From the Proceedings of the First IEEE Metadata Conference, Silver Spring, Maryland, April 16–18, 1996

Language-Based Support for Metadata

John Doppke, Dennis Heimbigner, and Alexander L. Wolf

Department of Computer Science
University of Colorado
Boulder, Colorado 80309 USA

{doppke,dennis,alw}@cs.colorado.edu

1 Introduction

Metadata is spreading beyond its historical application area, database management systems, to areas such as distributed object programming, large-grained parallel scientific computation, and large-scale continuously running systems. We therefore see significant benefits in moving from the current ad hoc, application-specific mechanisms to a unifying and generalizing metadata model.

Our goal is to develop packages and tools to support the development and use of metadata. Our approach is to exploit the power of language-processing technology to capture the range of metadata complexities conveniently and uniformly. In essence, we are creating a generic "metamodel" representation scheme that can be specialized to particular metadata applications. We have built a prototype, called CRIME, that demonstrates our ideas in the application area of distributed object technology.

There are several reasons why systems use metadata, including the following.

- *Completeness.* Some object-oriented systems, such as the Smalltalk programming language and the newer object-oriented databases, have as one of their goals the ability to represent everything as instances of classes. This includes classes themselves, which requires a fairly complicated "class class" structure that amounts to a run-time data structure that can be manipulated and browsed.

- *Extensibility.* It is often useful to extend the semantics of a language or database feature unobtrusively, such as to add the generation of tracing or triggering information to operation calls. The CLOS meta-object protocol [4], for example, supports this by allowing dynamic changes to metadata information.

- *Specialization.* Inheritance and subtyping mechanisms depend upon a dynamic binding of operation calls to operation bodies. Language and database systems dynamically resolve calls to bodies in the context of structural and semantic information represented in metadata.

- *Safety.* Strongly typed languages and databases that support dynamic type checking represent type information as metadata accessible at run time to the type checker.
- *Debugging.* metadata information is used in debugging to aid in the interpretation and modification of the state of an executing application.
- *Browsing.* Development of browsers for arbitrary types of data requires the ability to interpret the structure of those data, represented as metadata, in order to display it.
- *Program generation.* Given access to metadata, it is often possible to automatically generate programs that utilize information about that structure. Two examples are database query optimization and remote-procedure-call stub generation.

Recently, applications besides database management and programming language systems have begun to find a need to manipulate metadata. Distributed client-server computing provides a good example where, for instance, Prototech's MIF, OMG's CORBA, and ODP's Traders, all require access to metadata describing the interfaces (especially operation signatures) of available services so that distributed applications can locate and connect to appropriate services. Another example comes from large-scale, continuously running, high-assurance systems, such as the global telecommunications network, where both major and minor changes to the system, planned and unplanned, must be performed without disrupting ongoing processing; support for this implies that such a system must be dynamically self-configurable, which further implies that there must be a manipulable metadata representation of the system available to the system.

Historically, the development of support mechanisms for metadata have been largely *ad hoc* attempts to meet the specific needs of particular applications. This worked acceptably well while the applications were in limited and well-understood domains. We are at a critical time, however, where the need for metadata is beginning to spread widely and, therefore, there are significant benefits to be gained by attempting now to unify and generalize metadata support. In particular, we see the following benefits.

- *Reusability.* Currently, it is a major undertaking to add metadata support to applications. By providing reusable packages and tools, this effort can be significantly reduced. Additionally, it will be possible to reuse the metadata itself when defining the structure of data that is similar to some existing implementation.
- *Interoperability.* Developing consistent interfaces among applications or developing integrated interfaces on top of applications both require access to more than one metadata information structure simultaneously. If these "metadatabases" can be represented using the same metadata formalism, then the effort of effecting interoperability would be reduced.
- *Creativity.* Availability of metadata support should lead to a significant increase in the number and kinds of applications using metadata in novel and interesting ways. For example, one could imagine a metadata framework in which newly available components could dynamically self-integrate into a running system by exchanging information derived from metadata representations.

In the remainder of this paper, we describe requirements for generalized metadata support and discuss the approach we have taken in providing that support—namely, to employ *language technology* in the representation and manipulation of metadata information. Finally, we describe a prototype, called CRIME, that we have used to help refine and validate our ideas. First, however, we begin with a clarification of terminology.

2 A Note on Terminology

The term *schema,* in the database sense, is considered to be a formal description of the structure of some collection of data. A reified representation of those data is an example of metadata.

We treat the terms *catalog* and *data dictionary* as essentially synonymous. Both are interpreted to mean a database whose contents are metadata.

Analogous to the database terms *model, schema, data,* and *database,* we define the terms *metamodel, metaschema, metadata,* and *metadatabase.* The structure of metadata is defined by a metaschema. The metamodel is used to define the metaschema. The term metadatabase, or simply *metabase,* refers to some unified collection of metadata defined by some metaschema using some metamodel. As previously mentioned, a reified database schema, for example, would constitute a metabase.

3 Requirements

Support for metadata should satisfy at least the following requirements.

1. *Generality.* The metamodel should be capable of describing a range of metadata, from database schemas to application programs to scientific data.

2. *Simultaneous Representation.* The support mechanisms should allow simultaneous representation of multiple metadata schemas in order to support interoperability and integration of metadata and their applications.

3. *Reuse.* When developing metadata for a new application, it should be possible to reuse metadata from similar applications.

4. *Evolvability.* As sources and definitions of metadata evolve, their representations should smoothly evolve to match.

Notice that we do not include *reflectivity,* the use of the metamodel to represent itself, as a requirement. In practice, we expect any reasonable metamodel to be powerful enough to support itself; indeed, the prototype we have designed is an example of a metamodel used to represent itself. However, until more experience is gained, we do not see it as an essential requirement.

4 Applying Language-Processing Technology in Support of Metadata

These requirements imply that metadata need to provide a rather specific and complex set of information—in particular, the structure and semantics of data types and of module interfaces. This information goes far beyond what is captured, for example, in relational schemas. In fact, it is so similar to information found in modern programming languages that we have adopted a language view of metadata. The power of languages allows us to capture the range of metadata complexities conveniently and uniformly.

To provide support for manipulating metadata, an appropriate representation of the metadata must be chosen. Having placed metadata into a language framework, we turn to language-processing technology for that representation. The language and software-engineering communities have, for their purposes, settled upon the use of directed graphs, such as the Abstract Syntax Tree (AST) or the more sophisticated Abstract Semantics Graph (ASG), to represent the semantic content of programs. Such a representation scheme appears to be appropriate to our needs as well.

Because we wish to represent a variety of metamodels within a single metabase, the graph used to represent the metadata must be flexible. The approach we adopt is to use a regular, common structure for the graph plus a set of language-independent primitives from which multiple, language-specific representations can be defined. The primitives capture, in a language-independent manner, the structural and semantic concepts of the language being represented. The language-specific concepts are uniformly described within the representation (i.e., as data) to allow the development of general-purpose tools that can exhibit language-specific behavior by dynamically discovering the specifics within the representation. Language-specific tools can also be supported, if the representation is properly designed, by building a variety of optional language-specific interfaces on top.

Diana [2], Iris [1], and Reprise [6] are perhaps the best known of the abstract semantics graph models. Diana is inappropriate for our purposes because the graph structure is extremely language specific (i.e., to Ada). This means that one must define a new variant of Diana for every language. As a result, it is difficult for Diana to support the list of requirements in Section 3.

Iris and Reprise, which share a similar underlying conceptual model, appear to better match our requirements. The conceptual model is that of strongly typed expressions, where all semantic information about both user-defined entities and the language itself is uniformly represented as the application of operators to operands. For example, declarations are represented as the application of declaration operators that serve to introduce instances of particular entity kinds, such as types, functions, and objects. The declaration operators, which are specific to a particular language, are declared themselves using more primitive operators.

The model can best be understood in terms of three "layers" of operator declarations and uses, where operators defined within a layer can be used only within that and higher layers. The top, or *application layer,* consists of declarations and uses of operators specific to a particular application. For example, in a representation of a relational schema, there would be a use of a relation-declaration operator to create a particular relation in the schema. The relation-declaration operator in turn would be declared within the middle, or *language layer,* which consists of declarations and uses of operators specific to a particular language, such as that used in defining a relational schema. An object-oriented schema language would have a different set of operator declarations and uses at this level. Finally, the bottom, or *paradigm layer,* consists of operator declarations and uses specific to a particular language paradigm, such as the relational paradigm; we would expect different languages within the same paradigm to share operators at this level. The advantage of this layered approach is that it clearly provides significant opportunities for reuse of operator declarations.

Iris was developed in conjunction with the Arcadia project [3] primarily to represent Ada and several Ada variants. Reprise, which was inspired by Iris, was developed at AT&T Bell Laboratories primarily to represent C++. Because Reprise has been used to represent a language with a more modern, sophisticated type model, we have choose to use the Reprise approach in this project.

The actual implementation of the representation must take into account the complexity of the information and the expected access patterns. At this point, it is not clear whether traversal-oriented or query-oriented access to the graph-based representation will predominate. In fact, it is likely that both kinds of access will be heavily used. Initially, therefore, we turn to an object database approach, since complex graphs accessed both via traversal and query are the intended application domain of the object database technology.

4.1 Required Enhancements to Reprise

We have identified three enhancements to the basic Reprise technology that will provide significant additional functionality. They are independent of any particular set of operators and, thus, we could expect any Reprise-based metadata system to provide these enhancements.

4.1.1 Annotations

Experience has shown the need to *annotate* nodes in the Reprise graph with arbitrary pieces of additional information, including other Reprise graphs. In the C++ representation, for example, it is convenient for a data-flow analysis tool to annotate nodes with path predicates. In representing a relational database schema, one might annotate relation declarations with information about physical layout or about cardinalities.

The key issue in supporting annotations is independence. The set of possible annotations associated with a node cannot be known *a priori.* Moreover, different applications will be interested in different annotations. This means that the annotations should be kept *logically independent* of the nodes such that the nodes should have no "knowledge" of any associated annotations and applications should have "knowledge" of only those annotations in which they are interested.

We use the term "logically independent" because various physical implementations are possible for annotations that may unobtrusively violate independence. For example, one might keep annotations in physically separate binary relationships and use associative retrieval to find annotations. Alternatively, each node might point directly to a list of annotation name/value pairs, where retrieval is performed by searching the list.

4.1.2 Generic Traversal

Many algorithms over graphs have a similar form. They perform some action on an initial node and then recurse to apply themselves to some set of nodes reachable from the initial node. If the graph is not a tree, then information is often kept about already visited nodes. We would expect such algorithms to be common in the tools operating on Reprise graphs and so it seems useful to provide a generic mechanism to support such traversals.

4.1.3 Associative Retrieval

It should be clear that Reprise graphs are biased in the direction of navigation-based processing. But it is clear from database experience that associative access to metadata is also desirable. For example, one might want to obtain the set of all nodes in a graph that refer to some type declaration or one might want to obtain a list of all classes or of all relations.

Obviously, such questions can be answered by traversing the graph and collecting the information. But it is often more convenient and efficient to provide some form of associative access to the nodes of the graph to support such queries. Several implementation mechanisms are possible here; B-trees and hashing are two obvious candidates.

5 CRIME: A First Prototype

We have developed a prototype system to help illustrate the ideas about metadata that we have set out. This prototype, the CORBA Repository for Interface Metadata Exchange (CRIME), uses the Reprise notation to represent the interface definition language (IDL) used in the distributed object environment CORBA.

5.1 CORBA Interface Repository

Recognizing the need for dynamic, run-time access to interface metadata, the CORBA standard [5] defines the interface for an *interface repository.* The standard states several reasons why such a system ought to have access to such metadata, including:

- to provide type-checking of request signatures,
- to assist in checking the correctness of interface inheritance graphs,
- to assist in providing interoperability between different ORB implementations,
- to manage the installation and distribution of interface definitions,
- to provide components of a CASE environment (e.g., an interface browser),
- to provide interface information to language-bindings, and
- to provide components of end-user environments.

The approach taken by CORBA was to specify an interface to which such an interface repository (IR) should respond. Accordingly, implementing the IR seemed to be a suitable task: the standard constrained only the interface, not the implementation, so any IR implementation could use its own metadata representation to accomplish its tasks.

The IR interface given by the standard was, not surprisingly, written in IDL, and so the IR represents an example of a self-representing, or *reflective,* system. This interface represents each IDL declaration as a distributed object whose interface corresponds to the type of declaration. For example, all declarations inherit a base interface, so any declaration can report its name, its unique ID, and a description of itself. An interface object can return a list of attributes, base interfaces, etc.; an operation object can return a structure corresponding to its signature; an attribute object can return its type; and so on.

5.2 The CRIME System

The CRIME system consists of three parts. The first part reads in the interface metadata in IDL and generates a Reprise representation of the same metadata. The CRIME server then uses the metadata (in its Reprise notation) to provide distributed access to the metadata via the CORBA-specified IR interface. Finally, an interface browser presents the metadata available from the server in a convenient, Smalltalk-style browser.

5.2.1 Reprise generator for IDL

Given that our approach to metadata is based on the use of Reprise as a common metadata representation, any system based on this approach must be able to generate metadata in this common representation. Since CORBA interfaces are generally described in IDL, it was necessary to design CRIME in such a way that it is able to translate native IDL metadata into the common Reprise representation.

IDL proved to be an interesting source of metadata to attempt to represent in Reprise. Reprise had been developed to represent C++ and had not previously been tested in its ability to act as a metamodel for other languages. IDL borrows much of its syntactic and semantic style from C++, and as a result the definition of the operator set and expression structure for IDL within Reprise proved to be quite straightforward. In particular, IDL's conceptual overlap with C++ led to significant reuse of the C++ operators.

In the ability to perform this translation, then, lies a demonstration of the generality of the Reprise metamodel. While experiments with other languages would help validate the point further, this initial success is a very encouraging result.

5.2.2 CRIME Server

The CRIME server uses the Reprise IDL representation as its underlying metadata store; the contents of this store are presumably generated by the aforementioned translator of IDL into Reprise, though given the common representation other tools could conceivably generate interface metadata as well. The server then provides access to interface metadata via the CORBA-specified IR interface.

As mentioned in section 4.1.3, extending the Reprise notation to include means of access other than navigation may prove useful for many applications. Indeed, such an extension was necessary in the CRIME server, since the IR specification requires that the server provide unique IDs for all IDL declarations and that it be able to look up the declaration corresponding to a given ID. Since each IDL declaration always has a unique fully-scoped name, the IDs assigned were simply these fully-scoped names, and a hash table mapping names to declarations was created to permit this associative method of access.

5.2.3 CRIME Browser

In order to illustrate the utility of the availability of interface metadata, at least from the user's perspective, we created the CRIME Browser. This browser uses the server's CORBA interface and thereby allows the user to peruse IDL declarations via a Smalltalk-style browser.

A picture of the CRIME Browser is given in Figure 1. In this picture, the leftmost list

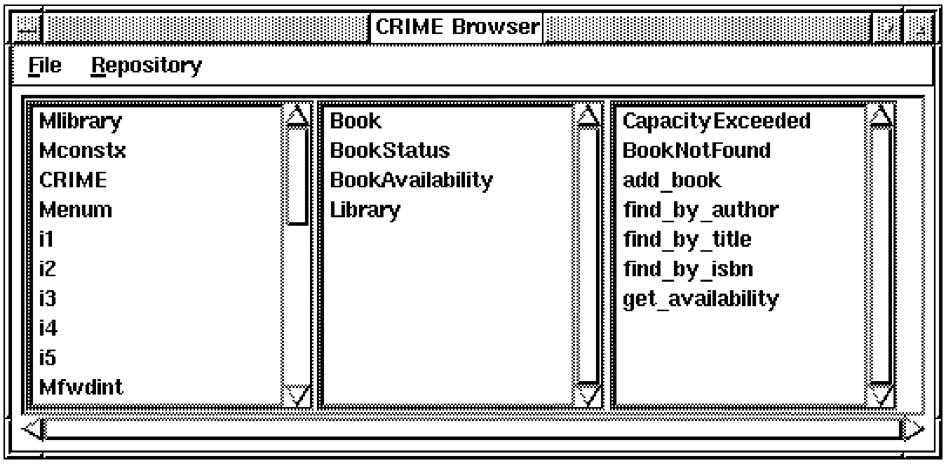

Figure 1: The CRIME Browser.

represents the set of top-level declarations, the middle list represents the contents of one of the top-level items (in this case, `Mlibrary`), and the rightmost list represents the contents of one of the middle items (in this case, `Library`). For any declaration, the user can examine the item and get any information available on it, such as operation signatures, attribute types, base interfaces, composite type declarations, etc.

5.3 Future Enhancements to CRIME

As mentioned in section 4.1, various enhancements to Reprise should be made in order to make the metadata easier to use in several ways. As Reprise evolves, CRIME may also evolve to make use of enhancements in the metamodel. For example, additions could be made to CRIME using Reprise graph annotations—for example, object interfaces could be annotated with information specifying where on the network the object is actually running.

Another addition to CRIME could result from the existence of associative retrieval methods over Reprise graphs beyond those based on declaration names. The CRIME interface could then perform more sophisticated queries—such as a search over only certain types of declarations—in a very general and efficient manner.

6 Conclusion

The availability of metadata is thus increasingly critical in a widening number of domains. By appealing to language-processing technology in our approach to metadata, we are able to exploit the substantial body of experience from that field and apply it to these increasingly numerous fields of application. In constructing our prototype, we have demonstrated the ability to use a metamodel from one domain and apply it to another without loss of generality, and such re-application of technology demonstrates the ability to reuse metadata representations and thereby to unify metadata support across diverse applications.

7 Acknowledgments

This material is based upon work sponsored by US West Advanced Technologies, the Air Force Material Command, Rome Laboratory, and the Advanced Research Projects Agency under Contract Number F30602-94-C-0253. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

References

[1] D.A. Baker, D.A. Fisher, and J.C. Shultis. *The Gardens of Iris.* Incremental Systems Corporation, Pittsburgh, PA, 1988.

[2] A. Evans, K.J. Butler, G. Goos, and W.A. Wulf. *Diana Reference Manual, Revision 3.* Tartan Laboratories, Inc., Pittsburgh, PA, 1983.

[3] R. Kadia. Issues Encountered in Building A Flexible Software Development Environment. In *Proc. Fifth ACM SIGSOFT/SIGPLAN Symposium on Software Development Environments*, Washington, D.C., 9-11 December 1992.

[4] Gregor Kiczales, Jim des Rivieres, and Daniel Bobrow. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, Mass., 1991.

[5] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 26 August 1991.

[6] D.S. Rosenblum and A.L. Wolf. Representing Semantically Analyzed C++ Code with Reprise. In *Proceedings of the Third C++ Technical Conference*, pages 119–134. USENIX Association, April 1991.