rd International IEEE Conference on Ada Applications and Environments*, pages 81–92, May 1988.

[27] Jack C. Wileden, Alexander L. Wolf, William R. Rosenblatt, and Peri L. Tarr. *UTM-0: Initial Proposal for a Unified Type Model for Arcadia Environments.* Arcadia Design Document UM–89–01, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts, February 1989.

[28] Alexander L. Wolf. Language and Tool Support for Precise Interface Control. Technical Report 85–23, Dept. of Computer and Information Science, Univ. of Massachusetts, September 1985.

[29] Alexander L. Wolf, Lori A. Clarke, and Jack C. Wileden. The AdaPIC Toolset: Supporting Interface Control and Analysis Throughout the Software Development Process. *IEEE Transactions on Software Engineering*, 15(3):250–263, March 1989.

[30] Stanley B. Zdonik and Peter Wegner. Language and Methodology for Object-Oriented Database Environments. In *Proc. 19th Annual Hawaii International Conference on System Sciences*, pages 378–387, 1986.