Preliminary Report on the OROS Type Model

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

COINS Technical Report 88–70
August 1988

[†]*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 work was supported at the University of Massachusetts in part by the following grants: National Science Foundation DCR-84-04217 and DCR-85-00332; National Science Foundation CCR-87-04478 with cooperation from the Defense Advanced Research Projects Agency (ARPA order 6104); and Rome Air Development Center F30602-86-C-0006.

ABSTRACT

An important component of the next generation of software environments will be an *object management* system for handling the numerous and diverse kinds of information that they contain. One aspect of our research on object management for Arcadia environments is based on the view that all entities populating an environment should be instances of abstract types. This paper describes OROS, a type model that we are developing through a top-down effort toward defining requirements and specifications for type models specifically tailored to support software environment typing needs. We present a series of observations concerning properties of the objects that populate environments and we describe the characteristics of the current OROS type model arising from those observations. The use of OROS and the value of a type model in environment object management are illustrated through examples.

1. Introduction

The Arcadia project [19] is 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.

One important component of Arcadia environments will be an *object management* system appropriate for handling the numerous and diverse kinds of information that will be manipulated in a software environment. We are studying two aspects of object management. One of our projects, described here, is investigating type systems and type models suitable for environment object management. The other, described in a companion paper [23], addresses persistence of typed objects in an environment.

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 OROS project described in this paper is a top-down effort toward developing functional requirements and interface specifications for a type model specifically tailored to support environment object typing needs. 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. We have studied such previous approaches to object management as object-oriented database systems [1,2,13,27], relational/relationship systems [17,12,11], work on Ada environments [15,10,20] and others; however, we have consciously attempted to avoid being biased toward any one of these directions. We have also chosen to pay little or no attention to syntax or implementation concerns during this initial stage of our investigation. Instead, we have tried to focus solely on the properties that we believe are necessary or desirable in a type model that will be part of the infrastructure of Arcadia environments.

In this paper we present the current version of the OROS type model. We first provide some background on the perspectives that motivate our model. We then describe the model itself and the specific motivations for its various features. We give the flavor of OROS via an example of a type system that describes GRAPHITE, an actual software development project. Finally, we offer

an example of the how our type model would facilitate development and evolution of Arcadia environments, and we discuss future directions for this research project. An appendix contains the complete GRAPHITE example.

2. Background

Two important goals for Arcadia environments are integration and extensibility. In particular, Arcadia environments are intended to support experimental investigation of software process models and evaluation of novel tools in the context of a complete environment. This requires that Arcadia environments support the easy addition, modification and replacement of any and all kinds of environment components. Such modifiability is facilitated by an "organizational perspective" that explains the roles of various components and their relationships to other components. Automated enforcement of that organizational perspective helps to make modification easier and more reliable. No *a priori* fixed perspective can be expected to suffice in a truly experimental setting, however, so it must be possible for environment builders to create and modify that perspective.

An important kind of organizational perspective, and the one of interest to us here, is a type system. A type system is nothing more than a classification scheme for "things". In the context of software environments, the "things" to be classified include *environment components*, such as tools, management data or process descriptions, and *software product components*, such as specifications, designs, code, test data, or documentation. If the classification scheme can be enforced, by checking that each "thing" is a proper instance of some known type and that its uses are always in accordance with the properties associated with its type, then the classification scheme can greatly facilitate reliability and modifiability of software. Naturally, it must be possible for users 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. Our research in software object management is motivated by the belief that these observations are equally valid in the software environment domain. We elaborate on our motivations for this work in Section 6..

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.

Our current version of OROS is a first step toward making such choices, motivated by our view of significant concerns for those building and experimenting with environments. In the next sections we elaborate on the observations that we have made to date concerning typing requirements for environment object management and the properties of OROS that have resulted from those observations.

3. 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 classification 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 between entities, such as the connection between 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 `entity`, according to the notion of subtyping used, for example, in the Emerald system [4]. Figure 1 summarizes the primitive types of OROS.

Our inclusion of `relationship` as a primitive type of OROS is in contrast to recent work on object-oriented databases, which use type definition schemes similar to those of object-oriented languages. These schemes include notions of objects and operations, as well as only some particular

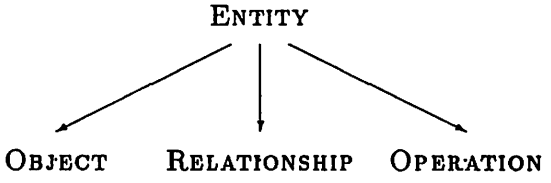

Figure 1: Primitive types of OROS

kinds of relationships between objects (e.g., binary relationships via attributes or properties); however, we feel that it is useful to define environment entities in terms of their general relationships to other 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[1] operating system [9]. 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].

Beyond the primitive types, everything in a given environment would normally be an instance of one or several subtypes. That is, a specific software environment's type system would include specialized subtypes derived from the primitive types of OROS. For example, something that is an instance of type `object` might also be an instance of the subtypes `text-object`, `programming-language-source`, and `Ada-source`, each of these subtypes being more specialized than its predecessor (see Figure 2). 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.

OROS, however, is a type model that makes no commitment to specific object types, relationship types or operation types. These vary with the particular process, environment, and so on. Thus, for example, an environment supporting a fairly traditional view of the software process might have object types such as `requirements`, `specification`, `design`, `programming-language-source` and

[1] UNIX is a registered trademark of AT&T.

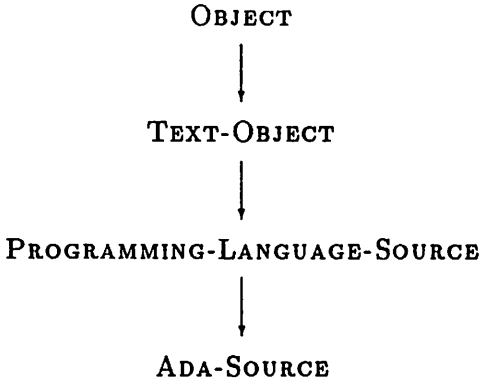

Figure 2: A hierarchy of object types

`test-case`, and operations such as `design`, `code`, `compile`, `link` and `load`. An environment supporting a transformational-style process [3], on the other hand, might have more types for specifications, no explicit design type, and operations such as `transform`, `select-transformation` or `record-transformation`.

Our second, related observation concerning the primitive types for an environment type model was that all three of the types shown in Figure 1 are of equal significance. That is, we do not believe that objects are of greater importance than relationships from the perspective of environment builders and experimenters, nor that any other such prioritization holds. This provides the rationale for the name OROS, which derives from Object, Relationship and Operation System[2].

This second observation also puts our requirements at odds with much of the previous work on type models. Most other typing mechanisms do not explicitly provide for all three of our fundamental types. Moreover, those that provide all three frequently treat one or more of them as subservient to the others. Thus, for example, object-oriented languages and databases have typically made object types primary, forcing all operations to belong to some object type. (A notable exception to this is the language C++ [18].) Entity-relationship models [5], on the other hand, have traditionally made relationships primary, relegating operations to secondary status.

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

4. 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 data abstractions have traditionally been defined in terms of the operations that can be applied to instances of some type, relationships are equally important determiners of a type's properties. Thus, for example, the fact that an object-code module is related to some source-code module by the `language-obj-code` relationship may be at least as important a property of type `obj-code` as the fact that the `link` operation can be applied to instances of that type.

Notice that relationships used in this way can subsume the notions of *attributes, properties* and *components* often used in object-oriented type models. In fact, those schemes use "attribute", "property" or "component" as a kind of syntactic shorthand for binary relationships between objects [20,21]. OROS, however, handles many-to-many relationships between entities and thus considers binary relationships as a special 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 both operation and 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.

The upshot of these two previous 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.

Our final observation along these lines was that not all of the relationships and operations applicable to instances of a given type are equally significant aspects of its definition. While a software developer may consider the `compile` operation to be central to the definition of a `programming-language-source` type, for example, a `format` (e.g., "pretty-print") operation might seem less significant, whereas a system documenter might consider them to be equally important. If an environment builder or maintainer can separate the parts of a type definition that appear

to capture the most significant aspects of that type's semantics from those that do not, then the environment can use this information to limit the potentially expensive impact of changes to the type system when they are made [26,22].

Our initial response to this observation has been to include a means for differentiating between two degrees of significance in the operations and relationships that contribute to the definition of a type. We currently use the terms "defining" and "auxiliary" to make this distinction. Thus, in our previous example, `compile` would be a defining operation and `format` an auxiliary operation in the definition of type `programming-language-source`.

We have found other possible uses for this defining/auxiliary dichotomy. For example, it might be useful to distinguish the operations and relationships that should be inherited by child types from those that should not. In fact, we suspect that a richer set of differentiations may be needed. We note the similarity of this concept to those of views in relational databases [8], capability lists in operating systems [16], and module interconnection descriptions in programming languages [24]; we have settled on the above scheme for our current version of OROS as a starting point for future investigation.

Finally, we also believe that an inheritance mechanism is a useful shorthand for type definition as well as a convenient way to represent a type system's organization. This has certainly been shown in previous work on object-oriented systems. In the example in the next section, we hope to show how inheritance can apply to environment *object* types. We suspect that inheritance may also be useful when applied to operation and relationship types; although our example does not motivate this, we expect to say more about it in future work.

Based on all of these observations, we have adopted the following method (and *ad hoc* syntax) for defining types in our current version of OROS:

```
type name is
    parents :          -- list of types to inherit from
    defining-ops :     -- list of defining operation types
    defining-rels :    -- list of defining relationship types
    aux-ops :          -- list of auxiliary operation types
    aux-rels :         -- list of auxiliary relationship types
    signature :        -- list of (name : type) pairs
end
```

The "signature" part is used in operation type definitions to specify the operation's parameters, while it is used in relationship type definitions to specify names and types of tuple elements. It is not used in the definitions of object types.

5. An Example

We now present an example of a type system defined via OROS, based on an ongoing development effort at the University of Massachusetts called GRAPHITE [6]. GRAPHITE is a fairly large system, composed of over 30,000 lines of Ada code in more than 50 files. We have used OROS to model GRAPHITE and various other large software systems, to help us get a clear picture of the objects found in an entire software environment and thus clarify the requirements for environment type models.

In this section, we present a few details of one possible type system that captures properties of the GRAPHITE system that may be interesting to a software developer. It is important to bear in mind that the following is by no means the only possible type system for GRAPHITE; in particular, different kinds of users (e.g., testers or maintainers) have different concerns and thus may prefer different type systems for the same entities.

Our complete OROS-based type system for GRAPHITE is given in Appendix A.

5.1 Background on GRAPHITE

GRAPHITE is a "meta-tool" that facilitates implementation of directed graphs. It takes as input a description of the types of graph objects, written in the language GDL (Graph Description Language), and outputs an Ada package that provides an interface between application code and the graph objects. The latest version of GRAPHITE is a "bootstrapped" version: its source code contains a GDL description of graph objects for the internal representation of its input. This GDL description is (initially) processed with an older version of GRAPHITE.

GRAPHITE's source code contains the following major components:

1. a description of the lexemes of the GDL language, which is input to the ALEX[3] lexical analyzer

[3] ALex, developed at the University of California at Irvine, is similar to the popular Lex tool of the UNIX operating system, except that it produces Ada code instead of C code. It has no connection with the third author of this paper.

generator;

2. a grammar for GDL with actions, which is input to the AYACC[4] parser generator;
3. a GDL description of the kinds of nodes used in GRAPHITE's internal representation of the GDL input;
4. the "main" routine of GRAPHITE; and
5. other Ada source-code modules, such as the one that implements GRAPHITE's symbol table.

5.2 The Scenario

A user is modifying GRAPHITE. Using the editor EMACS, the user makes a change to the grammar. As a result, it is necessary to run AYACC on the grammar, which in turn produces three Ada packages. Those three packages are then compiled, as is the "main" routine, which depends on the parser routines. The resulting object-code modules are finally linked with several others to produce a new GRAPHITE executable.

The above events involve the following entities in the system:

- `graphite-grammar`, the grammar for GDL;
- `parser`, `parse-tables` and `goto-tables`, Ada packages produced by AYACC;
- `graphite-main`, an Ada subprogram dependent on the above packages.
- object-code modules produced by the Ada compiler for the above packages;
- object-code modules corresponding to the other Ada source-code objects;
- `emacs`, the editor;
- `ayacc`, the parser generator;
- `ada`, the Ada compiler;
- `link`, the object code linker; and
- `graphite`, the executable being built.

[4] AYacc, also developed at the University of California at Irvine, is analogous to ALex for the Yacc parser generator of the UNIX operating system.

5.3 Type Definitions

Recall that the syntax we have adopted for defining types in OROS is purely *ad hoc*, since we have focused on type model requirements rather than user interface issues. In addition, any parts of type definitions we omit are assumed to be empty.

For the purposes of this example, we give these definitions of the OROS primitive types:

```
type entity is
    defining-ops:  create, delete
    defining-rels:  owner, permissions, creation-time
end

type object is
    parents:  entity
end

type operation is
    parents:  entity
    defining-ops:  invoke
    defining-rels:  obj-code-components-of-operation
end

type relationship is
    parents:  entity
    defining-ops:  get-tuple-element, put-tuple-element
end
```

The declaration of `entity`, the root of the type system, shows that all entities can be created and deleted, and that all have owners, permission structures and times of creation. These are represented by binary relationships with simple types such as integers or character strings. The `obj-code-components-of-operation` relationship on type `operation` is explained below.

Figure 3 depicts the inheritance hierarchy of type `object`'s descendent types. Remember that the types below `object` are not part of OROS *per se*, but represent part of one possible type system that can be constructed to describe the entities involved in GRAPHITE.

A few of these object types bear closer examination. Perhaps the most important child type of `object` is `text-object`:

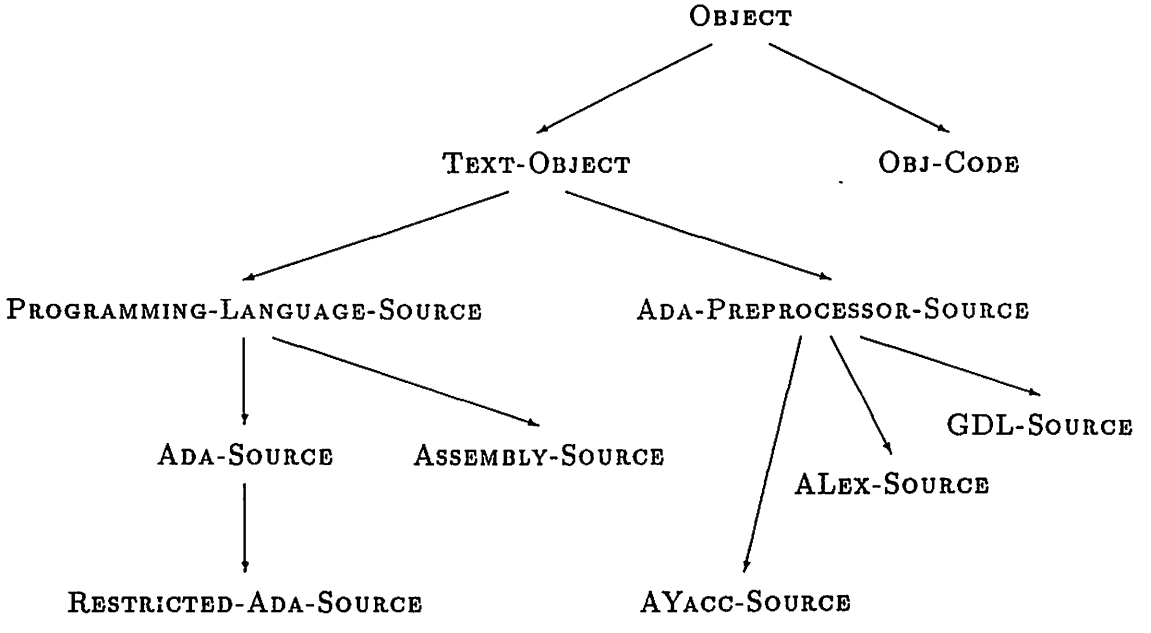

Figure 3: Hierarchy of object types for GRAPHITE system

```
type text-object is
    parents:  object
    defining-ops:  edit, print
    aux-ops:  format
end
```

That is, all text objects can be edited and printed. Less importantly, they can be run through some sort of formatter.

Support for (compiled) programming languages originates in type `programming-language--source`, of which all language-specific source-code types (such as `ada-source`) are child types:

```
type programming-language-source is
    parents:  text-object
    defining-rels:  language-obj-code
end
```

Because the development of GRAPHITE involves several Ada preprocessors (and because GRAPHITE itself is an Ada preprocessor), we found it useful to create a type for inputs to any of them:

```
type ada-preprocessor-source is
    parents:  text-object
    defining-rels:  unpreprocessed-form
end
```

Again, types for inputs to specific Ada preprocessors (e.g., `ayacc-source`, `gdl-source`) are child types of this one. An alternative perspective on the same kinds of entities might treat such things as AYACC as "languages", and thus might result in a type systemm in which `ada-preprocessor-source` would be a child type of `programming-language-source` in the type system.

The latter two type definitions contain relationships that express dependencies among instances of those types and instances of other types. For example, the definition of the `language-obj-code` relationship is:

```
type language-obj-code is
    parents:  relationship
    signature:  ( src :  programming-language-source, obj :  obj-code )
end
```

Recall that language-obj-code inherits such things as owner and creation-time from type entity. The signature specifies that, in each tuple (i.e., relationship instance), one programming-language-source object is related to one obj-code object. For example, (parser, parser-obj) could be a tuple of language-obj-code. Furthermore, a tool with the functionality of Make could use relationships like this to make sure that software components are up-to-date.

The relationship type unpreprocessed-form is an example of a non-binary relationship. It relates a language preprocessor's input to its output, and it allows for the fact that an Ada preprocessor may produce more than one output. For example, the AYACC parser generator produces three Ada source-code objects from its input. Its operation type is defined as follows:

```
type ayacc is
    parents:  operation
    signature:  ( grammar :  ayacc-source,
                  parser :  restricted-ada-source,
                  parse-tables :  restricted-ada-source,
                  goto-tables :  restricted-ada-source )
end
```

Finally, another interesting relationship is obj-code-components-of-operation, mentioned above as part of the definition of the primitive type operation:

```
type obj-code-components-of-operation is
    parents:  relationship
    signature:  ( objs+ :  obj-code, exe :  operation )
end
```

The objs+ in the signature is used to denote "one or more entities of type obj-code". This relationship is especially relevant to the notion of environment extensibility: by relating an operation (executable) to the object-code objects that link together to form it, the distinction between "software products" and "software tools" is effectively blurred. Thus, whenever a user (or automated

tool) links object-code objects to form an executable, they effectively install a new tool in the system. Of course, a type system builder may not wish to blur this distinction, and so may include a separate type such as `executable-object` for linker outputs.

Again, the full OROS description of the GRAPHITE system is given in Appendix A.

6. Value of Typing Mechanism

As we menitioned earlier, a primary goal of the Arcadia project is to support flexibility, extensibility and modifiability of software environments and software processes. A type model such as OROS can help with this in several ways.

First, the use of typing can help to catch or prevent errors in the construction of an environment. Standard type-checking techniques can be used to locate misuses of tools or data objects by environment builders and experimenters.

Second, the possibility of determining the complete set of types that would be affected by modification to some environment component can aid in prototyping and experimentation. For example, an environment experimenter contemplating a change to some object type could easily determine what relationships and operations might depend upon the current definition and thus gauge the potential impact of the contemplated change.

Third, the notion of auxiliary operations and relationships may help to limit the impact of change when an environment is modified. Specifically, one possible use of this distinction can be to differentiate between properties whose modification would have pervasive effects and those whose modification would have limited or no impact on anything beyond entities of the specific type being modified [26,22].

We illustrate the potential role of the OROS type model in supporting environment extensibility through the following example. Suppose that an environment builder has developed a new method for analyzing semantic dependencies among modules in a software system and wishes to experiment with a tool implementing that method. Suppose further that the environment in which the experiments will be conducted is partially described by a type system similar to the one in Section 5.. The experimenter might take the following steps.

First, the experimenter would determine how the new tool could fit into the existing type

system, which is tantamount to determining properties of objects to be operated on or produced by the tool. One aspect of this would be to specify a signature as part of the definition of an operation type to correspond to the tool. Another aspect would be to settle upon type definitions for the objects being operated on or produced, whose type names therefore appear in the signature. These might be pre-existing types whose properties match the tool's needs, or they might be new types. Appropriate locations in the existing type system (graph) would need to be determined for the tool's operation type as well as any new types. For example, the former might be most appropriately viewed as (a child of) type `analysis-op`, or `source-code-analysis-op`, or as a (possibly newly-defined) child of some other existing operation type. Similarly, the input to the tool might be most appropriately viewed as type `ada-source`, or a new child type of `ada-source` or of `programming-language-source` (perhaps containing new relationships that provide additional information). In addition to new types, this process might also suggest changes to existing types or the type system's organization.

Having made these type determinations and the corresponding modifications to the type system, and then having implemented the tool, the experimenter would be prepared to install it in the environment for experimental use. Analyses could now be carried out to guarantee consistent use of the typed objects, whether of new, modified, or pre-existing types. These analyses could discover erroneous assumptions about object properties on the part of the tool, or mistakes in modifying the type system. This, in turn, would substantially reduce the amount of debugging necessary to locate mismatches between tool requirements and the properties of other objects manipulated by the tool.

As this example suggests, a type model like OROS supports well-documented, and enforced, abstract interfaces among the objects populating environments. This facilitates changing or experimenting with an environment. It aids environment builders in preserving interface integrity across modifications and also helps to identify and limit the impact of changes. As a result, it contributes to environment extensibility.

7. Conclusion

We have presented a series of observations concerning properties of the objects that populate software environments, and we have described the characteristics of the OROS type model arising from those observations. We have also given examples illustrating the use and potential value of a type model like OROS in environment building, experimentation and maintenance.

To date, our focus has been entirely on requirements and specifications for a type model for environments. We plan to begin constructing a prototype implementation soon, so that we can gain some experience with actual use of OROS. The prototype should provide automated support for type definition, automated checking or enforcement of type usage, and ideally, automated conversion of existing entities when a type system is modified. Already Arcadia researchers have been building and using prototypes of subsets of the capabilities we seek. These include prototypes of an automatic implementation generator for persistent graph objects (PGRAPHITE, [23]), and systems supporting persistent relationship management (APPL/A, [11]) and analysis of module dependency relationships (PIC, [25]). We have also begun exploring candidates for the eventual underlying storage management system and a general-purpose interface to those systems (e.g., Mneme [14]).

Although we have intentionally ignored them to date, we also recognize the major importance of syntax and user interface to the successful adoption of a type model (e.g., the Smalltalk browser) and will be investigating these issues as well.

Through further examples and experimental use of prototypes, we expect to continue refining the OROS type model. The end result should be a specification for typing capabilities that are tailored to the specific needs of software enviroment builders and experimenters.

REFERENCES

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

[2] Thomas M. Atwood. An Object-Oriented DBMS for Design Support Applications. In *Proc. IEEE COMPINT 1985*, pages 299–307, 1985.

[3] Robert Balzer. A 15-Year Perspective on Automatic Programming. *IEEE Transactions on Software Engineering*, 1257–68, Nov. 1985.

[4] Andrew Black et al. Distribution and Abstract Types in Emerald. *IEEE Transactions on Software Engineering*, January 1987.

[5] P. Chen. The Entity-Relationship Model: Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1:9–36, 1976.

[6] Lori A. Clarke, Jack C. Wileden, and Alexander L. Wolf. GRAPHITE: A Meta-tool for Ada Environment Development. In *Proc. IEEE Computer Society 2nd International Conference on Ada Applications and Environments*, pages 81–90, April 1986.

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

[8] C.J. Date. *An Introduction to Database Systems, Vol. II.* Addison-Wesley, 1983.

[9] Stuart I. Feldman. Make – A Program for Maintaining Computer Programs. *Software – Practice and Experience, Vol. 9, No. 4*, 255–265, 1979.

[10] 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.

[11] Dennis Heimbigner, Leon J. Osterweil, and Stanley M. Sutton. *APPL/A: A Language for Managing Relations Among Software Objects and Processes.* Technical Report CU–CS–374–87, University of Colorado, Boulder, Colorado, 1987.

[12] Mark A. Linton. Implementing Relational Views of Programs. In *Proc. ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, pages 132–140, April 1984. Published as *ACM SIGPLAN Notices, vol. 19, no. 5*, May 1984.

[13] M.E.S. Loomis, A.V. Shah, and J.E. Rumbaugh. An Object Modelling Technique for Conceptual Design. In *Proc. European Conference on Object-Oriented Programming*, pages 325–335, 1987.

[14] J. Eliot B. Moss and Steven Sinofsky. Managing Persistent Data with Mneme: Designing a Reliable, Shared Object Interface. In *Proc. Second International Workshop on Object Oriented Data Bases*, September 1988.

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

[16] James L. Peterson and Abraham Silberschatz. *Operating System Concepts.* Addison-Wesley, 1983.

[17] 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.

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

[19] 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 *Proceedings SIGSOFT '88: Third Symposium on Software Development Environments*, December 1988. (to appear).

[20] 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.

[21] David S. Wile and Dennis G. Allard. Worlds: An Organizing Structure for Object-Bases. In *Proc. 2nd ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, December 1986. Published as *ACM SIGPLAN Notices, vol. 22, no. 1*, January 1987.

[22] Jack C. Wileden, Lori A. Clarke, and Alexander L. Wolf. Three Techniques Supporting the Development of Large Prototype Systems. In *Proc. 3rd International IEEE Conference on Ada Applications and Environments*, pages 28–38, May 1988.

[23] Jack C. Wileden, Alexander L. Wolf, Charles D. Fisher, and Peri L. Tarr. PGraphite: An Experiment in Persistent Typed Object Management for Environments. In *Proc. 3rd Software Development Environments Conference*, December 1988.

[24] 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.

[25] 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*, 1988 (to appear).

[26] Stanley B. Zdonik. Maintaining Consistency in a Database with Changing Types. *ACM SIGPLAN, Vol. 21, No. 10*, 120–127, 1986.

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

A. OROS Model of GRAPHITE

The entities described in Section 5. imply the type and entity definitions presented here.

The following is the complete OROS type system for GRAPHITE, which was excerpted in Section 5.. Any omitted parts of type definitions are assumed to be empty. Also, for brevity, we omit the definitions of "utility" types, such as `user-id`.

Additional notes on the syntax used:

1. In the signatures of relationships and operations, the notation `name*` is used to denote "zero or more entities of that type"; `name+` denotes "one or more entities of that type".
2. In the type `restricted-ada-source`, "`edit` = NULL" means that we wish not to inherit the operation `edit` from the parents. A restricted Ada source-code module is the output of an Ada preprocessor such as AYACC or GRAPHITE, and thus should not be modifiable by users.

A.1 Primitive Types

```
type entity is
     defining-ops: create, delete
     defining-rels: owner, permissions, creation-time
end

type object is
     parents: entity
end

type operation is
     parents: entity
     defining-ops: invoke
     defining-rels: obj-code-components-of-operation
end

type relationship is
     parents: entity
     defining-ops: get-tuple-element, put-tuple-element
end
```

A.2 Object Types

```
type text-object is
     parents: object
     defining-ops: edit, print
     aux-ops: format
end

type programming-language-source is          -- source for any compilable language
     parents: text-object
     defining-rels: language-obj-code
end

type ada-preprocessor-source is              -- inputs to alex, ayacc, graphite, etc.
     parents: text-object
     defining-rels: unpreprocessed-form
end

type alex-source is
     parents: ada-preprocessor-source
     defining-ops: alex
end

type ayacc-source is
     parents: ada-preprocessor-source
     defining-ops: ayacc
end

type gdl-source is
     parents: ada-preprocessor-source
     defining-ops: graphite
end

type ada-source is
     parents: programming-language-source
     defining-ops: ada
     defining-rels: ada-depends
end

type assembly-source is
     parents: programming-language-source
     defining-ops: assemble
```

```
end

type restricted-ada-source is                    -- outputs of ada preprocessors
     parents: ada-source
     defining-ops: edit = NULL
     defining-rels: unpreprocessed-form
     aux-ops: alex, ayacc, graphite
end

type obj-code is
     parents: object
     defining-ops: link
     defining-rels: obj-code-components-of-operation, language-obj-code
     aux-ops: optimize, disassemble
end
```

A.3 Operation Types

```
type create is
     parents: operation
     signature: ()                               -- empty signature
end

type delete is
     parents: operation
     signature: ( deletee : entity )
end

type invoke is
     parents: operation
     signature: ( op : operation, args* : entity )
end

type get-tuple-element is
     parents: operation
     signature: ( element-name : string,
                  element-value : entity)
end

type put-tuple-element is
```

```
    parents: operation
    signature: ( element-name : string,
                 element-value : entity)
end

type edit is
    parents: operation
    signature: ( texts* : text-object )
end

type print is
    parents: operation
    signature: ( text : text-object )
end

type format is
    parents: operation
    signature: ( text : text-object, formatted : text-object )
end

type alex is
    parents: operation
    signature: ( lex-spec : alex-source, lexer : restricted-ada-source )
end

type ayacc is
    parents: operation
    signature: ( grammar : ayacc-source,
                 parser : restricted-ada-source,
                 parse-tables : restricted-ada-source,
                 goto-tables : restricted-ada-source )
end

type ada is
    parents: operation
    signature: ( src : ada-source, obj : obj-code )
end

type assemble is
    parents: operation
    signature: ( asm-code : assembly-source, obj : obj-code )
```

```
end

type link is
    parents: operation
    signature: ( objs+ : obj-code, exe : operation )
end

type optimize is
    parents: operation
    signature: ( obj : obj-code, optimized-obj : obj-code )
end

type disassemble is
    parents: operation
    signature: ( obj : obj-code, asm-code : assembly-source )
end

type graphite is
    parents: operation
    signature: ( gdl-desc : gdl-source,
                 interface : restricted-ada-source )
end

type spell is
    parents: operation
    signature: ( text : text-object,
                 misspelled-list : text-object )
end
```

A.4 Relationship Types

```
type owner is
    parents: relationship
    signature: ( obj : object, owner-id : user-id )
end

type permissions is
    parents: relationship
    signature: ( obj : object, perms : perm-string )
end
```

```
type creation-atime is
    parents: relationship
    signature: ( obj : object, create-time : time-stamp )
end

type unpreprocessed-form is                    -- relates inputs to Ada preprocessors to the
                                               -- Ada code output by the preprocessor.
    parents: relationship
    signature: ( preproc-input : ada-preprocessor-source,
                 preproc-output+ : ada-source )
end

type ada-depends is                            -- used to express Ada package dependencies
    parents: relationship
    signature: ( depender : ada-source,
                 dependee+ : ada-source )
end

type language-obj-code is
    parents: relationship
    signature: ( src : programming-language-source, obj : obj-code )
end

type obj-code-components-of-operation is
    parents: relationship
    signature: ( objs+ : obj-code, exe : operation )
end
```

A.5 Entity Declarations

Here are the type declarations for the entities involved in the scenario of Section 5.:

A.5.1 Objects

```
graphite-grammar :  ayacc-source;
parser, parse-tables, goto-tables :  restricted-ada-source;
graphite-main :  ada-source;
parser-obj, parse-tables-obj, goto-tables-obj, graphite-main-obj,
```

```
    symtable-obj, lex-analyzer-obj, ...  :  obj-code;
```

A.5.2 Some Relationship Instances

```
owner =
    (graphite-grammar, Tarr)
    (parser, Tarr)
    ...

unpreprocessed-form =
    (graphite-grammar, parser, parse-tables, goto-tables)

ada-depends =
    (graphite-main, parser, parse-tables, goto-tables, symtable,
     lex-analyzer, ...)

language-obj-code =
    (parser, parser-obj)
    (parse-tables, parse-tables-obj)
    (goto-tables, goto-tables-obj)
    (graphite-main, graphite-main-obj)
    (symtable, symtable-obj)
    (lex-analyzer, lex-analyzer-obj)
    ...

obj-code-components-of-operation =
    (parser-obj, parse-tables-obj, goto-tables-obj, graphite-main-obj,
     symtable-obj, lex-analyzer-obj, graphite)
```