Object Management Technology for Environments: Experiences, Opportunities and Risks

Jack C. Wileden

Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

Alexander L. Wolf

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

ABSTRACT

For the past several years, we have been actively investigating object management technology for building software environments. Our experiences in this area have spanned type definition and implementation techniques, persistence models and implementation methods, type models for environments, and interoperability models and tools. In this paper we briefly summarize these experiences and then sketch both the opportunities and the risks that we foresee in future work on object management technology for environments.

1 Introduction

For the past several years, we have been actively investigating object management technology for building software environments. Most of this work has been done in the context of the Arcadia project [8], 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.

Our approach to object management technology for environments is based on the belief that its primary role is to support the operation of software development, maintenance and management *tools*. We further believe that these tools should be viewed as manipulating various kinds of *objects*, all of which should be instances of *(abstract) types*. Our experiences in defining and developing object management technology for environments reflect these beliefs. In particular, our experiences in this area have spanned type definition and implementation techniques, persistence models and

implementation methods, type models for environments, and interoperability models and tools. In this paper we briefly summarize these experiences and then sketch both the opportunities and the risks that we foresee in future work on object management technology for environments.

2 Type Definition and Implementation

A fundamental need of environment tool developers is support for defining and implementing relevant abstract types. Particularly in experimental, evolving environments, however, type definitions are subject to relatively frequent changes as new tool capabilities are investigated. Hence it is important that a type definition and implementation technique be *flexible* and able to *limit the impact of change.*

Our initial exploration of type definition and implementation techniques was through the development of the GRAPHITE [2, 15] system. GRAPHITE is a system for generating abstract interfaces to directed-graph data structures from declarative specifications of those structures; tools manipulate directed-graph objects through the abstract operations provided by the generated interfaces. GRAPHITE insulates tools from low-level object management concerns, making it easier to build and modify those tools. It also begins to address some object typing needs. For example, it has led to a better understanding of the requirements for an object specification mechanism (e.g., a language) and for the run-time type information to be maintained by an object management system. GRAPHITE has been used extensively in the development of a number of tools, including the PIC interface analysis toolset [16], a variety of design and programming language front ends, and concurrency analysis tools [1, 10]. It has also been used with some success by others, at a consortium that is building industrial strength software development environments and tools [3].

Largely through our work on GRAPHITE we have come to appreciate the range of possible approaches to type definition and implementation techniques for achieving flexibility and limiting the impact of change. We have attempted to characterize and comparatively evaluate such approaches [9, 14]. Our categorization distinguishes three categories of approaches to object definition, namely:

- **Specification-described.** Definitional information about an object is explicitly captured in the specification part of the object's description and can be referred to directly by clients of the object.
- **Implementation-described.** Definitional information about an object is described in the implementation part of the object's description and is referred to by clients through the values of parameters passed to general-purpose access routines.
- **Value-described.** Definitional information about an object is encoded in the values of a mutable data structure. Access to the description and to the object is through a general-purpose interface.

GRAPHITE, for example, represents an instance of the second category. While a definition technique's category is the main determining factor in how well it will support flexibility or limit the

impact of change, extensions and tool support provided with a technique can somewhat offset its categorical properties.

3 Persistence

We believe that the availability of a persistent object store, smoothly integrated into the language(s) used by environment and tool builders, will dramatically simplify the building of environments. Our research on persistence has been directed toward: 1) identifying an appropriate set of abstractions through which environment designers and tool builders can manipulate persistent objects; and 2) exploring implementation strategies for persistence.

The PGRAPHITE system [11] is a currently operational prototype of a persistent object capability built as an extension to GRAPHITE. Like GRAPHITE, this system is a preprocessor that accepts declarative specifications of abstract types for directed-graph objects (written, in fact, in the same Graph Definition Language (GDL) accepted by GRAPHITE) and produces an Ada implementation of abstract interfaces to the specified graph types. PGRAPHITE goes farther, however, incorporating persistence into the abstract interfaces that it generates. PGRAPHITE provides environment designers and tool builders with three sets of abstractions for manipulating persistent objects, namely persistent-object, persistent store and graph abstractions. Tools can access the persistent stores, which are called *repositories*, only during a *session*, and a tool must explicitly indicate the beginning and end of sessions. Sessions provide a basis for concurrency and transaction management and hence support sharing of persistent graph nodes, both among tools and between two or more graphs. In order to preserve abstract typing and information hiding, object classes manage their own persistence in PGRAPHITE [7]. Efficiency of both memory utilization and I/O traffic is increased through fault-driven retrieval of objects. The PGRAPHITE processor automatically generates Ada implementations of abstract types and repository managers. Finally, an important property of the PGRAPHITE approach is that persistence is orthogonal to other properties of typed objects.

Through use of the PGRAPHITE system we are exploring several important issues concerning persistent typed object management in environments. At the University of Massachusetts we are currently working to extend PGRAPHITE by automating the generation of additional types, and by adding support for concurrency, version control and garbage collection. We have also developed a version of PGRAPHITE that implements persistence using the Mneme storage manager [4] rather than Ada Direct_IO, as the original version did. This has significantly improved the system's performance and paves the way for future experiments in hosting PGRAPHITE on other storage managers.

Meanwhile, at AT&T Bell Laboratories we are pursuing a generalization and expansion of the model of persistence provided by PGRAPHITE. Where PGRAPHITE is strongly oriented toward an Ada-like type model and currently provides immediate support only for directed-graph objects, the generalization applies the basic persistence model within the context of C++'s richer ("object-oriented") type model and to arbitrary user-defined types. Furthermore, it provides a facility for deletion of persistent objects, something that virtually all other persistence mechanisms, including

PGRAPHITE's, do not attempt to handle. This broader model is implemented as a C++ library called Persi.

4 Type Models

Another component of our work on typing support for environment developers has been the definition of a type model[1] that we call OROS [5, 6]. The OROS type model is intended to permit environment builders and users to define types for both environment components, such as tools, process programs, etc., and software product components, such as requirements, designs, code, plans, etc. We believe that such pervasive typing can play a central role in improving the organization, increasing the reliability and facilitating the evolution of both environments and the software systems that they are used to build.

The OROS type model supports both the definition of types and the determination of inter-type relationships. Our choices of the primitive types, type definition scheme and type constructors for OROS reflect our views concerning the fundamental kinds of entities that make up a software environment, the equally important roles of *relationships* and *operations* in defining the types of those entities, and the need for precise, powerful and flexible specification of inter-type relationships. We are currently experimenting with the application of these facets of the OROS model to the description of various environment components to assess the accuracy of our views and the efficacy of the model. We are also designing prototype implementations of OROS. Through experimenting with such prototypes and refining OROS we expect to produce a specification of the type modeling capabilities needed in advanced software development environments such as Arcadia.

5 Interoperability

There is an increasing need and desire to develop environments by combining components that are written in different languages and/or that are run on different kinds of machines. Success at this depends in large part on the *interoperability* of the components—that is, the ability of the components to communicate and work together despite their differing backgrounds. While most previous approaches to interoperability have provided support at the representation level, we are pursuing an approach that will provide support at the specification level [13]. We have developed a model of such support that consists of four components: 1) a *unified type model*, which is a notation for describing the entities to be shared by interoperating programs; 2) *language bindings*, which connect the type models of the languages to the unified type model; 3) *underlying implementations*, which realize the types used by the different interoperating programs; and 4) *automated assistance*, which generally eases the task of combining components into an interoperable whole.

Central to the SLI approach is the unified type model (UTM). The intent is that a UTM should serve as a basis for tool cooperation within a richly structured collection of environment

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

components. Integration will be enhanced by permitting the tools to share and exploit the rich structure of those components. This is in direct contrast to the model found in the Unix[2] operating system, in particular, which forces tools to interact at the level of byte streams. The richer structure supported by a UTM will permit consistency checking (e.g., type checking) and will free tools from the necessity of explicitly translating (parsing and/or unparsing) inputs and outputs. At the same time, a UTM should not unduly restrict the typing choices of tool builders. Instead, it should facilitate experimentation with types, with internal representations for those types, and ultimately with the UTM itself. Hence a UTM can be viewed as a semi-strong coupling of environment components—stronger than typeless byte streams, but weaker than a single fixed type system. We believe that this intermediate position is the key to simultaneously attaining interoperability, integration and extensibility.

In the context of Arcadia, we have proposed an initial version of a UTM, called UTM-0 [12]. The UTM-0 proposal represents an attempt to be compatible with a variety of existing or proposed type models and type systems and is also intended to be appropriate for short term practical use in Arcadia prototypes. The UTM-0 proposal is based upon a simple subset of OROS concepts. It consists of a set of type definition primitives, a set of relationships or constructor functions, a set of "special" types, and some semantics for manipulation of instances. Results of initial experiments, in which we have manually applied the UTM-0 model to achieve interoperation of two Arcadia subsystems employing distinct type systems, have been extremely encouraging. A prototype implementation of automated support for UTM-0 is currently working and has been successfully used to support interoperation of tools written in Ada and Common LISP that are part of a concurrency analysis toolset.

6 Fitting It All Together

The experiences described in the preceding sections have addressed various facets of object management for environments. We are currently working on fitting all of these facets together into a coherent whole.

Not surprisingly, the facets are all relatively compatible. The OROS model represents an attempt to define an "ideal" type model for use in environment object management. As such, it reflects our experience with PGRAPHITE, in that it employs a similar (but extended) approach to type definition and is fully compatible with the PGRAPHITE approach to persistence. Our experience with both PGRAPHITE and OROS is also apparent in the SLI prototype, particularly in UTM-0. As noted above, UTM-0 is based on (a subset of) OROS. The SLI prototype is modeled on the PGRAPHITE preprocessor and explicitly incorporates the PGRAPHITE persistence model. The compatibility of PGRAPHITE and SLI is evident from the fact that our first demonstration application of SLI involved using a UTM-0 definition of a PGRAPHITE-defined object as the basis for interoperability between tools written in Ada and Lisp.

In the near term, we expect to fit the facets together primarily by constructing the necessary

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

support infrastructure, such as type definition tools, type definition browsers, object space browsers, and various implementation strategies for interoperation, to permit us to experiment with the combined capabilities of the prototypes that represent those facets. Such a combination, with the appropriate automated support, should provide a useful set of object management features for environment builders or users, and should permit us to experimentally evaluate the various facets and their interactions under more realistic circumstances than have been possible to date.

In the longer term, we expect that this collection of object management features will evolve and grow. In particular, we expect to explore additional facets of object management for environments, such as name space control, versioning, configuration management and concurrency control, with the results of those explorations leading to modifications and enhancements to the combined object management capabilities.

7 Opportunities

Although object management technology still needs to mature a good deal before it can really contribute to software environment building, we see several important opportunities for major improvements in environment state-of-the-art that can derive from this technology. We list some of those opportunities here:

1. *Easier tool construction.*

   Availability of appropriate abstract data types and orthogonal persistence should dramatically simplify the job of tool developers. Given this technology, they will be able to build tools in terms of operations on appropriate high-level typed objects, not forced to simulate them with complex manipulations on low-level types provided by C-class languages or current database technology. They will also be able to pay minimal attention to the persistence attribute of the objects that tools manipulate, defer decisions about when and whether an object should persist, and avoid writing the high percentage of tool code (estimates run from 30% to 70%) traditionally devoted to data translation and secondary-storage manipulation. Freed to focus on the primary functions of the tools that they are creating, developers should be able to produce more powerful and useful tools, faster and more reliably.

2. *Increased tool component reuse.*

   The increased use of abstractly typed objects and operations on them can lead to more component-based development of tools. This in turn should result in availability of more interchangeable tool parts, with better defined interfaces. Moreover, specification level interoperability offers the prospect of reuse of tool components across implementation language boundaries, which currently represent an almost impermeable barrier to component assembly.

3. *More environment interoperability and integration.*

   Interoperability support at the specification level should make it much easier for environment developers to mix and match environment components. It should also reduce the impediments

to environment integration that language differences presently impose. More generally, tool cooperation based on a powerful common type model like OROS can facilitate much higher levels of environment integration than the current low-level mechanisms of byte streams, files or relational database schemas.

4. *Easier environment evolution.*

   Adding or replacing tools in an environment where interfaces are all rigorously and precisely defined through common type models will be substantially easier than in present day environments. Having a clear, abstract description of how a new or replacement tool must interact with other tools in the environment should greatly ease the job of both the tool developer and the tool integrator. The set of type definitions existing for an environment at any point in time will also constitute a definition of the integration standards for the environment, much clearer and more reliable than any documentation could possibly be. Automated support for creating implementations from the type definitions will also simplify the environment evolution problem for the tool developer. Finally, the orthogonality of persistence opens the possibility of a new-tool developer deciding that some existing tool should preserve some object that it creates so that the new tool can operate on it, then making that change to the existing tool by simply adding one operation invocation (e.g., MakePersistent(ObjectOfInterest)).

8 Risks

While our experiences with various aspects of object management technology for environments have convinced us that such technology offers many opportunities for supporting the construction of powerful, integrated and extensible environments, we also believe that it is important to understand and address the risks associated with this technology. Following are several such risks that already can be identified:

1. *Complexity too great for use.*

   Relational databases were invented in part as a reaction to the complexity of network models. We are seeing in the area of object management a swing back toward complexity. We will be inundated by a sea of objects potentially representing every detail of design, development and management of a project. How will we cope with this information explosion? A related problem is that the theory underlying object management, namely *type theory*, is not nearly as tractable as that underlying such things as relational databases. We are not arguing in favor of relational databases, a technology clearly insufficient for this complex application. Rather, we are arguing that we must develop tools that help users deal with the complexity.

2. *The promise of integration is not delivered through interoperation.*

   There appears to be a confusion between the notions of *interoperation* and *integration*. Interoperation is essentially the movement of data (and sometimes control) from one tool to another. A simple way of achieving this is to have a common (external) representation and/or

object base (a primitive example: the byte stream and a file system). Integration, on the other hand, is a tight coupling of the functionality of tools, which presumably cooperate to achieve some goal. Interoperation is certainly necessary for integration, but it is not sufficient. An object management system can provide for interoperation, and thus facilitate integration, but only tool designers can effect integration—after a substantial amount of work. This confusion leads to false expectations; interoperation may "come for free", but integration does not.

3. *Abstraction implies loss of efficiency.*

   Current object management research is predicated on the substantial use of *abstraction*, which in turn implies a layering of information. Our experience with compilers for abstraction-oriented languages, such as Ada, has shown us that layering can cause significant degradation of performance. We must develop the means to automatically collapse layering. Abstractions are present for the benefit of humans; machines have little use for them.

4. *Poor insertion/migration.*

   Object management systems, as envisioned by most researchers, are large, complex, and costly. We must find a way of convincing people of the utility of object management systems without incurring all the costs of building them in their entirety. In other words, we must be able to show visible and viable intermediate products. We cannot expect users to accept object management systems on an all-or-nothing basis, given the radical changes implied by their use. A related issue is the need to develop an architecture for object management systems that will allow their evolution in the field.

5. *Early standardization without evolution path.*

   We seek to have the object management system lie at the heart of our software environments, serving as a platform for a large portion of the environment's functionality. But environment functionality is evolving just as object management systems are evolving. We must find a way to coordinate that evolution or the two may diverge.

Acknowledgments

Lori Clarke, William Rosenblatt and Peri Tarr have all contributed to several aspects of the object management technology described here. Many others of our Arcadia colleagues have provided valuable input to various parts of this work.

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] Avrunin, G.S., Dillon, L.K. and Wileden, J.C., *Experiments with Automated Constrained Expression Analysis of Concurrent Software Systems*, **Proceedings TAV3-SIGSOFT '89: Third Testing, Analysis and Verification Symposium**, December 1989, pp.124–130.

[2] Clarke, L.A., Wileden, J.C., and Wolf, A.L., GRAPHITE*: A Meta-tool for Ada Environment Development*, **Proc. IEEE-CS Second Inter. Conf. on Ada Applications and Environments**, Miami Beach, Florida, April 1986, pp.81–90.

[3] Dowson, M., *Experience Using the* GRAPHITE *Meta-Tool*, **Proceedings of the Twelfth International Conference on Software Engineering**, March 1990 (to appear).

[4] Moss, J.E.B., and Sinofsky, S., *Managing Persistent Data with Mneme: Designing a Reliable, Shared Object Interface*, **Proceeding of the Second International Workshop on Object Oriented Data Bases**, September 1988, Springer-Verlag.

[5] Rosenblatt, W.R., Wileden, J.C., and Wolf, A.L., *Preliminary Report on the OROS Type Model*, **COINS Technical Report 88-70**, Computer and Information Science Department, University of Massachusetts, August 1988.

[6] Rosenblatt, W.R., Wileden, J.C. and Wolf, A.L., *OROS: Toward a Type Model for Software Development Environments*, **Proceedings OOPSLA'89: Conference on Object-Oriented Programming: Systems, Languages, and Applications**, October 1989, pp.297–304.

[7] Tarr, P.L., Wileden, J.C. and Wolf, A.L., *A Different Tack to Providing Persistence in a Language*, **Proceedings Second International Workshop on Database Programming Languages**, June 1989, pp.41–60.

[8] Taylor, R.N., Belz, F.C., Clarke, L.A., Osterweil, L.J., Selby, R.W., Wileden, J.C., Wolf, A.L. and Young, M., *Foundations for the Arcadia Environment Architecture*, **Proceedings SIGSOFT '88: Third Symposium on Software Development Environments**, December 1988, pp.1–13.

[9] Wileden, J.C., Clarke, L.A. and Wolf, A.L., *Three Techniques Supporting the Development of Large Prototype Systems*, **Proceedings of the Third International IEEE Conference on Ada Applications and Environments**, May 1988, pp. 28-37.

[10] Wileden, J.C. and Avrunin, G.S., *Toward Automating Analysis Support for Developers of Distributed Software*, **Proceedings of the Eighth International Conference on Distributed Computing Systems**, IEEE Computer Society Press, June 1988, pp. 350-357.

[11] Wileden, J.C., Wolf, A.L., Fisher, C.D. and Tarr, P.L., *PGRAPHITE: An Experiment in Persistent Typed Object Management*, **Proceedings SIGSOFT '88: Third Symposium on Software Development Environments**, December 1988, pp.130–142.

[12] Wileden, J.C., Wolf, A.L., Rosenblatt, W.R. and Tarr, P.L., *UTM-0: Initial Proposal for a Unified Type Model for Arcadia Environments*, Arcadia Design Document UM 89-01, February 1989.

[13] Wileden, J.C., Wolf, A.L., Rosenblatt, W.R. and Tarr, P.L., *Specification Level Interoperability*, **Proceedings of the Twelfth International Conference on Software Engineering**, March 1990 (to appear).

[14] Wileden, J.C., Clarke, L.A. and Wolf, A.L., *A Comparative Evaluation of Object Definition Techniques for Large Prototype Systems*, **ACM Transactions on Programming Languages and Systems**, (to appear).

[15] Wolf, A.L., GRAPHITE *Reference Manual*, **COINS Technical Report 87-109**, Computer and Information Science Department, University of Massachusetts, October 1987.

[16] Wolf, A.L., Clarke, L.A., and Wileden, J.C., *The AdaPIC Toolset: Supporting Interface Control and Analysis Throughout the Software Development Process*, **IEEE Transactions on Software Engineering** vol. 15, no. 3, March 1989, pp.250–263.