Programmable Relations for Managing Change During Software Development

Stanley M. Sutton, Jr.   Dennis Heimbigner
Leon J. Osterweil*

CU-CS-418-88   September 15, 1988

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309-0430
(303) 492-7514

* Present address: Department of Information and Computer Science, University of California, Irvine, CA 92717.

Abstract

Management of changes during software development is a central problem in software object management. Successful propagation of changes requires effective use of relationships among objects. The structure of relationships identifies the direction, order, and extent of change propagation, while the semantics of relationships indicate the kinds of changes to be made.

Many data models proposed for software object management offer some useful features for controlling change but lack others. Configuration control and version management systems support some useful relationships and types but are too specialized. Object-oriented systems are more general, but while they may support derived relationships they represent those relationships only as attributes of objects. Conversely, entity-relationship models provide for the independent representation of extensional relationships but typically do not support derived relationships fully. The relational model also represent relationships separately, but efficiency and restrictions to atomic-valued attributes have been problems for software object management.

We propose a model of programmable relations over composite objects. This general model allows relationships among objects to be represented explicitly, and it enables derivation and other processes to be programmed into the implementation of relations. Programmable implementations, combined with an operationally-efficient interface, should help to overcome the performance problems of conventional relational systems.

Based on this model we have developed the language APPL/A, an extension of Ada that includes active, programmable relations plus mechanisms for forward and backward inferencing over relations. We introduce APPL/A and present an example system of relations to illustrate relation specifications and the opportunities for alternative relation implementations.

1 Introduction

Software environments must help people use tools to support the processes required to develop and maintain software products. We believe that this support must take the form of aids for building and aggregating software objects, and also aids for maintaining the integrity and consistency of these objects in the face of change.

Recently many interesting software environments have integrated tools around persistent object stores within which software products develop as growing aggregates of smaller objects. This work has focused attention on the types of software objects out of which software products are built, the structures into which the objects should be aggregated, and the tools and processes with which to build and aggregate the objects.

In defining the structures into which software objects should be aggregated, this work has addressed important questions about how software objects should be related to each other. These structures define a *static* view of software products, indicating how a product's constituent parts should be juxtaposed and organized at the conclusion of a successful development activity. Much less understood is the difficult and equally important problem of controlling the emerging product's constituent parts during the *dynamic* process of development, when many kinds of changes can be expected to occur. Effective change management is especially dependent on the relationships between objects.

During development, for example, new software objects are constantly being created and existing objects modified. The creation of new objects may trigger the derivation of additional objects (e.g. compilation of object code from a new source module). The modification of existing objects may affect the correctness and consistency of dependent objects that are derived from, or defined in terms of, the changed object. Changes to dependent objects may in turn affect other objects.

The propagation of changes requires first that the relevant dependency relationships be recognized and traced to identify the direction and extent of changes. It also requires that the semantics of these relationships be understood so that the proper kinds of changes can be determined. Finally, the changes must be effectively and rapidly implemented in accordance with the structure and semantics of the relationships.

Change propagation is complicated by the fact that not all significant relationships can be known and planned for in advance. For example, during team development, different components of a software product may be built by different people. Sometimes only after these components have been completed does it become clear that they must bear some strict relationships to each other (e.g. they must use consistent calling sequences). Thus it becomes necessary to establish unplanned relationships between software objects during the development process.

To summarize, managing the process of change requires an object manager with the capability to specify dependency relationships and then to be able to rapidly apply them, determine where inconsistencies have arisen as a result of these relationships, and then control the process of changing the software product so that consistency is restored.

We believe that a formalism for dynamically defining very diverse sorts of relationships among the classes and entities in a software environment, and for defining flexible responses to the detection of inconsistencies with respect to those relationships, is an indispensible mechanism upon which to base an environment's support for change. In the first part of this paper we briefly survey previous formalisms that have been proposed for software managers. We argue for the use of relations over objects as the appropriate formalism and for programmability as a major feature for overcoming the limitations of previous attempts to use relations. In the second part of the paper we describe APPL/A, which is an extenstion of of Ada to include a relation construct. APPL/A is our initial prototyping vehicle for exploring the utility of relations for representing software products and managing their changes. In the third part of the paper we show an example in which APPL/A is applied to some typical kinds of changes and their effects.

2 Non-Relational Approaches to Software Object Managment

A number of non-relational data-management systems are available or proposed for software development environments. These offer a variety of data models and other features for data management, including consistency maintenance and change management. Our belief, though, is that none of these provides quite the right combination of model and features.

These systems fall into three broad groups:

- Specialized systems with *ad hoc* models.
- General object-oriented systems which represent relationships between objects solely as attributes or properties of objects.
- General systems with data models that represent relationships between objects with constructs that are separate from objects.

Each of these is discussed below.

2.1 *Ad Hoc* Models

Considerable research in software engineering has been directed toward configuration management and version control systems [45,18]. These systems may be integrated parts of a larger environment, such as Apollo DSEE [24,25] or Cedar [42,23], or they may comprise independent tools, such as the version control systems RCS [43] and SCCS [36]. In between these extremes are systems like Make [16], Odin [11], and Adele [15], which support configuration management and version control by integration and coordination of independent tools.

Configuration management systems typically support the automatic "building" of software systems based on formal "system models" (which may integrate version control). These systems support derived data of certain types, and they are able to represent derivation and "component-of" relationships among the objects and/or types involved, typically in the form of a graph. Version control systems typically represent successive and/or alternative versions of objects of various types in terms of trees (or occasionally DAGs). Heimbigner and Krane [18] describe a general graph-transformation model for configuration management that subsumes many specific models.

While these systems can represent important kinds of relationships among important kinds of objects, their models of objects and relationships may both be lacking. The objects are usually files of limited types; they are static, large-grained objects with few attributes (although they may have versions). The relationships are also usually of limited types and while they may represent derivation processes they may nevertheless be relatively static.

There are a few exceptions to these generalizations. Odin (but not Make, Adele, or DSEE) supports multi-step inferencing in the derivation of objects. Odin, also unlike these other systems, automatically rederives derived objects that are out-of-date. Adele, however, provides a relatively powerful mechanism for assembling components dynamically, and allows a user-definable set of attributes to be associated with these components. DSEE provides active elements separate from its configuration and version facilities, in the form of "monitors".

Overall, while their capabilities vary, the systems discussed here are good at representing certain important kinds of objects and relationships. However, their models are too specialized and while they provide structure for the coordination of some processes they lack the degree of activity that is necessary for consistency maintenance and change propagation in general.

2.2 Models without Relationship Constructs

Systems that lack separate constructs for of inter-object relationships tend to be strictly object-oriented in the structural sense. Objects are defined in terms of slots (and possibly also methods), and new types or classes of objects may be defined by inheritance from existing types or classes (of course, the terminology and concepts vary somewhat from system to system). Examples of such systems include VBase [4], EXODUS [7], and Trellis/Owl [38,30].

One characteristic shared by these is that inter-object relationships are represented solely as attributes or properties of objects. They provide no "free-standing" relationship construct that can be referenced independently of the objects related. This leads to two general problems. First, the relationship must be associated with one object when it may apply equally well to two or more. Second, the relationship is reduced to a "second-class citizen"; it cannot be manipulated independently or assigned attributes of its own. This can be a problem in managing change since many relationships are naturally symmetric (e.g. between a specification and its implementation).

For example, consider the "compiles" relationship between types source-code and object-code, which represents the derivation of an instance of the latter from an instance of the former. This relationship must be defined in terms of both types, but in a strict structurally object-oriented system it must be associated with one or the other. Additionally, the relationship may have attributes of its own, such as compile-time, compiler-flags, or error-messages, but there is no relationship object to which these can be attached (and it is dissatisfying to have to attach these to either one of the objects involved).

An alternative in the object-oriented model is to create two relationships, "compiles-into", from source-code to object-code, and its inverse "compiled-from", from object-code to source-code (and indeed VBase supports the automatic generation of inverse relationships). This solves the problem of "asymmetrical" representation, but a single process, the compilation, is now represented in two places, and the consistency of these representations is now an issue. Moreover, it is still necessary to attach each relationship to an object, and there is no way to refer to a relationship independently or attach attributes to it.

While structurally object-oriented data models tend to be weak with respect to the representation of relationships, those systems that also behaviorally object-oriented tend to be strong with respect to the implementation of relationships. The behavior of objects arises from the methods or operations that can be applied to them, and behaviorally object-oriented systems necessarily allow users to program methods. Some of these systems, such as VBase and Trellis/Owl, also allow users to program the implementation of attributes or properties of objects. In such systems attributes may take on computed as well as stored values. Thus programmable attributes can encapsulate derivation processes, which can be used to automate the creation and maintenance of relationships between objects. Additionally, they may also allow a high-level data management system to be abstracted from the underlying persistent storage system; that, in turn, may facilitate the evolution of the storage system and enable access to "foreign" data.

2.3 Models with Relationship Constructs

A variety of non-relational data management systems offer independent constructs for representation of relationships between objects. DAMOKLES [14] has an extended entity-relationship model which includes objects, relationships, and versions as its prime constructs. Objects may be composite, and may include subobjects, versions, and relationships, but relationships may also be "free-standing". PCTE [17] and CAIS [1] also take an entity-relationship approach. Encore [46] has an object-oriented type hierarchy, but it includes independent constructs for "properties" (which can relate objects) and operations (which correspond to methods). Cactis [22] has a semantic model that includes types for both objects and relationships.

In each of these systems the decision was made to provide a separate relationship construct to enable relationships to be manipulated independently and/or to allow attributes to be associated with them. Thus these models directly attempt to overcome the limitations of a strict object-oriented model.

On the other hand, in these systems the relationships are extensional: the objects to be related must be specified (directly or indirectly) by the user. Unlike the programmable attributes of VBase and Trellis/Owl, the relationships do not encapsulate computations. Thus the maintenance of dependency relationships must be performed by every process that modifies those relationships. This is not to say that the systems cited here cannot support derived data. For example, in Cactis [22] an object can have derived attributes, those computations can depend on data that are propagated over relationships, and derived data are automatically rederived as needed when they are out-of-date. However, the derived attributes do not represent relationships (even though they may depend on them), the relationships must be established manually, and the derivation process is associated with the object rather than the relationship.

In our opinion the most appropriate way to treat relationships among objects for purposes of software development is to combine the strengths of the object-oriented and entity-relationship approaches to provide a model in which relationships are "first-class" citizens. This implies that software object management should be based on a model of programmable relationships over objects, in which relationships are represented explicitly and in which they can encapsulate processes to automate derivations and propagate changes.

3 Relational Approaches to Software Object Management

There have been several attempts to support object-management for software development, VLSI design, and CAD using systems based on the relational data model [27,35,34,8,10, 26,29,20,44,33]. Many advantages have been cited for this approach, such as support of teamwork [8], tool integration [8,10], support for multiple views of data [27,35], and data independence [29,20,10], among others. We believe that a relational data model is also appropriate because it provides for the independent representation of inter-object relationships (in the form of tuples) and it allows those representations to be aggregated (into relations).

However, systems based on the standard relational data model of Codd [12] have been criticized on two important grounds. First, the data model has been criticized because relations are "flat": relations apply only to simple, atomic objects. Second, the performance of commercial relational databases has been found to be inadequate when used for software development and other design activities.

3.1 The Problem of "Flat" Relations

An obvious response to the first criticism is to allow the attributes of relations to include non-atomic types. One alternative is that taken in ALGRES [9], which provides several type constructors (sets, multi-sets, lists, and records) which may be nested in any combination and to any degree. This enables the construction of relations over relations, for example. Another alternative is that taken in POSTGRES [37], in which attributes of relations can include abstract data types, although not other relations. POSTGRES [39] also includes other features that are important for change management, such as automatically-derived data and active elements ("alerters" and "triggers").

Strictly speaking, such approaches move away from the conventional relational data model. However, they retain several of its important features. These include independent representation of relationships between objects, the ability to aggregate these relationships, and the ability to query these aggregations.

3.2 The Problem of Performance

The performance of relational systems is a known problem. For example, Linton [27] reports performance tests on the OMEGA system, which is implemented using the INGRES relational database [40]. In this system the retrieval and display of a simple five-line program took 40 seconds (elapsed time) in response to an interpreted, non-buffered query and 7 seconds (elapsed time) in response to a compiled, buffered query. Performance with a longer program was notably worse. Navathe [29] cites studies that showed that commercially-available DBMS systems performed one and one-half to five times slower than specialized design systems for comparable tasks.

Consequently, relational systems that have been developed specifically for software and design data typically include features to enhance performance [8,44,10,20]. While these techniques are successful to a degree, the implementations of these systems are nevertheless specialized and fixed. Horwitz [21] has argued that individual relations should have programmable implementations. This would allow access to non-relational data structures that may have improved performance or other benefits. Moreover, she has argued that an operationally-efficient relational interface, combined with programmable implementations, should overcome many of the performance problems typically associated with relational systems. She has shown that query-evaluation based on the three access functions *membership-test*, *selective-retrieval*, and *relation-instantiation* may be more efficient than traditional evaluation methods for set operations on conventional relational databases. A similar notion is applied in the "multi-level data structures" of [28], in which the performance of a given abstract storage structure (such as a search table) is enhanced by an implementation in terms of several other storage structures (such as binary search trees and sorted arrays) among which data are moved by background processes. AP5 [13], an extension of Lisp to include a combination of relations and predicate logic, also provides programmable implementations for relations (albeit relations stored in virtual memory).

Programmability of relation implementations appears to offer a general mechanism for overcoming the performance limitations of conventional relational systems. It should be noted that programmability of this type is missing from both POSTGRES and ALGRES. Nevertheless, programmable implementations are not mutually incompatible with the kinds of features found in those systems. Indeed, our approach is based on the combination of an advanced relational data model with programmable implementations.

4 Active, Programmable Relations as an Organizing Principle

The principal tenet of this paper is that active, programmable relations over complex objects is a fundamental concept for software object management. More particularly, we believe that such relations are especially appropriate for tracking and managing the changes to software products.

To summarize the conclusions of Sections 2 and 3, a data model for software development that provides adequate support for change management should

- Be general and extendible
- Represent both objects and relationships explicitly and independently
- Allow relationships between complex objects
- Allow relationships to represent and encapsulate processes
- Provide for the aggregation of relationships
- Allow programmable implementations
- Support other forms of activity such as triggers and demons

We are exploring the utility of programmable relations through the vehicle of a programming language. We feel that this language must be capable of representing typed objects, relations and their operations. Abstractly, relations are a subset of the cross-product of a list of object types. Each element in the relation is a tuple of attribute values. The attribute values can include composite objects and can be derived and constrained. The operations available for relations include of insert, delete, update, and selective-retrieve (corresponding approximately to the operations of [21]). More powerful operations can be programmed from these. Additionally, some operations can be omitted, for example to define "views" which are "read only".

The implementation of relations is assumed to be programmable. Relations are implemented in terms of processes, where processes correspond to programs in execution. By providing appropriate implementations for relations, a number of capabilities can be provided:

- Relations and the values they contain can be made persistent. The choice of mechanism for persistence if determined by the implementation. For example, one might use a conventional file system for storage, or a commercial relational database.
- Relations may be programmed to derive attribute values, or check and maintain constraints. Derivations can be performed using both conventional tools, such as compilers and analyzers, and also software process programs [31]. Constraints may be specified and programmed in terms of some constraint language, or by invocation of various monolithic analysis tools.
- Relations can be proactive, i.e. operating independently, or reactive, i.e. responding directly to operations on the relation and indirectly to operations on other relations.
- Computation strategy for derived attributes (e.g. "eager" vs. "lazy") and storage strategy for derived attributes (caching vs. recomputation), may be controlled through appropriate implementations.

In order to explore the capabilities of programmable relations, we have chosen initially to extend Ada [2] with features for supporting relations. This extension is referred to as APPL/A [41]. Our research efforts have focussed on defining the structure and semantics of APPL/A and then applying the language to various software object management problems such as change maintenance, requirements specifications and design specifications. In the rest of this paper, we describe some of the details of APPL/A and show by example how it may be used to manage some typical kinds of changes in a software product.

5 APPL/A

APPL/A [41] is an extension to Ada [2] that includes features related to active, programmable relations as described in Section 4. APPL/A supplements Ada with a relation unit, a tuple type, and related control constructs including an indirect reactive mechanism, the **upon** statement. Relations provide persistent storage of Ada objects. APPL/A is intended to be used as a prototype for experimentation in developing systems based on these features.

The following subsections outline the principle features that APPL/A adds to Ada. An example of a system of APPL/A relations appears in the next section.

5.1 Relation Units

APPL/A defines a *relation* library unit. A relation unit has a specification and a body. The specification defines the specific logical properties of an abstract data type that is generally a multiset of **tuples**. The body provides a *programmed* implementation for those properties in accordance with the definition of the language. Like tasks, relations represent parallel threads of execution in a program; thus relations are *active*. Also like tasks, relations can be used to define both types and instances. All relation instances must provide for the *persistent* storage of tuples up to the lifetime of the relation type definition.

5.2 Relation Specifications

A relation specification includes

- An external storage specification.
- A tuple type declaration, which defines the type of tuple abstractly stored in the relation.
- Relation entry declarations, which represent a restricted set of operations on the relation.
- Computational dependencies, which define the ways in which attribute values are computed.
- Constraint specifications, which restrict the tuples stored in the relation.

Each of these is discussed in a following subsection.

5.3 Relation Bodies

The general syntactic and semantic properties of relation bodies are defined by APPL/A, but the particular implementation of those properties is not defined. Thus APPL/A relations have *programmable* implementations.

Syntactically, relation bodies are similar to task bodies. To support the semantics of relations the body must

- Implement the operations on the relation, as indicated by specified entries, in accordance with their defined semantics.
- Implement the computation of attribute values as indicated by specified computational dependencies.
- Enforce specified constraints on attribute values and tuples.

- Provide persistent abstract storage for tuples up to the lifetime of the type definition for the relation.

However, the body of the relation is not constrained with respect to storage systems, computation and caching strategies, inferencing for constraint maintenance or other purposes, or other activities in general.

We believe that the semantic restrictions on relation implementations support an especially useful abstraction. The lack of other constraints on implementations facilitates prototyping, evolution, adaptation, and extension of systems.

5.4 Tuples

APPL/A **tuple** types are syntactically similar to Ada record types. The components of a tuple type are known as **attributes**; each attribute has a **mode**, which is one of **in**, **in out**, or **out**. Tuples are abstractly stored in and are retrievable from relations. Values for attributes of mode **in** must be given by the user; values for attributes of mode **out** are computed automatically in the relation; values for attributes of mode **in out** may be given by the user but then may be replaced by values computed in the relation. Tuples retrieved from a relation receive copies of values abstractly stored in the relation; to help maintain consistency between those tuples and the relation values may not be assigned individually to attributes outside of the body of a relation. Otherwise whole tuples may be assigned and compared for equality.

5.5 Relation Entries

Operations on relations are effected through entries that are syntactically similar to task entries. Relation operations are restricted to **find** and **selected** plus any subset of **insert**, **update**, and **delete**.

The **insert** entry takes values for those attributes of mode **in** and **in out** and effects the logical storage of a tuple with those values. The **update** entry effects the logical update-in-place of attributes of mode **in** and **in out** of selected tuples in the relation. The **delete** entry effects the logical deletion of selected tuples in the relation.

The **find** and **selected** entries are used to selectively retrieve tuples from a relation and to trigger operations by that retrieval. These entries are not directly callable from user programs; instead they are used to implement iteration over relations, which is described below. When a tuple is retrieved it includes not only the attribute values given by a user but also any attribute values computed automatically in the relation.

5.6 Computational Dependencies and Constraints

Attributes of APPL/A relations can take on computed values. These attributes must have mode **out** or **in out**. The way in which an attribute value is to be computed can be indicated by a *computational dependency specification.* These specifications stipulate that a given attribute (or list of attributes) is to be computed using a given subprogram or entry with given values as input; the inputs can include the values of other attributes in the tuple or other identifiable values. If a dependency specification is given for an attribute then that attribute can only take on values in accordance with that specification. Computed attributes provide for *derived* objects; in conjunction with computational dependency specifications they enable the representation of *derivation relationships* and allow for automation of the derivation process.

Constraints are predicates that characterize the state of a relation. They are expressed in terms of *relational predicates*, which include conditional and quantified forms. They can apply both to attributes and tuples, and they can be expressed in terms of other attributes and tuples in the same or other relations. The constraints must be true of all tuples that are retrievable from a relation; APPL/A constraints are conventional in this way. Planned extensions to the language include constraint-like expressions that are to be regarded as *goals* to be satisfied eventually rather than as immediate restrictions.

5.7 Iteration over Relations

Tuples are retrieved from relations using an iterative construct of the form

```
for t in R where P loop
    S;
end loop;
```

where R is a relation, t is a loop variable of the tuple type for R, P is a predicate used to select values from R for t, and S is a statement that may operate on t. Each tuple in R that satisfies P is assigned to t in turn. This iteration is implemented in terms of repeated calls to the **find** and **selected** entries for R. Calls to the **find** entry effect the retrieval of some subset of the tuples in R; upon the conclusion of iteration (if it is not interrupted) the retrieved tuples must comprise a superset of the tuples in R that satisfy P. For each retrieved tuple that actually satisfies P the **selected** entry is called with that tuple as a parameter and S is executed. The **selected** entry can thus be used by the relation to invoke processes in response to the selective retrieval of a tuple.

5.8 "Upon" Statements

The **upon** statement provides a reactive mechanism that allows a process to respond independently and asynchronously to operations on relations. The **upon** statement has the general form

```
        upon E invoke
            S;
        end upon;
```

where S is a statement and E is an "invocation event." Invocation events are entry calls into relations. Thus, an **insert** call into relation R generates an event "R.insert(...)."

Each relation, task, subprogram, or package that includes an **upon** statement has associated with it a conceptual *event queue*. This event queue stores records of information about relation entry calls that are referenced as invocation events in the **upon** statements of that program unit. When an invocation event occurs a record for it is enqueued in the event queue of each unit that has cited that invocation condition. These records are queued in the order in which the invocation events occur; the records contain the identity of the corresponding relation entry and the values of the actual parameters with which it was called.

When control in a process reaches an **upon** statement it may proceed only if the event queue has at its head a record for the corresponding invocation event. otherwise control is suspended at that point, until the invocation event occurs, if ever. If and when the above condition is satisfied, the record is dequeued from the event queue, the appropriate values are assigned to the formal parameters of the invocation event, and execution of the **upon** statement proceeds.

Upon statements can be used wherever **accept** statements can be used, in particular in **select** statements, so it is possible to execute **upon** statements selectively and conditionally.

6 An Example System of Relations

This section presents an example of a small system of interconnected relations. It describes their specifications and implementations. These have been chosen to illustrate various features of APPL/A, including a range of possible implementation strategies, and to show how these features facilitate changes of different kinds in a software environment.

The example begins with two relations, C_to_Object and Object_to_Executable, that represent a simple "Make"-like [16] system for maintaining relationships between C source code and the corresponding object and executable code. (For simplicity, it is assumed that each C source module represents a complete program; it is straightforward to define relations that enable programs to be constructed from multiple modules.) The example continues with the addition of two more relations, C_DFA and Bug_Reports. Reference is made to various tools and abstract types; for simplicity, these are assumed to be defined elsewhere.

The implementations of these relations are described generally in terms of policies governing the derivation and storage of data and mechanisms for the maintenance of consistency between relations. The implementations are not described with respect specific storage systems. However, note that the storage systems for these relations are programmable, i.e. that the choice of a storage system is up to the programmer. All of the relations could share a common storage system, or each could have its own storage system. These can be programmed independently; consequently, the storage system for any one relation can be changed without affecting the others. In this way the programmability of storage implementations provides an additional mechanism for system evolution and prototyping.

6.1 Relation C_to_Object

Relation C_to_Object (Figure 1) represents the derivation relationship between C source code and the object code that is compiled from it. The tuple type for this relation has four attributes: "name", the name of the source-code module, "c", the source code itself, "obj", the object code, and "errs", error messages produced during the compilation. "Name" and "c" have mode **in** and are given by the user. "Obj" and "errs" have mode **out** and are be computed automatically; this computation is indicated in the "dependencies" part of the relation specification. The relation thus encapsulates the compilation process.

The relation has entries for insertion, deletion, updating, and retrieval (the "find" and "selected" entries). Constraints on the relation require that "name" and "c" have non-empty (i.e. non-null) values and that (as a result of the compilation process) either "obj" or "errs" must have a non-empty value. Constraints also require that tuples have unique "name" and "c" values.

There are several generic implementation strategies for this relation (i.e. strategies that are independent of the particular storage system used). For this example we propose the following implementation of the entries:

- **Insert**: Upon insertion the constraints on given "name" and "c" values are checked. If these are violated the exception "constraint_error" is raised. If not, the compiler is invoked to derive the object code and error messages, and the results are stored persistently. This represents an *eager* strategy for evaluation of computed attributes plus *caching* of the computed values. If the compilation is successful then the "insert" entry for relation Object_to_Executable is invoked with the resulting object code to propagate that code (see Section 6.2). This effects the automatic *forward propagation* of the object code.

- **Delete**: The selected tuples are deleted. If the object code in any of these would have been propagated to Object_to_Executable then the "delete" entry for that relation is invoked to delete the corresponding tuples. Thus deletion is automatically propagated. In this way no object code and executable modules are left "dangling" in Object_to_Executable without a corresponding source module in C_to_Object.

```
relation C_to_Object is
-- Relates C code to object code compiled from it
--
    type c_to_object_tuple is tuple
        name:  in name_type := empty;
        c:  in c_code := empty;
        obj:  out object_code := empty;
        errs:  out compile_error_msgs := empty;
    end tuple;
entries
    entry insert(
        name:  in name_type := empty;
        c:  in c_code := empty);
    entry delete( ... );
    entry update( ... );
    entry find( ... );
    entry selected( ... );
dependencies
    t.c determines t.obj, t.errs,
        by compile_c(t.c, t.obj, t.errs);
constraints
    no t in C_to_Object satisfies
        t.c = empty or t.name = empty;
    or
        (t.obj = empty and t.errs = empty);
    end no;
    --
    all t1 in C_to_Object satisfy
        no t2 in C_to_Object satisfies
            t1 /= t2 and (t1.c = t2.c or t1.name = t2.name);
        end no;
    end all;
End C_to_Object;
```

Figure 1: Sketch of Specification for Relation C_to_Object

- **Update**: Only "name" and "c" values may be updated directly (since these are the only attributes with mode **in**). If "name" is updated then the stored value is simply changed. If "c" is updated then "obj" and "errs" are recomputed and stored, and if the compilation is successful the object code is propagated to Object_to_Executable (as with "insert"). In this way derived data are updated automatically, and changes are propagated between relations.

The "find" entry simply returns the first or next selected tuple, as requested. The "selected" entry is empty (but it is nevertheless included because it is invoked automatically during execution to trigger the signal of a retrieval event, which may be of interest to other relations).

6.2 Relation Object_to_Executable

This relation represents the derivation relationship between object code and executable code (Figure 2). Its specification is similar to that of C_to_Object (Figure 1). It has three attributes: "obj", the object code, "exe", the derived executable code, and "errs", the error messages produced by the loader. In this case "obj" has mode **in** and must be inserted directly, while "exe" and "errs" have mode **out** and are derived automatically by applying the loader to the object code (as specified under "dependencies"). The relation thus encapsulates the loading process.

The relation has "insert" and "delete" entries, plus entries "find" and "selected", but not "update". Constraints on the relation require that "obj" be non-empty and that either "exe" or "errs" be non-empty (as a result of loading). Also each tuple is required to be unique, and no tuple can have object code that is not also present in relation C_to_Object (this implies that the source code for each executable module must be retained).

Despite the abstract similarity of this relation to C_to_Object, the implementation can be very different. To illustrate this, we can design the implementation of Object_to_Executable to act as follows:

- **Insert**: When object code is inserted simply store it persistently. *Do not* compute the executable code and error messages at this time.

- **Delete**: Delete the selected tuples. If the selection of tuples depends on the executable code or the error messages and these have not yet been computed then invoke the loader. Cache any computed values that are not deleted.

- **Find**: Retreive the first (or next) selected tuple, as requested. Invoke the loader to derive the executable code and error messages if they are not already available. Cache any newly computed values.

This implementation illustrates a *lazy* evaluation strategy for the derived attributes, with *caching* of computed values for future reference. (An alternative implementation would be to simply rederive them upon every request and not store them at all.)

```
relation Object_to_Executable is
-- Relates object code to executable code derived from it
--
    type object_to_executable_tuple is tuple
        obj:  in object_code := empty;
        exe:  out executable_code := empty;
        errs:  out load_error_msgs := empty;
    end tuple;
entries
    entry insert(obj:  in object_code := empty);
    entry delete( ... );
    entry find( ... );
    entry selected( ... );
dependencies
    t.obj determines t.exe, t.errs,
        by load(t.obj, t.exe, t.errs);
constraints
    no t in Object_to_Executable satisfies
        t.obj = empty;
    or
        (t.exe = empty and t.errs = empty);
    end no;
    --
    all t1 in Object_to_Executable satisfy
        no t2 in Object_to_Executable satisfies
            t1 /= t2 and t1.obj = t2.obj;
        end no;
    and
        some t in C_to_Object satisfies
            t1.obj = t.obj;
        end some;
    end all;
End Object_to_Executable;
```

Figure 2: Sketch of Specification for Relation Object_to_Executable

6.3 Relation C_DFA

Suppose that subsequent to the programming of relations C_to_Object and Object_to_Executable a data-flow analysis tool for C programs becomes available. We can encapsulate this analysis in another relation, C_DFA, that represents the derivation relationship between C code and the data-flow analysis, and we can provide automated connections between C_DFA and C_to_Object without reprogramming that relation.

The tuple type for C_DFA will have "name" and "c" as given attributes and "dfa" as an attribute derived from the corresponding "c" value. The "c" values will be constrained to be non-empty and to be a superset of those in C_to_Object (to ensure that data flow analyses exist for those modules).

This relation, like C_to_Object, can be implemented using an "eager" evaluation strategy with caching of the computed results. However, in addition to the relation entries, the body of this relation will include **upon** statements that are activated in response to operations on C_to_Object. In particular, in response to C_to_Object.insert(c, name), the new C code is "captured" by C_DFA, analyzed, and stored. A similar response occurs to C_to_Object.update. (An analogous response could be programmed in response to C_to_Object.delete, but there is no requirement that C_DFA include only that C code in C_to_Object, and it may be useful to preserve flow analyses for source code that is not compiled.)

The use of **upon** statements enables the behavior of C_DFA to be linked to that of C_to_Object without affecting C_to_Object or requiring that it be reprogrammed. In this example the **upon** statement is used to maintain the state of one relation relative to another. Other uses are possible, including, for example, the collection of software metrics and monitoring of system use. This illustrates that the **upon** statement generally facilitates the extension of interconnected systems of relations and provides an indirect *reactive* mechanism by which their behavior can be automated and observed.

6.4 Relation Bug_Reports

Suppose now that a decision is made to collect information on problems or errors in C programs to facilitate their maintenance. This information can be represented in a relation "Bug_Reports". Abstractly, for each source code module named in C_to_Object, Bug_Reports collects the compiler error messages from C_to_Object, the loader error messages, if any, from Object_to_Executable, and the data flow analyses from C_DFA.

This relation can be implemented as a *view* of relations C_to_Object, Object_to_Executable, and C_DFA. In this case no information will be stored for Bug_Reports; whenever a user requests information for a unit, that information will be retrieved from the underlying relations. The tuple type for Bug_Reports will look like

```
type bug_reports_tuple is tuple
    name: out name_type;
```

```
        compile_errs: out compile_error_msgs;
        load_errs: out load_error_msgs;
        dfa: out c_data_flow_analysis;
    end tuple;
```

Note that *all* of the attributes have mode **out**, since all are computed within the relation. Additionally, Bug_Reports will lack entries for insert, delete, and update, since these operations are not applicable to the view.

The design of Bug_Reports as a view enables it to be added to the previous relations without recoding or otherwise affecting those relations. Moreover, since no data are stored for Bug_Reports, there is no problem in maintaining consistency with those underlying relations. The use of views is thus another approach by which systems of interconnected relations can be effectively extended and maintained.

7 Status and Experience

A formal syntax and semantics have been defined for APPL/A, as have rules for the translation of APPL/A constructs into standard Ada. These have been evolving gradually as we gain experience with the language. APPL/A has been used to program REBUS, a system which supports the specification of software requirements in a functional hierarchy. REBUS maintains data about requirements in APPL/A relations; the relation specifications and bodies comprise about 2700 lines of code, exclusive of runtime support systems and storage system interfaces. APPL/A is being used to extend REBUS to include features based on RSL/REVS [3,6] and to construct a design support systems based on the Rational Design Methodology of Parnas [32] and the IEEE design standard [5].

The use of APPL/A in REBUS has helped us to refine and reenforce both APPL/A and the principles on which it is based [19]. Ongoing research will enable us to continue to evaluate APPL/A and the model behind it. As principles evolve and the APPL/A mechanisms are updated we expect to gain additional insights into the requirements for managing change during software development.

8 Acknowledgements

This research was supported by the Defense Advanced Research Projects Agency, through DARPA Order #6100, Program Code 7E20, which was funded through grant #CCR-8705162 from the National Science Foundation. The authors wish to thank Deborah Baker, Roger King, Shehab Gamalel-din, and Mark Maybee for their advice. The comments of the members of the Arcadia consortium were also important in clarifying the issues surrounding

APPL/A.

References

[1] DoD requirements and design criteria for the common APSE interface set (CAIS). October 1986. Prepared by the KAPSE Interface Team (KIT) and the KIT-Industry-Academia (KITIA) for the Ada Joint Program Office (AJPO).

[2] *Reference Manual for the Ada Programming Language.* United States Department of Defense, 1983. ANSI/MIL–STD–1815A–1983.

[3] Mack W. Alford. A requirements engineering methodology for real-time processing requirements. *IEEE Trans. on Software Engineering*, SE-3(1):60 – 69, January 1977.

[4] Timothy Andrews and Craig Harris. Combining language and database advances in an object-oriented development environment. In *OOPSLA*, 1987.

[5] Jack H. Barnard, Robert F. Metz, and Arthur L. Price. A recommended practice for describing software designs: ieee standards project 1016. *IEEE Trans. on Software Engineering*, SE-12(2):258 – 263, February 1986.

[6] E. Bell, Thomas, David C. Bixler, and Margaret E. Dyer. A requirements engineering methodology for real-time processing requirements. *IEEE Trans. on Software Engineering*, SE-3(1):49 – 59, January 1977.

[7] Michael J. Carey, David J. DeWitt, Daniel Frank, Goetz Graefe, Joel E. Richardson, Eugene J. Shekita, and M Muralikrishna. *The Architecture of the EXODUS Extensible DBMS: a Preliminary Report.* Technical Report Computer Sciences Technical Report #644, University of Wisconsin, Madison, Computer Sciences Department, May 1986.

[8] S. Ceri and S. Crespi-Reghizzi. Relational data bases in the design of program construction systems. *SIGPLAN Notices*, 18(11):34–44, November 1983.

[9] S. Ceri, S. Crespi-Reghizzi, L. Lavazza, and R. Zicari. *ALGRES: A System for the Specification and Prototyping of Complex Databases.* Technical Report 87-018, Dipartimento di Elettronica – Politecnico di Milano, P.zza Leonardo da Vinci 32, I-20133 Milano, Italy, 1987.

[10] Kung-chao Chu, John P. Fishburn, Peter Honeyman, and Y. Edmund Lien. Vdd – a VLSI design database system. In *Database Week: Engineering Applications*, pages 25–37, IEEE, 1983. IEEE catalog number CH1886-1/83/0000/0025.

[11] Geoffrey M. Clemm. *The Odin System: an Object Manager for Extensible Software Environments.* Technical Report CU-CS-314-86, Univ. of Colorado, Dept. of Computer Science, Boulder, Colorado, 1986.

[12] E. F. Codd. "A Relational Model for Large Shared Data Banks". *Comm. ACM*, 13(6):377–387, 1970.

[13] Don Cohen. *AP5 Manual.* Univ. of Southern California, Information Sciences Institute, March 1988.

[14] Klaus R. Dittrich, Willi Gotthard, and Peter C. Lockemann. DAMOKLES – a database system for software engineering environments. In *International Workshop on Advanced Programming Environments*, IFIP WG2.4, 1986.

[15] J. Estublier. "A Configuration Manager: The ADELE Data Base of Programs". In *Workshop on Software Engineering Environments for programming–in–the–large*, pages 140–147, Harwichport, Mass, June 1985.

[16] Stuart I. Feldman. Make – a program for maintaining computer programs. *Software – Practice and Experience*, 9:255 – 265, 1979.

[17] Ferdinando Gallo, Regis Minot, and Ian Thomas. The object management system of pcte as a software engineering database management system. In *Proc. Second ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 12 – 15, 1986.

[18] Dennis Heimbigner and Steven Krane. A graph transform model for configuration management environments. In *Proc. Third ACM SIGSOFT Symposium on Software Development Environments*, November 1988. to appear.

[19] Dennis Heimbigner, Leon J. Osterweil, and Stanley M. Sutton, Jr. *Active Relations for Specifying and Implementing Software Object Management.* Technical Report CU-CS-406-08, Univ. of Colorado, Dept. of Computer Science, Boulder, Colorado 80309, July 1988.

[20] Lee Hollaar, Brent Nelson, and Tony Carter. The structure and operation of a relational database system in a cell-oriented integrated circuit design system. In *21st Design Automation Conference*, pages 117–125, ACM/IEEE, 1984.

[21] Susan Horwitz. *Adding Relational Databases to Existing Software Systems.* Technical Report Computer Sciences Technical Report #674, University of Wisconsin, Madison, Computer Sciences Department, 1986.

[22] Scott E. Hudson and Roger King. The cactis project: database support for software environments. *IEEE Trans. on Software Engineering*, 14(6):709–719, June 1988.

[23] B. W. Lampson and E. E. Schmidt. Organizing software in a distributed environment. *SCM SIGPLAN Notices*, 18(6):1 – 13, 1983.

[24] David B. Leblang and R. P. Chase, Jr. Computer-aided software engineering in a distributed workstation environment. In *Proc. of the ACM SIGSOFT/SIGPLAN Symposium on Practical Software Environments*, pages 104 – 112, April 1984. Also ACM Software Engineering Notes, v. 9, n. 3, May 1984.

[25] David B. Leblang and Gordon D. McLean Jr. Configuration management for large-scale software development efforts. In *Workshop on Software Engineering Environments for Programming in the Large*, pages 122 – 127, 1985.

[26] Y. C. Lee and K. S. Fu. A CGS based DBMS for CAD/CAM and it's supporting query language. In *Database Week: Engineering Applications*, pages 123–130, IEEE, 1983. IEEE catalog number CH1886-1/83/0000/0123.

[27] Mark A. Linton. Implementing relational views of programs. *SIGPLAN Notices*, 19(5):132–140, May 1984. Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, April 23-25, 1984.

[28] Abha Moitra, S. Sitharama Iyengar, Farokh B. Bastani, and I. Ling Yen. Multilevel data structures: models and performance. *IEEE Trans. on Software Engineering*, 14(6):858 – 867, June 1988.

[29] Shamkant B. Navathe. Data base management of computer-aided-design data. In G. V. Reklaitis and J. J. Siirola, editors, *Data Base Implementation and Application*, pages 43–50, AIChE, 1983.

[30] Patrick O'Brien, Bruce Bullis, and Craig Schaffert. *Persistent and Shared Objects in Trellis/Owl.* Technical Report DEC–TR–440, Digital Equipment Corporation, Hudson, Massachusetts, July 1986. See also the International Workshop on Object–Oriented Database Systems, 23–26 September 1986, Asilomar Conference Center, Pacific Grove, California.

[31] Leon J. Osterweil. Software processes are software too. In *Proc. Ninth International Conference on Software Engineering*, 1987.

[32] David L. Parnas and Paul C. Clements. A rational design process: how and why to fake it. *IEEE Trans. on Software Engineering*, SE-12(2):251 – 257, February 1986.

[33] M. H. Penedo. Prototyping a project master database for software engineering environments. In *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 1–11, ACM, 1986.

[34] Michael A. Powell and Mark A. Linton. Database support for programming environments. In *Database Week: Engineering Applications*, pages 63–70, IEEE, 1983. IEEE catalog number CH1886-1/83/0000/0063.

[35] Michael L. Powell and Mark A. Linton. A database model of debugging (preliminary draft). 1983. ACM catalog number 0-89791-111-3/83/007/0067.

[36] Mark J. Rochkind. The source code control system. *IEEE Trans. on Software Engineering*, SE-1:364 – 370, December 1975.

[37] L. A. Rowe and Michael R. Stonebraker. "The POSTGRES Data Model". In *Proc. of the 13th VLDB Conference*, pages 83–96, 1987.

[38] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. "An Introduction to Trellis/Owl". In *OOPSLA '86 Conf. Proc.*, pages 9–16, 1986. Available as ACM SIGPLAN Notices 21, 11, November 1986.

[39] Michael Stonebraker and Lawrence A. Rowe. The design of postgres. In *Proc. of the ACM SIGMOD International Conf. on the Management of Data*, pages 340 – 355, 1986.

[40] Michael R. Stonebraker, P. Kreps, and Gerald D. Held. "The Design and Implementation of INGRES". *ACM TODS*, 1(3), September 1976.

[41] Stanley M. Sutton, Jr. *The APPL/A Programming Language – Definition and Status.* Arcadia Document CU–88–02, Univ. of Colorado, Dept. of Computer Science, Boulder, Colorado 80309, February 1988. Draft.

[42] W. Teitelman. A tour through cedar. *IEEE Software*, April 1984.

[43] W. Tichy. Design, implementation, and evaluation of a revision control system. In *Sixth International Conf. on Software Engineering*, 1982.

[44] Zara Robert V. and David R. Henke. Building a layered database for design automation. In *22nd Design Automation Conference*, pages 645–651, ACM/IEEE, 1985. Paper 40.1.

[45] Jurgen F. H. Winkler, editor. *International Workshop on Software Version and Configuration Control*, Tubner–Verlag, Grassau, West Germany, January 1988.

[46] Stanley B. Zdonik and Peter Wegner. A database approach to languages, libraries, and environments. In *Workshop on Software Engineering Environments for Programming in the Large*, pages 89 – 112, 1985.