A Comparative Evaluation of Object Definition Techniques for Large Prototype Systems

JACK C. WILEDEN and LORI A. CLARKE
University of Massachusetts
and
ALEXANDER L. WOLF
AT&T Bell Laboratories

Although prototyping has long been touted as a potentially valuable software engineering activity, it has never achieved widespread use by developers of large-scale, production software. This is probably due in part to an incompatibility between the languages and tools traditionally available for prototyping (e.g., LISP or Smalltalk) and the needs of large-scale-software developers, who must construct and experiment with *large* prototypes. The recent surge of interest in applying prototyping to the development of large-scale, production software will necessitate improved prototyping languages and tools appropriate for constructing and experimenting with large, complex prototype systems. We explore techniques aimed at one central aspect of prototyping that we feel is especially significant for large prototypes, namely that aspect concerned with the definition of data objects. We characterize and compare various techniques that might be useful in defining data objects in large prototype systems, after first discussing some distinguishing characteristics of large prototype systems and identifying some requirements that they imply. To make the discussion more concrete, we describe our implementations of three techniques that represent different possibilities within the range of object definition techniques for large prototype systems.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques—*modules and interfaces*; D.2.7 [**Software Engineering**]: Distribution and Maintenance—*enhancement, extensibility, portability*; D.2.m [**Software Engineering**]: Miscellaneous—*rapid prototyping*; D.3.3 [**Programming Languages**]: Language Constructs—*abstract data types, data types and structures*; E.1 [**Data**]: Data Structures—*graphs*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms: Languages, Reliability

A preliminary version of this paper, entitled "Three Techniques Supporting the Development of Large Prototype Systems," appeared in *Proceedings of the Third International IEEE Conference on Ada Applications and Environments*, (May 23–25, 1988, Manchester, NH). IEEE, New York, 1988, pp. 28–37.
At the University of Massachusetts, this work was supported in part by the following grants: National Science Foundation DCR-84-04217, DCR-84-08143, 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.
Authors' addresses: J. C. Wileden and L. A. Clarke, Software Development Laboratory, Computer and Information Science Department, University of Massachusetts, Amherst, MA 01003; A. L. Wolf, AT&T Bell Laboratories, 600 Mountain Ave., Murray Hill, NJ 07974.

Additional Key Words and Phrases: Comparative evaluation, consistency management, controlled impact of change, easy modification, evolving software systems, object definition techniques, rapid development, reuse

1. INTRODUCTION

Although prototyping has long been touted as a potentially valuable software engineering activity [14], it has never really fulfilled that potential. In particular, prototyping has not achieved widespread use by developers of large-scale, production software. This is probably due in part to the dubious suitability of traditional prototyping languages and tools (e.g., LISP or Smalltalk) for use in constructing and experimenting with *large* prototypes.

Recently there has been a surge of interest in applying prototyping to the development of large-scale, production software, and a corresponding increase in efforts to create suitable languages and tools for this purpose (e.g., [3]). This is a response to the fact that software systems are getting larger, more complex, and costlier to build. In addition, the organizations acquiring these systems are demanding more, and earlier, involvement in the development process and, therefore, need to be "shown" something sooner. Creating appropriate prototyping languages and tools for responding to these needs will depend upon achieving a better understanding of how to support prototyping. In particular, we see a need for techniques that support the development of large, complex prototype systems, since prototypes of large, complex systems are likely to be themselves large and complex.

In this paper, we explore techniques aimed at one central aspect of prototyping, namely that aspect concerned with the definition of data objects. We seek to characterize and compare various techniques that might be useful in defining data objects in large prototype systems. Our characterization and evaluation address questions such as: how those definitions are made, what form they take, where they are located, how they are used, how they are changed, and what can be done to control the effects of those changes. Section 2 first discusses some distinguishing characteristics of large prototype systems and identifies some requirements for object definition techniques that can support such prototyping. Section 3 then characterizes a range of object definition techniques that fulfill those requirements to a greater or lesser extent. To make the discussion more concrete, Section 4 contains descriptions of implementations that we have constructed for three techniques that represent different points within the characterized range. A single example is used to illustrate and compare those implementations and, by extension, the techniques they represent. Section 5 presents a comparative evaluation of the techniques laid out in Section 3. We conclude with a summary of the results presented in this paper, a look at related work, and a prospectus of future work.

We have chosen to focus this paper on large prototypes primarily because that was the context in which our work originated and because of the current surge of interest in large-scale prototyping. All nontrivial software systems evolve, however, and it may well be that the only significant distinction between large prototypes and other large software systems is the rate at which their developers expect them to change. Since many of the properties that we discuss here are relevant to any evolving software system, we hope that our observations and analysis may find even broader applicability.

2. REQUIREMENTS FOR LARGE PROTOTYPE SYSTEMS

The motivation for prototyping in software development is the same as in other engineering activities: the prospect of gaining information about, and experience with, the behavior and structure of a system before that system is actually built. The sooner and more thoroughly a prototype can be experimented with, the more information and experience it will provide, and the more valuable it will be. Thus, there are two fundamental requirements for prototyping of software systems:

(1) *Rapid development.* A first version of a prototype software system should be up and running as quickly as possible. In other words, a developer should experience minimal delay between conceiving of a system and being able to experiment with a first prototype of that system.

(2) *Easy modification.* Changes in the prototype, often suggested by the results of previous experiments, should be easy to incorporate. In other words, a developer should experience minimal delay between experiments.

Small prototypes of small- to medium-scale programs, constructed and used by a single developer, have often been able to meet these goals rather easily. Since they have typically consisted of relatively few distinct components that are relatively loosely coupled, and since efficiency, in terms of execution speed or space consumption, has generally been of little importance, it has been acceptable to construct small prototypes using interpreted, weakly typed languages such as LISP or Smalltalk. Indeed, the Smalltalk environment was developed, at least in part, to provide a language and tools for prototyping, and both it and LISP have proven useful in many small- to medium-scale prototyping efforts.

The development of large prototype software systems seems to require approaches qualitatively different from those used for smaller prototypes. In general, large prototypes are distinguished from small prototypes, not only by their greater number of modules and lines of code, but also by their higher cost, longer lifetimes, and the involvement of multiple developers. Consequently, they seem to require more extensive and stricter management. Large prototypes are by nature complex and highly interrelated collections of components. In addition, partly due to their size and partly because of the kinds of experiments they are to be used for, efficiency in both time and space is significant, although not as significant as for the final product.

A good example of a large prototype system, and one that we are currently involved in developing with a number of other researchers, is a prototype of a software development environment [15]. A full-fledged software environment prototype will consist of a large number of components interacting with one another in a variety of complex ways. Those components include tools, such as editors, compilers, testing and debugging support systems and the like, and also data objects, such as source text, abstract syntax trees, load modules, symbol tables, test data sets, test results and many others. The components are highly interrelated in that a typical activity by a user will involve coordinated actions by several tools affecting several (typically shared) data objects. Evaluation of an environment prototype will entail experimentation with individual components as well as with the integration of those components. Because a software environment is intended to be a highly interactive system, evaluation of the prototype will be unavoidably affected by performance concerns such as response time.

An important implication of the characteristics of large prototype systems is that they are likely to be developed and modified "component-wise". That is, a developer is likely to experiment with one component at a time, by adding one or modifying one while leaving the rest of the prototype unchanged. The experiment itself will not be restricted to assessing the changed component in isolation, however. Instead, the developer's interest will be in how the changes integrate with all the other components of the prototype. A developer of a software environment prototype might, for example, conceive of a new code-analysis capability. Incorporating it might involve adding or modifying one tool and making a few minor changes in the definitions of a few shared data objects. Experimenting with this new capability will not be limited to use of the new or revised tool, but will also address how well the new capability integrates with other tools in the environment. While experimentation of this kind should be encouraged and facilitated, the characteristics of large prototype systems also imply that there must be security against the introduction of inconsistencies into the prototype; controlled and disciplined change is vital, especially when dealing with a large, complex system composed of highly interrelated components.

As our experience with prototyping of software environments has demonstrated, a central aspect of constructing and experimenting with large prototypes is the creation and manipulation of various kinds of data objects used in the prototype system (and eventually in the "real" system). Experimentation with a prototype will often involve defining new kinds of data objects or modifying existing ones, as in the example cited above. Based on our experience with this aspect of prototyping, we have identified the following *requirements* on object definition techniques for large prototype systems:

—Easy definition and redefinition of data objects
—Easy reuse of object definitions and of *clients* of object definitions
—Easy and reliable maintenance of consistency between object definition and use
—Control over the impact of changes to object definitions

The requirement for easy definition and redefinition of data objects follows directly from the primary goals of rapid development and easy modification in prototyping. The implications of this requirement range from powerful and concise language constructs for object definition to minimizing the effort required for effecting a modification to an object's definition.

Easy reuse also contributes to both rapid development and easy modification of prototypes. Properties such as modularity and understandability clearly affect how easily an object definition can be reused. But properties of various object definition techniques also affect how easily other components (i.e., the *clients* of objects) in prototype systems can be reused. For example, certain tools in a prototype software environment can be made generic across a broad class of loosely similar kinds of objects if the descriptions of objects are available for interpretation by those tools at run time.

The requirement for consistency of object definition and usage, as mentioned previously, is extremely important in large prototyping projects. Not only must it be possible to establish such consistency when a prototype is first created, but it must also be possible to reliably determine and/or enforce the preservation of consistency across modifications to the prototype.

Finally, all of the preceding requirements imply the need for controlling the impact of change to a prototype. The implications of this requirement include both an ability to clearly identify the parts of a prototype that will be affected by some modification and the ability to limit the impact of the change to only those parts that actually need to be affected. We have come to describe this in terms of limiting the impact of the change to only those components of a prototype that are "interested" in the change.

In our efforts to support the development of large prototype systems, we have taken as our starting point the use of a compiled, strongly typed, and statically type-checked language, in part because use of such a language generally tends to result in fewer errors and better efficiency than interpreted, weakly typed languages. Examples of suitable languages include Ada, C++, Modula-2, and Trellis/Owl [12]. These languages provide mechanisms for modularizing a prototype, specifying its data objects and module interfaces, and checking the consistency of those objects and interfaces. Thus, these languages clearly have the potential to support reuse and consistency management. They also provide a basis for controlling the impact of change through their facilities for information hiding and separate compilation.

Unfortunately, despite their apparent potential, use of compiled, strongly typed, and statically type-checked languages in the development of large prototype systems can lead to unacceptably slow development and modification. This is because such languages do not, in their native form, provide sufficiently powerful support for the kinds of reuse, consistency management or control over the impact of change that are needed for large prototyping applications. Their shortcomings are especially severe with respect to controlling the impact of change. Frequently, a small change in a program written in such a language, especially if that change involves a system component that is widely used by other components, necessitates code regeneration and a complete consistency check, which are done through recompilation.[1] This is generally true even if the change being made actually affects only a very few components.

As stated above, we believe that support for large-scale prototyping requires the ability to limit the impact of change to only those components of the prototype interested in the change. Consider the example mentioned above of experimenting

[1] Although "object oriented" languages such as C++ and Trellis/Owl provide *dynamic binding* of operation bodies to operation calls, they still rely on static, compile-time consistency checking. Moreover, the set of possible bindings, from which a particular binding is chosen dynamically, is established statically.

with a new code-analysis capability in a prototype software environment. Suppose that the changes required for this experiment are to add one new tool and to introduce one new field into the definition of one data object shared by many of the tools already populating the prototype. The new tool and the shared data object are the only components of the prototype environment interested in this change; the other tools need never be aware that the new field exists in the data object. The impact of the change is likely to extend well beyond just the interested components, however. When the interface to a data object is modified, most language processing systems (typically compilers) for compiled, strongly typed and statically type-checked languages will perform widespread rechecking of type consistency and regeneration of code. In particular, all tools and objects that refer to, or worse, that might possibly refer to, the modified object will typically be type-checked again and the code implementing those references will typically be regenerated, often entailing a complete recompilation. In contrast, if our goal were achieved, at most only the two interested components would be subject to type rechecking or code regeneration. Several approaches to achieving this goal are discussed in this paper.

Unfortunately, object definition techniques that are good at controlling the impact of change often have diminished capabilities for reuse and consistency management. The difficulty here stems from a basic conflict in the amount of information that should be contained in an object's interface. On the one hand, the desire to make an object easy to reuse, as well as the desire to check consistency, seems to argue for having an information-rich interface—that is, an interface that contains a detailed specification of the object. For example, an information-rich interface might provide distinct functions to access each of the components of an object, thereby revealing the objects' structure (see Figure 5). On the other hand, the desire to limit the impact of change seems to argue for an information-poor interface so that the details of the object can change without necessarily affecting all clients of the object. For example, an information-poor interface might provide a single function to access all the components of an object, parameterized by an indication of which component of an object is desired, where the possible values of the parameters are not explicitly specified in the interface (as illustrated by the use of the string-valued parameter TheAttribute to operations GetAttribute and PutAttribute in Figure 3). While the most obvious use of the constructs provided by languages such as Ada, C++, Modula-2, and Trellis/Owl—that is the programming style implied by the designs of those languages and advocated by a variety of texts—seems to favor information-rich interfaces, the languages can equally well support information-poor interfaces, so there is no answer inherent in the languages themselves. In fact, there is probably no one answer that is appropriate for all prototyping situations. What is required, therefore, is an understanding of the range of possible techniques for object definition and a set of good implementations for those techniques. This paper attempts to increase that understanding and also describes some example implementations and our experiences with them.

3. A RANGE OF TECHNIQUES FOR DEFINING OBJECTS

We are concerned here with the definitional information associated with a data object in a large prototype system. As mentioned above, we assume the use of a language like Ada, C++, Modula-2, or Trellis/Owl to describe that information. Those languages have within them the concepts of *abstract data type*, *module*, and *separate compilation*, all of which are important in prototyping. In Ada, for example, *packages* are modules and consist of a *specification part* and a *body part*. The specification part defines what is exported from, and imported into, the package and can be compiled separately from the body part. The body part provides the implementation of the package. The concept of abstract data type is captured in the package—that is, packages can be used to specify and implement abstract data types. For presentation purposes, we use Ada terminology and examples below.

The most basic questions that can be asked about definitional information are: where is it described and how is it accessed? Of course, different choices could be made for different portions of an object's definitional information, but we will assume for now that all information for a given object is treated the same. We have identified three choices and characterize them as follows:

—*Specification-described*. Definitional information about an object is explicitly captured in the immutable specification part of a package and can be referred to directly by clients of the package.

—*Implementation-described*. Definitional information about an object is described in the immutable implementation of a package 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.

We can further refine this coarse characterization, at least within each of the three choices described above (Table I). Under the specification-described approach, the definitional information can be presented in either an *abstract* or a *nonabstract* way. The former usually takes the form of a functional interface, in the style advocated by proponents of information hiding and data abstraction. The latter usually takes the form of explicit and visible data structure definitions. Under the implementation-described approach, the definitional information can be captured either as the values of *data structure* or as actual *code*. This is analogous to the distinction between a table-driven parser (e.g., one generated by Yace [6]) and a hard-coded parser. Finally, under the value-described approach, the definitional information about an object either can reside in a *separate structure* or it can form a part of the object itself and therefore be *self-describing*.

It is important to notice the rather subtle, but nonetheless significant, distinction that we draw between the two value-described techniques on the one hand and the data-based implementation-described technique on the other, since all three are based on the use of data structures. We have found it convenient to differentiate the approaches along two dimensions. The first is concerned with the mutability of the definitional-information data structure, while the second is concerned with where responsibility for interpreting the definitional information lies. Under the value-described techniques, as we define them, the definitional information is mutable at run time and responsibility for interpreting the

Table I. Range of Techniques

Specification-described		*Implementation-described*		*Value-described*	
abstract	non-abstract	code-based	data-based	separately-described	self-described

definitional information (e.g., to perform consistency checks) lies with the clients of an object. Under the data-based implementation-described technique, the definitional information is immutable at run time and responsibility for interpreting the definitional information is given to the package providing the definition. The full significance of this distinction becomes evident in the detailed evaluation presented in Section 5.

This characterization leaves us with six *basic* techniques from which to choose. But these six techniques are really only select points within the range; while they serve as convenient touchstones, it is possible to develop *hybrid* (or *enhanced*, or *extended*) implementations that lie at other points within the range. In fact, as our evaluation clearly demonstrates, it can be highly beneficial to do so. Examples of such hybrids appear in the next section, where we illustrate the range of techniques by describing three implementations that we have both developed and used.

Before introducing the implementations, we wish to preview the specific criteria used in the evaluation so that the reader can gain a feeling for what we consider to be most important. Here, we formulate the criteria as questions.

(1) Ease of Definition and Redefinition
  (a) How easy is it to develop a definition?
  (b) How easy is it to understand a definition?
  (c) How easy is it to modify definitional information?
    i. To locate the part(s) of a definition that need changing?
    ii. To make the change?
  (d) How quickly can a change to definitional information take effect?
  (e) How much code needs to be regenerated?
(2) Ease of Reuse
  (a) How easy is it to reuse an object (definition)?
    i. To identify a suitable candidate for reuse?
    ii. To make any necessary modifications?
  (b) How easy is it to develop general-purpose clients? That is, is there good support for reuse of clients?
(3) Consistency Management
  (a) How easy is it to check consistency?
  (b) When (how early) can an inconsistency be detected?
  (c) How reliably can inconsistency be detected?
(4) Controlled Impact of Change
  (a) How well can we limit the impact of a change to "interested" clients?
  (b) How accurately can we determine which clients are "interested"?

4. THREE IMPLEMENTATIONS OF OBJECT DEFINITION TECHNIQUES

As part of our work on a prototype software environment [15], we have been experimenting with a variety of techniques to facilitate our prototyping activities.

On reflection, we have recognized that three of these that we have used most extensively represent distinct approaches to supporting the development of large prototype systems, essentially based on the three major categories of techniques described in the previous section. They are:

—IRIS, a graph-based scheme (due to Fisher) for representing the semantics of software-system descriptions, such as specifications, designs, and programs, written in a formal language [1];

—GRAPHITE, a system that generates packages for manipulating directed graphs [2]; and

—PIC, a language framework and analysis technique for precisely describing and analyzing module interfaces [17, 18].

In this section, we describe in detail our implementations for these systems and briefly indicate their advantages and disadvantages, leaving a thorough evaluation for Section 5.

Throughout this section, a single example is used to demonstrate some of the relevant capabilities of the implementations. The example application is the development of an interface to a directed-graph data structure for representing program semantics in a software environment. This is a very realistic example, since many tools in a software environment would be expected to make use of such an interface. Of course, it is also a “common denominator”; while this is a characteristic application of IRIS, GRAPHITE can be used for any kind of directed graph and PIC can be used for interfaces to any kind of data object or module.

The terminology for graphs used below is as follows. A graph consists of a set of *nodes*, where each node is of some *node kind*. A set of node kinds is referred to as a *class*; a graph consists of nodes from one or more classes of node kinds. A node kind is associated with a set of *attributes*. Attributes are used to describe the properties of the objects represented by the nodes in the graph and each such attribute has a type, referred to as an *attribute type*. An instance of a node kind is a set of values, one for each attribute associated with that node's kind. Some of the attribute types are actually node kinds, which makes it possible to connect nodes into directed graph structures. In the example given here, node kinds are the only interesting attribute types employed.

To simplify the example, we restrict discussion to the process of developing a representation for an *if-statement*. Definition and redefinition, reuse, consistency management, and control over impact of change supported by the three implementations are demonstrated by considering what happens when the developer of the representation switches from one form to another. The first form, referred to below as if_1, is the standard *if-then-else* construct. The second form, if_2, accounts for the appearance of any number of specialized *else-if* clauses. The example is further simplified by assuming that the interface to the representation graph is to be in the form of an abstract data type realized as an Ada package, which we refer to as an *interface package*.

4.1 IRIS

IRIS, which stands for Internal Representation Including Semantics, is based on the use of abstract syntax graphs to capture the semantics of a software-system description in terms of expressions. IRIS represents the elements of a software-system description as literals and operators applied to a set of operand expressions. For example, the expression 2 + 3 is represented as the application of an addition operator to the literals 2 and 3. The if_1 form of *if-statement* might be represented as the application of an "if" operator to two operands, the first being an IRIS graph representing the *if-then* part of the statement and the second being an IRIS graph representing the *else* part of the statement. Figure 1 shows such a representation for the statement

if X = Y **then** . . . **else** . . . **end if**

where if, condition clause, list, and = are operators, X and Y are identifier literals, rectangles denote expressions, and circles denote references to literals.

There are only two node kinds in IRIS; one is used to represent expressions and the other is used to represent references to literals. Expression nodes include one attribute for referring to the declaration of the operator and an arbitrary number of other attributes for referring to the operands. Literal nodes for identifiers include an attribute for referring to the declaration of that identifier, while literal nodes for numbers and strings include an attribute for holding that number or string.

A key feature of IRIS that distinguishes it from other graph representations of semantics (e.g., TCOL [8]) is that the descriptions of all language-defined operators are themselves represented as IRIS graphs.[2] Each use of an operator is represented by a reference to an IRIS graph representing the declaration of that operator, which includes such information as the operator's name, the number of operands it takes, and the types of those operands. Thus, in Figure 1, "→ "if"" indicates a reference to the IRIS-encoded declaration of the "if" operator, which would specify its two operands. There is essentially no difference between the declarations for the language-defined operators, such as "if" and "list," and the user-defined procedures and functions that have been translated into IRIS. IRIS is therefore a (conceptually) self-describing, general-purpose structure for representing software-system descriptions. To represent the descriptions written in a particular language, one must provide declarations for the language-defined operators; different sets of these operators would of course yield different languages.

One can consider implementing IRIS in a number of different ways. For instance, in our implementation of IRIS [20], we chose to use a self-describing, mutable data structure, where interpretation of the definitional information is left to clients. Thus, our implementation uses a basic self-described value-described technique.

The Ada interface package used in our implementation of IRIS defines a type for nodes and defines the operations that allow manipulation of instances of the graph. The operations include those that allow clients to create and delete nodes, to get and put attribute values, and to read and write graphs. Nothing in this interface package, however, is specific to a particular language (i.e., the language-defined operators). Therefore, it would be possible for a client to use this interface

[2] Of course, the most primitive semantics of any IRIS description, namely function application and operand evalution are not represented in the graphs.

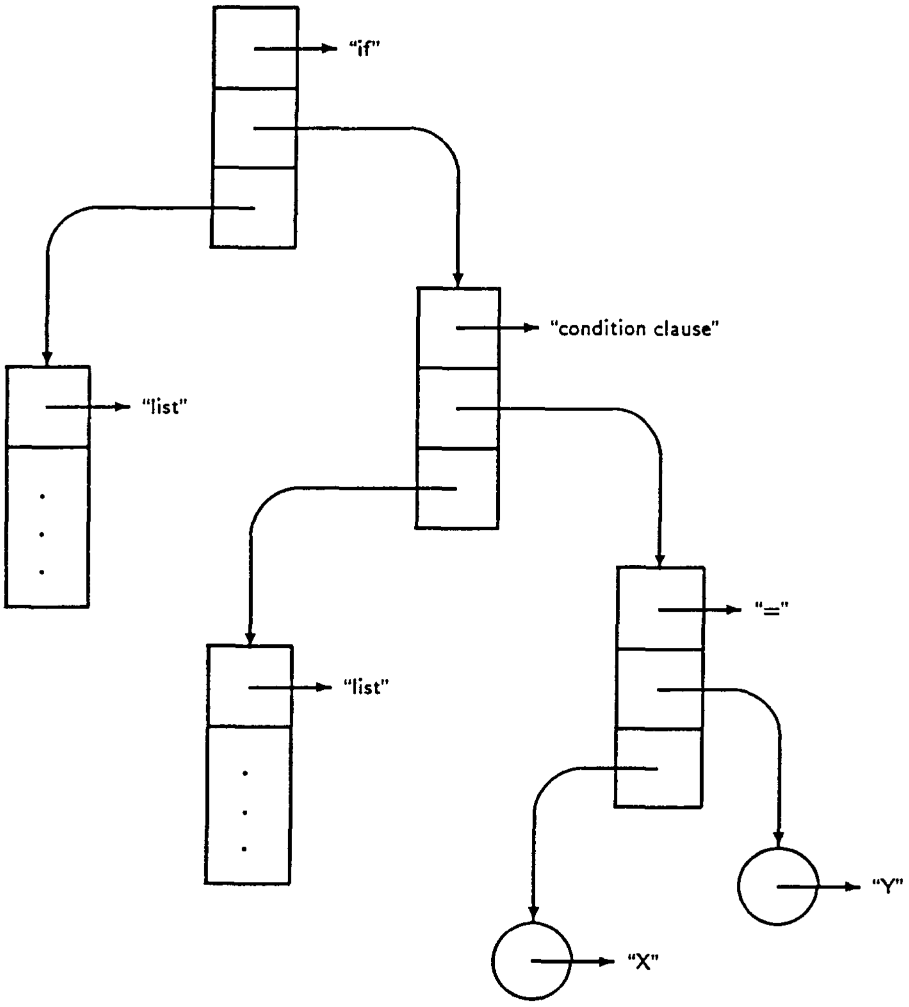

Fig. 1. Schematic example of IRIS representation for if_1.

package with any language. Moreover, the interface package would be immune to any changes to the operators of a given language. For example, changing from the if_1 form of *if-statement* to the if_2 form would involve a change to the declaration of the "if" operator, but not to the interface package, since the second form still uses the same two node kinds. Figure 2 shows the if_2 representation for the statement

if X = Y **then** . . . **elsif** X = Z **then** . . . **else** . . . **end if**

where the first operand of the "if" operator is evidently now a list of condition clauses, the first of which represents the *if-then* part. Because the interface package does not change, clients uninterested in a change will not be affected.

We are using our implementation of an interface to IRIS in the prototype development of several tools, including front-end tools (i.e., internal-represen-

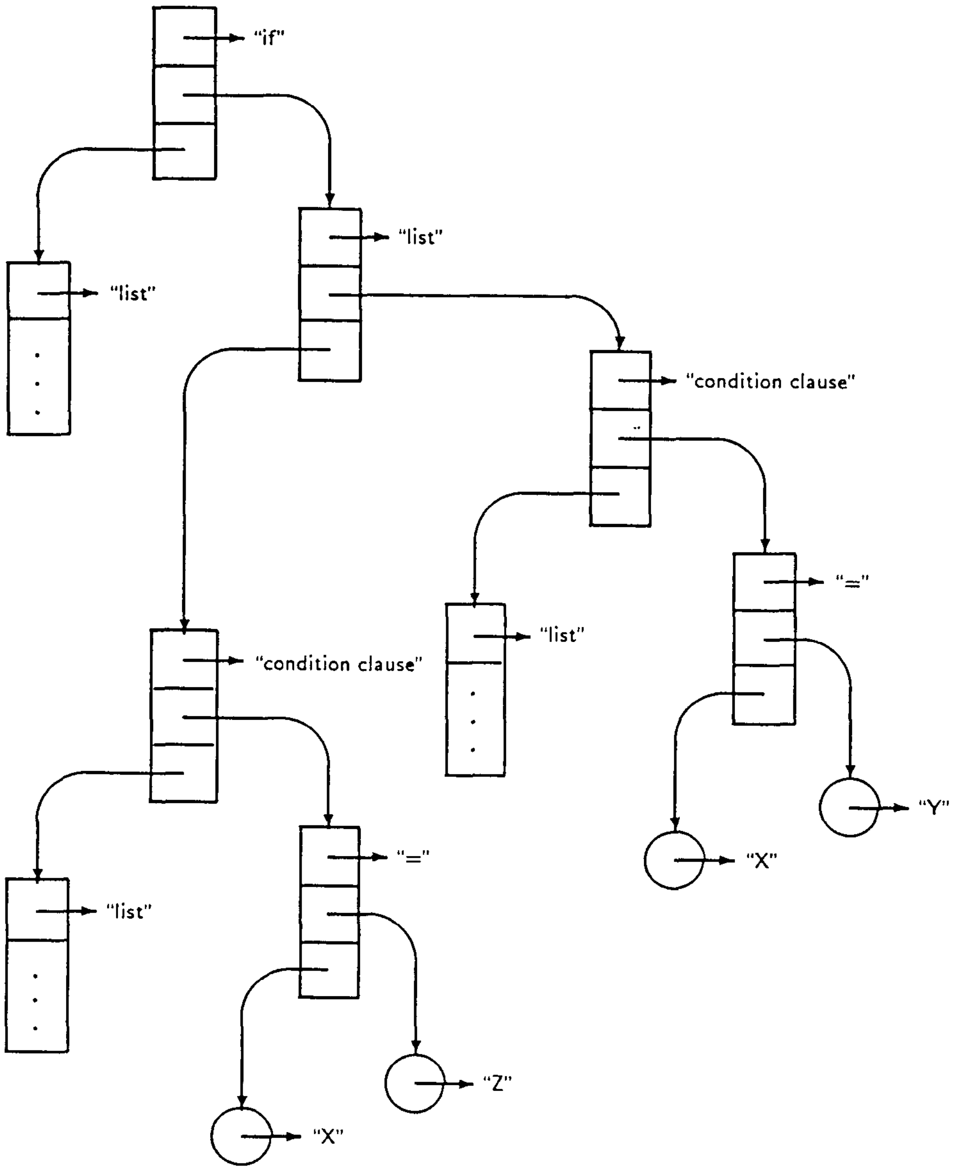

Fig. 2. Schematic example of IRIS representation for *if*$_2$.

tation generators, as well as lexical, syntactic, and semantic analyzers) for a variety of languages. A particularly interesting tool benefiting from the approach embodied in IRIS is a generic interpreter, called ARIES, which stands for Arcadia Interpretive Execution System [21]. ARIES has been designed with two goals in mind: to serve as a general-purpose interpretation engine for any IRIS-described language, and to allow the simultaneous interpretation of a program using a variety of execution models, such as symbolic execution and dynamic data-flow tracking, as well as conventional actual-value execution. IRIS has not only made it easier to build and test the interpreter incrementally, but it contributes to the generic nature of ARIES by limiting the impact on interpreter components of changes to language semantics.

There are several advantages to the value-described approach typified by our implementation of IRIS. Foremost, it limits the impact of change. The language being represented by IRIS can change and, as long as the client tools always interpret the definitional information, those tools do not need to be recoded or recompiled. This also facilitates reuse of clients, since some of these tools might be generic tools (e.g., ARIES) that work on different languages. Of course, there is a price for this much flexibility. In particular, no static type checking can be done to assure, for example, that only "if" information is put into an "if" node. Also, unless adequate external documentation is provided, it may be very difficult to understand the information content of the object. This lack of visibility can have a negative impact on software reuse.

4.2 GRAPHITE

Many of the data objects manipulated by software environment tools are graphs. For example, parse trees, abstract syntax trees, control flow graphs, and call graphs are all graphs that are likely to be manipulated by tools in an environment; IRIS is another such graph. We have therefore placed considerable effort into a general design for interfaces to graph objects.

Figure 3 illustrates the basic form of the interfaces that we have adopted for some of our graph objects by showing a skeleton of the interface-package specification part for a version of our representation-graph example. As can be seen, the interface package implements an abstract data type for classes of node kinds by providing a set of general-purpose access routines, such as GetAttribute and PutAttribute. These routines are tailored by parameters supplied by the clients invoking them, such as a parameter to specify the desired node kind for a "create" operation. In most cases, these parameters are character strings that must be interpreted in the body of the interface package. If we assume, for example, that there are node kinds for representing an *if-statement* and a *condition-clause*, then character strings such as IfNode and ConditionClauseNode might be used to identify them.

Under Ada's recompilation rules (as for other compiled, strongly typed, statically type-checked languages, such as C++, Modula-2, Trellis/Owl, etc.), recompilation of the clients of a package is avoided only if the specification part of that package does not itself require recompilation after a change has been made. Toward this end, the interface-package specification part is nearly devoid of all definition- and representation-specific information about the node kinds being managed and so insulates clients of that package from most changes in class definitions and representations. For example, in the specification part shown in Figure 3, there is no mention of particular node kinds, such as IfNode and ConditionClauseNode. Indirection through access types allows the details of the representation of a node to be confined to the body part of a package. Once there, those details can be changed without affecting the specification part of the package and, by extension, the clients using that package. In Figure 3, the private

```
package RepInterface is
    -- node-handle types
       type RepGraph is private;
       NullRepGraph : constant RepGraph;
       . . .
    -- user-defined attribute types
       . . .
    -- types for communicating names
       type NodeKindName is new String;
       type AttributeName  is new String;
       . . .
    -- types for listing a node's attributes
       type AttributeNamePointer is access AttributeName;
       type AttributeNameList     is array ( Positive range <> ) of AttributeNamePointer;
    -- operations to manipulate a node
       function   Create ( TheNodeKind : NodeKindName ) return RepGraph;
       procedure DeleteNode ( TheNode : in out RepGraph );
       procedure PutAttribute ( TheNode : RepGraph; TheAttribute : AttributeName;
                                TheValue : RepGraph );
       function   GetAttribute ( TheNode : RepGraph; TheAttribute : AttributeName )
                                return RepGraph;
       . . .
    -- operations to ascertain a node's definition
       function Kind ( TheNode : RepGraph ) return NodeKindName;
       function NodeKindAttributes ( TheNodeKind : NodeKindName ) return AttributeNameList;
       . . .
    -- operations to input and output graphs
       procedure ReadGraph ( FileName : String; TheGraph : in out RepGraph );
       procedure WriteGraph ( FileName : String; TheGraph : in out RepGraph );
private
    -- representations; complete declarations given in body part
       type RepGraphRep;
       type RepGraph is access RepGraphRep;
       NullRepGraph : constant RepGraph := null;
       . . .
end RepInterface;
```

Fig. 3. Skeleton of interface-package specification part for representation-graph example (implementation-described technique).

type RepGraph is shown to be an access type that designates the incomplete type RepGraphRep. The full declaration of RepGraphRep, which defines the actual data structure for representing nodes, would appear in the body part of package RepInterface.

It is important to point out that although one type (e.g., RepGraph) is used to designate nodes of all kinds, the interface package will guarantee at run time that a node is used in a manner consistent with its kind. For instance, if a node kind has an attribute A whose type is another node kind NK, then only nodes of kind NK will be allowed as values of attribute A.

It should be evident that in implementing the form of interface packages discussed here, we have primarily used the implementation-described approach.

```
class RepGraph is
  package RepInterface;

  . . .

  node ConditionClauseNode is
     Operator   : DeclarationNode;
     Condition  : ExpressionNode;
     Statements : ListNode;
  end node;

  node IfNode is
     Operator           : DeclarationNode;
     ConditionClause    : ConditionClauseNode;
     ElsePartStatements : ListNode;
  end node;

  node IdentifierNode is
     Identifier : DeclarationNode;
  end node;

  . . .

end RepGraph;
```

Fig. 4. Portion of GDL specification for class to represent if_1.

In particular, definitional information is confined to the implementation part of an interface package and referred to through the parameters of general-purpose access routines. But it should also be evident that there are certain aspects of the interface package that are characteristic of the value-described approach, which means that our implementation is in fact a hybrid. For instance, there is a set of operations provided to allow a client to ascertain, although not alter, the kind of a node and its associated attributes, which amounts to a run-time interpretation of the definitional information, as can be done under the value-described approach.

As discussed in Section 5, a potential problem with the implementation-described approach is the effort involved in developing or modifying definitional information. To help alleviate this problem for the interface packages discussed above we developed the GRAPHITE system which is similar in some respects to IDL [7, 13]. GRAPHITE, which stands for GRAPH Interface Tool for Environments, accepts specifications of classes of node kinds written in the graph description language GDL. Given the GDL specification for a particular class of node kinds, GRAPHITE automatically produces an interface package for manipulating nodes of those kinds.

A given GDL specification defines a particular class by declaring the node kinds, attributes, and attribute types making up the class. Figure 4 shows a portion of a GDL specification for some of the graph nodes for representing if_1, the form of *if-statement* that cannot contain *else-if* clauses. In Figure 4, RepGraph is the class being defined and RepInterface is the interface package (Figure 3) that is to be generated by GRAPHITE.

As an illustration of how GRAPHITE-generated interface packages can limit impact of change, consider what happens as a result of changing from the if_1

representation to the if_2 representation by replacing the definition of node kind IfNode in Figure 4 with the following definition:

```
node IfNode is
  Operator            : DeclarationNode;
  ConditionClauseList : ListNode;  --includes "if" and "else-if's"
  ElsePartStatements  : ListNode;
end node;
```

Although the definition of a node kind has changed (the second attribute of IfNode has had both its name and attribute type changed), the specification part of what would be the new interface package is *identical* to that of the old one; the necessary changes are confined to the body part. Thus, clients uninterested in this change need not be affected.

GRAPHITE actually produces two different kinds of interface packages. One, referred to as the *development interface*, is intended to support experimental systems and is the one discussed up to this point. It is designed so that when developers modify the definition of a class, there is a minimal effect on other components in the system, even on those components that use the modified class. The second kind of interface package, called the *production interface*, is designed for efficient manipulation of nodes. When the definition of a class has become relatively stable, the second interface package can be easily substituted for the first (with virtually no recoding of clients being required) so that a more efficient, although less flexible, version of the system can be created.

The difference between the two kinds of interface packages comes down to the balance between definitional information captured using the implementation-described approach and that captured using the specification-described approach. In particular, shifting from the development interface to the production interface means a concomitant shift of some of the definitional information from the body part of the interface package into the specification part. This happens, for example, to the information about the names of node kinds and attributes; in production interface packages, these names are represented as enumeration literals defined in the specification part instead of as character strings defined by clients and interpreted by the interface package. Placing the information into the specification part in an appropriate form makes it possible for the compiler to take advantage of that information when performing type checks and generating code, resulting in the greater efficiency exhibited by production interface packages. Of course, production interface packages also exhibit less flexibility.

GRAPHITE has been used extensively in the development of the graph data structures of several environment tools, including front-end tools for a number of languages, a suite of interface analysis tools, a loop analyzer, and even the implementation of GRAPHITE itself. One of these tools is Athena, a table-driven internal-representation generator, lexical analyzer, and syntactic analyzer for Ada. In all, Athena consists of 23 separately compilable units that total over 750 kilobytes of source code. Compilation of a moderate-size program such as this takes a substantial amount of computer time and, perhaps more importantly, programmer time. The component of Athena that generates GDL-specified program-representation graphs was actually developed incrementally by successively handling larger and larger subsets of the Ada language. Growth from one subset to the next often involved changes to the definition of the program-representation graph. The only part of the program interested in such changes was the set of so-called "actions" that are performed by the tool; these actions are embodied in a single compilation unit, called Actions. By using the development interface package generated by GRAPHITE, we were able to minimize the recompilation necessitated by changes to the definition of the program-representation graph. In particular, only the body part of the interface package (and, of course, Actions) had to be recompiled; the other units in the program were insulated from such changes. This reduced recompilation time by over half as compared to what would have been required if the definition of the program-representation graph had been exposed.

GRAPHITE facilitates development of large prototype systems in several ways. Most importantly, the design of the generated development interface package, in which definitional information is confined to the body, insulates clients from changes in the definition and representation of a class of node kinds. This approach sacrifices static checking in favor of minimizing the impact of change. It still permits an interface package to enforce the consistency of the specified class definition, however, since the body part contains all the information necessary to check at run time the legality of node kind and attribute names as well as the operations applied to instances. In addition to using the implementation-described approach, GRAPHITE provides some other capabilities that foster prototyping. Specifically, it facilitates reuse by automating the creation of an abstract data type for a user-specified class of node kinds and by providing, through GDL, good documentation of the graphs used in a system.

4.3 PIC

The previous two implementations primarily illustrate the value-described and implementation-described approaches. What remains is to illustrate the specification-described approach, where the definitional information is captured in the specification part of a package. As mentioned in Section 2, there is a sense in which this is the most "obvious" approach to use in languages like Ada, C++, Modula-2, and Trellis/Owl.[3]

Figure 5 shows one possible use of the specification-described approach for the interface to our representation-graph example. Specifically, it shows a use of the abstract-based-interface technique, where each node kind and each attribute used in the representation graph has associated with it an appropriate set of subprograms (i.e., operations), such as to create a node of a particular kind or to get a value of a particular attribute in a node.

As we point out in Section 2, having an information-rich interface means that it is easier to reuse an object definition, as well as to statically check appropriate use of an object by that object's clients, but it also means that it severely increases the impact of change. We examine these issues fully in Section 5. Here we describe an enhancement to the basic technique that can help limit the impact of change. The enhancement is based on the use of an interface control mecha-

[3] This is the (controversial) technique used to exemplify interfaces to Diana, an internal representation for Ada, that appeared in [4].

```
package RepInterface is
        type RepGraph is private;
        NullRepGraph : constant RepGraph;
        . . .
        function   CreateIfNode return RepGraph;
        procedure  DeleteIfNode ( TheNode : in out RepGraph );
        procedure  PutConditionClause ( TheNode : RepGraph;
                                        TheValue : RepGraph );
        function   GetConditionClause ( TheNode : RepGraph )
                                        return RepGraph;
        procedure  PutElsePartStatements ( TheNode : RepGraph;
                                           TheValue : RepGraph );
        function   GetElsePartStatements ( TheNode : RepGraph )
                                           return RepGraph;
        . . .
private
        . . .
end RepInterface;
```

Fig. 5. Portion of interface-package specification part for if_1 form of representation graph (specification-described technique).

nism that can distinguish between clients interested in a change and clients not interested in a change.

Interface control is concerned with describing and limiting the interactions that can occur between the entities in different modules of a software system. Entities are named language elements such as objects, types, and subprograms; a module serves to group together related entities. The interface control mechanism of a language is used to specify what (and sometimes how) entities within one module can be used by another module. Thus, given a suitably precise interface control mechanism—that is one that allows the description of module interactions to any desired level of detail—the extent to which a particular change to a module affects other modules can be easily determined. Once determined, this information can then be used, for example, by a recompilation tool to limit the impact of that change.

PIC, which stands for Precise Interface Control, is a research project aimed at improving support for interface control in large software systems. Results from this project include the design of a small set of language features for precisely specifying module interfaces and a collection of tools for analyzing those specifications for consistency. Prototypes of the analysis tools have been implemented for a family of PIC-oriented languages based on Ada [18]; the example below is given in one of these languages, namely PIC/Ada.

The conceptual foundation for the PIC language features is provided by a general view of interface control that is richer than views based solely on traditional entity-visibility concepts of declaration, scope, and binding. This view distinguishes two aspects of visibility:

(1) *requisition* of access; and
(2) *provision* of access.

```
package Client1Interface is
      request RepInterface.GetIdentifier, . . .;
      . . .
end Client1Interface;


package Client2Interface is
      request RepInterface.GetConditionClause, . . .;
      . . .
end Client2Interface;
```

Fig. 6. Portions of PIC/Ada specification parts of interfaces to two clients that use the interface of Figure 5.

Access to an entity is the right to make reference to, or use of, that entity in declarations or statements. Requisition of access occurs when an entity (implicitly or explicitly) requests the right to refer to some set of entities. Provision of access occurs when an entity (implicitly or explicitly) offers, to some set of entities, the right to refer to that entity. Given this view, an interface control mechanism is simply a means for specifying requisition and provision.

The PIC language features used to capture these two aspects of entity visibility are the *request clause*, for specifying requisition, and the *provide clause*, for specifying provision. They can appear only in the specification parts of modules, and therefore these parts act as a sort of "module interconnection language" for software systems (cf., [5]).

Request and provide clauses can be used in a variety of ways to express the relationships among modules. In particular, notice that request clauses are akin to capabilities in operating systems and, similarly, provide clauses are akin to access lists. Just as there are situations where use of capabilities is more appropriate than use of access lists, and vice versa, there are situations where use of one clause is more appropriate than use of the other. Having both clauses available in a language allows extreme flexibility in the description of interface relationships. In addition, support for both can result in a redundancy that facilitates more rigorous analysis of the interface relationships of a system's components. For example, based on this view it is possible to formulate complementary descriptions of exactly how two modules are intended to interact, giving one description from the perspective of each of the modules, and then to analyze those interactions by checking the two descriptions for consistency.

Figure 6 shows one possible use of the PIC language features for describing the relationship between the representation-graph interface and the clients of that interface. Using request clauses in a "capability" style, each client's specification indicates exactly those parts of the representation interface in which it is interested. Thus, it is clear that when the developer alters RepInterface to work with if_2, Client 2 and not Client 1 is interested in the change.

Using the basic abstract specification-described approach, but enhancing it with the interface control constructs provided by PIC, has several advantages. Clearly indicating which modules must be recompiled when a change in a specification occurs makes it possible for "smart" compilers to significantly limit the impact of change, recompiling only when a client actually uses the changed object. Moreover, meaningful static type checking can be performed. Finally, this approach aids reuse by explicitly providing definitional information in the specification part of the interface.

5. COMPARATIVE EVALUATION

In the preceding sections we have defined a range of object definition techniques for large prototype systems and described our implementations of three points within that range. We now offer a comparative evaluation of the various points within the range, based in part on extrapolations from our experiences in designing and using the particular implementations described in Section 4. We begin with a detailed evaluation, in which each technique is measured against each of the questions listed at the end of Section 3. We then summarize our observations, distilling the detailed evaluation into rankings of the various techniques against a set of more general properties implied by the list of questions. Finally, we consider how extensions or enhanced implementations, like those represented by our GRAPHITE system or PIC toolset, can affect the suitability of some of the techniques relative to certain of the properties important for large prototype systems.

5.1 Detailed Evaluation

Table II presents our detailed evaluation of the six basic object definition techniques identified in Section 3. The rows correspond to the questions listed at the end of that section. The columns correspond to the techniques. The numerical entries represent our comparative evaluation of each technique in terms of each question. The numerical scores are intended to express relative, not absolute, rankings, and hence are not comparable across questions (i.e., between rows). A rank of "1" is considered best.

Definition and redefinition. The easiest way to develop the definition of an object is to use the primitive mechanisms provided in the language in which the prototype is being programmed. For the class of compiled, strongly typed, statically type-checked languages that we have taken as our starting point, this would typically mean constructs such as array or record. Developing a more abstract definition, with a functional interface, generally requires additional effort, such as writing the procedure bodies corresponding to the interface functions. Implementation-described or value-described object definitions require even more effort to develop, since a general-purpose interface, which includes special functions and data structures, must be developed. Hence the one-two-three ranking of these techniques in the first row of Table II.

A virtue of information hiding and data abstraction is that object definitions are easier to understand when these methods are used, since irrelevant details of implementation are suppressed. Hence, on the ease of understanding, abstract specification-described object definition techniques rank above nonabstract. Humans generally find interpreting the information in a code-based or a data-structure-based object definition much more difficult than understanding either of the specification-described techniques. Since understanding descriptions is

Table II. Detailed Evaluation

	Specification-described		*Implementation-described*		*Value-described*	
	abstract	non-abstract	code-based	data-based	separately-described	self-described
Definition & Redefinition						
ease of development	2	1	3			
ease of understanding	1	2	3			
ease of change — finding where	1	2	3			
ease of change — making change	3	2	3	1		
quickness of change effect	3			2	1	
amount of regenerated code	3			2	1	
Reuse						
of object — identifying	1	2	3			
of object — what to modify	1	2	3			
of object — making change	3	2	3	1		
of general-purpose clients	2				1	
Consistency Management						
ease of checking	1		2		3	
earliness of detection	1		2			
reliability of detection	1		2		3	
Controlling Impact of Change						
limit to "interested"	3	4	2		1	
determine "interest"	1		2		3	

fundamental to determining where a change should be made during modification of an object definition, the same ranking applies both for ease of understanding and for this aspect of ease of change.

On the other hand, changing the values in a data structure is certainly the easiest way to actually carry out a modification to an object definition, whether that change is made dynamically, by altering values in a running prototype, or statically, by changing data initialization statements. Changing a nonabstract specification-described definition is next easiest (e.g., simply changing a record declaration), while both abstract specification-described and code-based implementation-described definitions require code modifications, making them the most difficult.

Value-described object definitions also rank highest for how quickly a change to a definition will take effect and how little code must be regenerated as a result of a change. In fact, value-described techniques are optimal in these respects, since changes can take effect immediately and no code need be regenerated. Changing the immutable (during prototype execution) data structures employed in data-based implementation-described object definitions simply requires revising initialization statements, so this class of techniques approaches the optimum. Changes in specification-described or code-based implementation-described object definitions all require significantly more code regeneration, and hence are the slowest to take effect.

Reuse. Object-definition techniques affect the reusability of object definitions by influencing how easy it is to identify suitable definitions for reuse and how easy it is to modify definitions for use in a new context.[4] The first of these is similar to the ease-of-understanding property considered previously, and hence produces similar rankings of the techniques. In particular, specification-described techniques make information about the structure and function offered by an object clearly and easily visible in the specification, and thus they rank highest here. The second is similar to the ease-of-change property considered in connection with definition and redefinition. Hence, we distinguish the same two aspects of modifying for reuse that we did for ease-of-change, and assign the techniques the same rankings for those aspects.

The value-described object definition techniques offer some unique support for reuse of an object definition's clients, since in some instances a client may be reusable with no changes at all, despite a change in the definitional information contained in the data structure. Client components of this kind are typically general-purpose utilities (e.g., ARIES) that are designed to base their actions on the description contained in the data structure. Such components offer the ultimate in reuse. Among the six basic object definition techniques, neither the implementation-described nor the specification-described techniques offer similar support for reuse of general-purpose clients, and hence they rank lower with respect to this property.

Consistency management. Consistency management is stronger in the specification-described techniques than in either of the other two. Verifying that uses

[4] We do not consider in this paper how, if at all, the techniques address the difficult problem of continuing to use instances of an object definition after that definition has been modified.

of a data object are consistent with that object's definition amounts to type checking. In the specification-described techniques, type checking can be static and strong, with type errors producing compile-time error notification. From the point of view of the developer of the object definition, this provides the easiest checking, since it does not require the writing of any consistency checking code. It also results in the earliest possible detection, since inconsistencies can be detected and reported at compile time. It is the most reliable form of consistency checking since it depends on established type checking utilities in the language processing system rather than any user-supplied checking code.

Under the code-described approach, consistency management is not static but dynamic. Thus consistency checking code must be created as part of the object definition, making consistency management both more difficult and less reliable. Because consistency checking is dynamic, type errors will lead to run-time exceptions, rather than compile-time error notifications, and hence will not be detected as early as with a specification-described technique. Since type checking can be centralized in the code that implements the object's definition (or that interprets the data structure describing the object's definition), consistency management is stronger than in the value-described approach. The fact that specifications are static, and hence cannot change during prototype execution, precludes the possibility that objects may become inconsistent with the definition and with each other during execution.[5]

Under the value-described approach, consistency management on uses of a data object is again not static but dynamic. Thus, as with the code-described techniques, type errors will lead to run-time exceptions, rather than compile-time error notifications, and hence will not be detected as early as with a specification-described technique. Moreover, since type checking is typically decentralized in this approach, being left to each individual client of a given data object, consistency management may be nonexistent in some cases. Although some checking can be built into an object's interface, in general only a client component can ensure that it is making correct use of an object, through interpretation of the definitional-information data structure. The fact that definitional information is dynamic and may change during prototype execution introduces the possibility that objects may become inconsistent with the definition and with each other during execution, further complicating consistency management. Hence, both ease of checking and reliability of detection rank lower for the value-described techniques than for any of the others.

Controlling impact of change. The specification-described techniques are least successful at limiting the impact of a change in an object's definition to only those components of a prototype system that are interested in the change. This is because when the specification of a data object is modified, most language processing systems (typically compilers) for compiled, strongly typed and statically type-checked languages will require that all tools and objects that refer to, or worse, that might possibly refer to, the modified object be type-checked again,

[5] Note again that we are discussing instance/definition consistency within a given prototype execution, not consistency of instances created during one execution with definitions that are in use during some subsequent execution.

which usually means recompiled. Of course, for nonabstract specification described techniques, a change to the representation of an object, even if that change does not otherwise alter the object's definition, will have this effect. Hence the nonabstract techniques receive an even lower rating than the abstract techniques here. On the other hand, determination can be made of what components are interested in a change to the object's definition (e.g., deletion of an information field in a data object) via a cross-reference analysis, so determining which ones are interested is straightforward.

The code-described approach limits the impact of change by encapsulating changes within the code bodies that implement the object's definitional information (or interpret the data structures describing the object's definitional information). The assumption is that only interested components will invoke these code bodies and hence that impact of a change will be limited to interested components. Determination of what components are interested in a change to the kind of information contained in the data object (e.g., deletion of an information field) only requires inspection of the calls to the relevant code bodies and the values of the parameters to those calls. Although not always trivial, this is simpler than the corresponding analysis for value-described data objects. In sum, impact of change is limited to only those components interested in the change, i.e., those who call the relevant code bodies with relevant parameter values, but determining which ones are interested may be somewhat complicated.

The value-described approach limits the impact of change by encapsulating all information about the change within the mutable data structure containing definitional information. The assumption is that only those components interested in the change will interpret the relevant section of this data structure, and hence the impact of the change will be restricted to interested components, as desired. Determination of what components are interested in a change to the definitional information concerning the data object (e.g., deletion of an information field) may require an interpretive trace of component behaviors to find which ones refer to the relevant part of the data structure. Thus, although impact of change is limited to only those components interested in the change, determining which ones are interested is extremely complicated.

5.2 Summary of Comparative Evaluation

Table III represents a summary of the detailed comparative evaluation that we have just presented. Here we have restricted our attention to the coarse characterization of object definition techniques as either specification-, implementation- or value-described. We also cluster the properties described by the fifteen questions of our detailed evaluation into three, more abstract properties. Once again we have used numerical scores that express only relative rankings within a row, not absolute rankings in any sense.

The first row, labeled Development and Reuse Effort, summarizes the properties concerning development and reuse except for those that involve modifying an object's definition. Thus, this row ranks the various techniques according to their ease of development, ease of understanding, ease of identifying candidates for reuse, and ease of identifying what needs to be changed, whether in the context of prototype modification or object reuse. The only result from our

Table III. Summary of Evaluation

	Specification-described	*Implementation-described*	*Value-described*
Development & Reuse Effort	1	2	2
Turnaround Time	3	2	1
Consistency Management	1	2	3

detailed evaluation that does not fit with the summary presented in this row is the ranking on reuse of general-purpose clients, an issue that we address below.

The second row, labeled Turnaround Time, summarizes the properties concerning modifications to object definitions. Thus, this row ranks the various techniques according to the ease of actually making a change, how quickly changes take effect, how much code must be regenerated due to a change and how well the impact of change can be controlled. One aspect of controlling the impact of change, namely the ease of determining which other components of a prototype are interested in a change, is not accurately reflected in the summary presented in this row, however. While this discrepancy is worth noting, we do not feel that it is significant enough to alter the overall ranking of the techniques with respect to turnaround time.

The third row, labeled Consistency Management, ranks the various techniques according to how easily consistency can be checked, how early inconsistency can be detected and how reliable consistency checking can be.

The conclusions that can be drawn from this summary seem to be the following:

(1) The implementation-described and value-described techniques would be more valuable for large prototyping efforts if they could be augmented to make development of object definitions easier.

(2) The specification-described and implementation-described techniques would be more valuable for large prototyping efforts if the turnaround time associated with them could be reduced.

(3) The implementation-described and value-described techniques would be more valuable for large prototyping efforts if their support for consistency management could be improved.

We now consider the extent to which extensions or enhanced implementations can alter the relative rankings of these various object definition techniques.

5.3 Effect of Extensions and Implementations

Table IV represents the potential effects of extensions and enhanced implementations on the summarized comparative evaluations presented in Table III. Again, we have restricted our attention to the coarse characterization of object definition techniques as either specification-, implementation- or value-described and clustered the properties described by the fifteen questions of our detailed evaluation into three, more abstract properties. And again, we have used numerical scores that express only relative rankings within a row, not absolute rankings in any sense.

The greatest potential effect of extensions and enhanced implementations is on the properties that we have summarized in the row labeled Development and

Table IV. Potential Effects of Extensions and Enhanced Implementations

	Specification-described	*Implementation-described*	*Value-described*
Development & Reuse Effort	1	1	1
Turnaround Time	2	2	1
Consistency Management	1	2	3

Reuse Effort. As noted in our detailed evaluation, the implementation-described and the value-described techniques do not automatically provide a clear, easily visible specification of an object's definition. Such information can only be obtained by interpreting the code and/or data structures that embody the object definition. Appropriate support tools can circumvent this shortcoming, however, making development and reuse of implementation-described and value-described object definitions as easy as that for specification-described object definitions. Our GRAPHITE system, for example, is a realization of the implementation-described technique augmented to provide automatic creation of object definition implementations (interface packages) from a human-readable form of the definitional information, namely GDL. Thus, GRAPHITE overcomes several aspects of the implementation-described techniques that could impede development and inhibit reuse. The fact that our current implementation of IRIS does not provide similar support for human-readable versions of object definitions or for generating the value-described representations has proven an impediment to both development and reuse of object definitions in our large prototyping efforts. It is clear, however, that such support could be implemented, in a manner similar to facilities provided in our GRAPHITE implementation. Hence we conclude that the properties summarized under the heading Development and Reuse Effort need not influence a choice between object definition techniques for large prototype systems.

As noted above, the ranking on reuse of general-purpose clients presented in our detailed evaluation was not accurately reflected in the summary represented by Table III. In particular, this was the one property under the Development and Reuse Effort heading that favored the value-described techniques over the others. This discrepancy disappears when we consider potential extensions and enhanced implementations, however. For example, our GRAPHITE implementation is augmented with definitional information that a client component may choose to interpret, and hence does support reuse of general-purpose utilities that are designed to base their actions on such a description. Similarly, a specification-described object definition could certainly be extended to provide functions that returned definitional information in a form that a client could interpret. In other words, it is entirely possible to simulate this feature of a value-described object definition in any of the other object definition techniques. Hence this property, like the others collected under the heading Development and Reuse Effort, need not influence a choice between object definition techniques for large prototype systems.

The other area in which extensions and enhanced implementations can have an effect is turnaround time. Changes to code, or to the definitions of immutable data structures that are interpreted by the code implementing an object definition, will necessarily induce some type-rechecking and code-regeneration delays that simply modifying a mutable data structure will not. Hence, the value-described techniques retain the top ranking for how easily a change can be made, how quickly it can take effect and how little code regeneration it causes. Enhanced implementations of the specification-described techniques, however, can make their performance in these areas essentially as good as that of the implementation-described techniques, tying them for second place.

In particular, the main reason that the specification-described techniques rank lower in turnaround time than the implementation-described techniques is that they are not as good at limiting the impact of change. This can be overcome through "smarter" compilation tools that only recheck type consistency and regenerate code for those clients *actually* affected by a modification to an object definition rather than, as is currently standard, all clients *potentially* affected. Fundamentally, this requires that the compilation tools have access to more detailed information about how objects and their clients interact. In keeping with our terminology from Section 4.3, we refer to such information as *interface control* information. Given that compilation tools have available and can exploit more detailed interface control information, developers of large prototypes can limit the impact of change by restricting the set of clients affected by a change to only those that are actually interested in that change.

We distinguish two different approaches to providing the more detailed interface control information. One is the *explicit* approach, represented by our PIC language constructs and tools or by Inscape [10]. The other is the *implicit* approach, as employed, for example, by Tichy in his work on "smart recompilation" [16]. As shown in Section 4.3, the explicit approach allows the developer to indicate intended interactions among the components of a prototype. This information can then be analyzed (e.g., by the PIC analysis tools) for such properties as consistency, and can also be used by an appropriately "smart" compiler to restrict rechecking of type consistency and regeneration of code during a recompilation. The implicit approach is based on a very detailed cross-reference analysis that determines such things as which aspects of an object's definition are being referenced and how, in addition to determining what system components are making those references. The results of this analysis can then be used, just as the explicit information could, by an appropriately "smart" compiler to restrict rechecking of type consistency and regeneration of code during a recompilation. While we favor the greater control over impact of change, and added error detection opportunities, offered by the explicit approach, obviously both provide the same substantial improvement in turnaround time.

It must be noted, however, that while these enhanced implementations of the specification-described techniques can equal the implementation-described techniques at limiting the impact of change, they will still lag slightly in terms of how quickly a change can take effect. This is because they will still involve more compile-time overhead, due in part to the more extensive type checking made possible by their information-rich interfaces and in part to the analysis involved in creating the interface control information. Of course, the compile-time overhead is still very much less than would accrue in the absence of that information, and should be only marginally more than the implementation-described techniques will require.

The differences in the row labeled Consistency Management seem to be inherent properties of the techniques and hence are not susceptible to modification through extensions or enhanced implementations. Dynamic consistency checking is necessitated by both the implementation-described and the value-described techniques. As a result, both must detect consistency errors at run time and hence cannot equal the early detection of the specification-described techniques. The value-described techniques must ultimately depend on decentralized consistency checking, performed by the clients themselves, and must also cope with the possibility of object definitions changing during prototype execution. Both of these factors make consistency checking more difficult and less reliable.

6. SUMMARY AND CONCLUSIONS

In this paper, we have discussed some distinguishing features of large prototype software systems, identified some requirements for object definition techniques for such prototypes, and characterized and compared a range of potentially suitable techniques. We have outlined our implementations of and our experiences with three such techniques to illustrate the range of possibilities and to support our evaluation.

Our comparative evaluation does not suggest that any of the techniques we have considered is clearly preferable to any other. A choice among them must depend upon the relative importance assigned to the various properties against which we compared them (and possibly some others that we did not include but are of special importance for some specific prototyping application). For example, those who consider turnaround time to be of overriding importance might find the value-described techniques irresistible.

For our particular prototyping applications, we find the level of consistency checking available with the value-described techniques to be unacceptable. On the other hand, we consider the level of consistency checking available with the implementation-described techniques to be a reasonable tradeoff for an improvement in turnaround time. Finally, we do not have available the enhanced implementation necessary to achieve a comparable turnaround time for specification-described techniques (e.g., a "smart" Ada compiler that could exploit our PIC/Ada interface control information). Therefore, we are currently relying primarily on an implementation-described technique, delivered through an enhanced implementation that provides good support for development and reuse, namely GRAPHITE. Clearly, however, the availability of a different set of options (e.g., a sufficiently "smart" Ada compiler) could lead us to a different choice.

Many languages and tools have been implemented and used to support prototyping. Examples include such standards as LISP, Smalltalk, PROLOG, and YACC, as well as more recent efforts such as the Cornell Synthesizer Generator [11], the SARA Interface Specification System [19], DOSE [6], and others. Many of these have included one or more object definition techniques. For instance, the structure editor generator DOSE essentially uses a value-described technique, for representing abstract syntax trees, as a means to support its interpretive/interactive style of editor development. Our goal in this paper has not been to present or argue for specific implementations of techniques. Rather, it has been to offer a basis on which such techniques can be compared and evaluated. We hope that this will provide a foundation for improved understanding and more informed choices among both current and future techniques considered for inclusion in languages or tools intended to support prototyping.

Naturally, there are other dimensions to prototyping languages and tools besides their object definition techniques. An example is Notkin and Griswold's work on a software extension mechanism [9]. The thrust of their research is to support the incremental addition of functionality to programs written in a compiled language, thus attaining some of the benefits of interpreted languages with much less performance overhead. Their mechanism can be viewed as an approach to limiting the impact of adding procedures to a prototype system, where our work has focused on limiting the impact of change to objects in prototype systems. The two are thus complementary components of an emerging trend toward support of large-scale prototyping activities.

ACKNOWLEDGMENTS

Many people have been instrumental in some aspect of the work reported here. David Fisher was the originator of IRIS, and we appreciate the many hours he has spent discussing it with us. Steven Sykes has also been instrumental in refining IRIS. Steven Zeil has contributed to refining IRIS, and he and George Barbanis are responsible for our IRIS implementation. Peri Tarr, Mary Burdick, Robert Graham, and William Rosenblatt have all contributed to the development and implementation of GRAPHITE. Alan Kaplan and Walter Kopp implemented the PIC tool set.

REFERENCES

1. BAKER, D. A., FISHER, D. A., AND SHULTIS, J. C. *The Gardens of IRIS*. Incremental Systems Corp., Pittsburgh, Pa., 1988.
2. CLARKE, L. A., WILEDEN, J. C., AND WOLF, A. L. GRAPHITE: A meta-tool for Ada environment development. In *Proceedings IEEE-CS Second International Conference on Ada Applications and Environments* (Miami Beach, Fla., April 8–10, 1986). IEEE, New York, 1986, 81–90.
3. *Draft Report on Requirements for a Common Prototyping System*. R. P. Gabriel, Ed., U.S. Department of Defense (DARPA-ISTO), Nov. 1988. Appeared in *ACM SIGPLAN Not. 24*, 3 (Mar. 1989), 93–165.
4. EVANS, A., JR., AND BUTLER, K. J., Eds. *Diana Reference Manual* (*Rev. 3*). Tech. Rep. TL 83-4, Tartan Laboratories Inc., Pittsburgh, Pa., Feb. 1983.
5. HABERMANN, A. N., AND PERRY, D. E. System composition and version control in Ada, In *Software Engineering Environments*, North-Holland, New York, 1981, 331–343.
6. KAISER, G. E., FEILER, P. H., JALILI, F., AND SCHLICHTER, J. H. A retrospective on DOSE: An interpretive approach to structure editor generation. *Softw. Pract. Exper. 18*, 8 (Aug. 1988), 733–748.
7. LAMB, D. A. IDL: Sharing intermediate representations. *ACM Trans. Program. Lang. Syst. 9*, 3 (July 1987), 297–318.
8. LEVERETT, B. W., CATTELL, R. G. G., HOBBS, S. O., NEWCOMER, J. M., REINER, A. H., SCHATZ, B. R., AND WULF, W. A. An Overview of the production-quality compiler-compiler project. *IEEE Comput. 13*, 8 (Aug. 1980), 38–49.

9. NOTKIN, D., AND GRISWOLD, W. G. Extension and software development. In *Proceedings Tenth International Conference on Software Engineering.* (Singapore, April 11–15, 1988), Computer Society Press, Washington, D.C., 274–283.
10. PERRY, D. E. The inscape environment. In *Proceedings Eleventh International Conference on Software Engineering,* (Pittsburgh, Pa., May 1989). Computer Society Press, Washington, D.C., 2–11.
11. REPS, T., AND TEITELBAUM, T. The synthesizer generator. In *Proceedings ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (April 23–25, 1984). In ACM *SIGSOFT Software Eng. Not. 9,* 3 (May 1984), 42–48.
12. SCHAFFERT, C., COOPER, T., BULLIS, B., KILIAN, M., AND WILPOLT, C. An introduction to Trellis/Owl. In *Proceedings ACM Conference on Object-Oriented Programming Systems, Languages, and Applications: OOPSLA '86,* (Portland, Ore., Sept. 29–Oct. 2, 1986), ACM, New York, 1986, 9–16.
13. SNODGRASS, R. T. *The Interface Description Language: Definition and Use.* Computer Science Press, Rockville, Md., 1989.
14. SQUIRES, S., ZELKOWITZ, M., AND BRANSTAD, M. Eds. Special Issue on Rapid Prototyping: Working Papers from the ACM SIGSOFT Rapid Prototyping Workshop. *ACM SIGSOFT Softw. Eng. Not. 7,* 5 (Dec. 1982).
15. 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. In *Proceedings ACM SIGSOFT '88: Third Symposium on Software Development Environments* (Boston, Mass., Nov. 1988). In *SIGSOFT Not. 13,* 5 (Nov. 28–30, 1988), 1–13.
16. TICHY, W. F. Smart recompilation. *ACM Trans. Program. Lang. Syst. 8,* 3 (July 1986), 273–291.
17. WOLF, A. L., CLARKE, L. A., AND WILEDEN, J. C. Interface control and incremental development in the PIC environment. In *Proceedings Eighth International Conference on Software Engineering.* (London, Aug. 28–30, 1985). Computer Society Press, Washington, D.C., 75–82.
18. WOLF, A. L., CLARKE, L. A., AND WILEDEN, J. C. The AdaPIC tool set: Supporting interface control and analysis throughout the software development process. *IEEE Trans. Softw. Eng. 15,* 3 (March 1989), 250–263.
19. WORLEY, D. R., AND KRELL, E. User interface specification in the SARA design system. In *Foundation for Human-Computer Communication.* K. Hopper and I. A. Newman Eds., Elsevier Science, Amsterdam, 1986.
20. ZEIL, S. J. *An Iris Interface in Ada.* Arcadia Design Doc. UM-87-06, Univ. of Massachusetts, Amherst, Aug. 1987.
21. ZEIL, S. J., AND EPP, E. C. Interpretation in a tool fragment environment. In *Proceedings Tenth International Conference on Software Engineering* (Singapore, April 11–15, 1988). Computer Society Press, Washington, D.C., 241–248.

Received February 1989; revised May 1989; accepted September 1989.