Active Relations for Specifying and Implementing Software Object Management

Dennis Heimbigner
Leon J. Osterweil
Stanley Sutton Jr.

CU-CS-406-88 July 1988

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

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.

Active Relations for Specifying and Implementing Software Object Management

Dennis Heimbigner Leon J. Osterweil
Stanley Sutton Jr.

University of Colorado
Boulder, CO 80309-0430

July 5, 1988

Abstract

Object management in software environments can be enhanced by providing programmable object management systems. Key requirements for object managers include typing, persistence, constraints, event monitoring, sharing, activity, derived data, support for foreign tools, query, and programmable implementations. We define a model – Aspen – which uses the notion of programmable relations as a unifying mechanism for meeting many of these requirements. We describe a language – APPL/A – which is a particular realization of Aspen. APPL/A provides a vehicle for experimentation with programmable object management systems. APPL/A has been used to support the development of a requirements specification system. Our experiences from the use of APPL/A are being applied to adapt the language, refine the model, and in general enhance our understanding of the basic requirements for software object management.

1 Introduction

Over the past several years, software environments have come to be viewed less collections of large monolithic tools and more as sets of smaller tool fragments that are coordinated by development processes whose job is to create and manage structured collections of software objects stored in a central repository [32,23,9,13,5,7]. In recent years considerable attention has been focused on the nature of that object store and the objects which must populate it. These objects range from source text and executable code, to test cases, documentation, requirements, and designs.

A variety of disciplines and systems have been proposed as the basis for effective software object management. These range from simplistic attempts to use turnkey relational databases [24,33,8], to more sophisticated approaches that treat the software objects as instances of types which are organized into potentially elaborate and complex type hierarchies [42,20]. These efforts have met with varying degrees of success, and as a result there is currently a great deal of research interest in software object management.

We believe that currently there is insufficient understanding of the basic requirements for software object management and consequently a great deal of uncertainty about how to design and implement software object management systems. In evaluating earlier systems we were struck by the fact that most failed to place sufficient emphasis on the management of relationships among software objects. As relationship management strikes us as one of the key tasks of an object manager, and as other projects seem to be addressing other tasks, we have focused on relationship management and related issues such as constraint maintenance, derived data, inferencing, and implementation independence.

Moreover, we believe that no single approach to software object management is generally appropriate: the needs for object management vary from environment to environment, project to project, and user to user, and they evolve over time. We believe that the best way to accommodate the uncertainty and change in software object management systems is to design them as programmable components of environments. For this reason, we are investigating concepts and languages appropriate to this task.

Our approach is based on the iterative development of research prototypes, each of which is designed to incrementally elucidate requirements for software object management while also shedding light on design and implementation issues. In particular, having identified some key requirements, we have defined a model – Aspen – which formulates general principles for programmable software object management [17]. We

have also developed a language – APPL/A – as one implementation of this model [39]. APPL/A is intended as an vehicle for experimentation with programmable object management systems. Both Aspen and APPL/A emphasize the importance of relationships among objects and use the concept of "relation" to integrate other object-management capabilities. We have used APPL/A to support the development of user-level software tools. This has given us experience that we are applying to adapt the language, refine the model, and in general enhance our understanding of the basic requirements for software object management.

This paper describes our progress in iterating through the prototyping activity. First, we discuss the general characteristics which seem to describe the requirements for an object store. Second, we describe the concepts of Aspen, and, in particular, the role of relations. Third, we describe our experimental language, APPL/A, and our experiments with systems implemented using it. Finally, we present our conclusions and the directions which they indicate for subsequent iterations.

2 Object Manager Requirements

In order to initiate the cycle of experimentation, we needed to define an initial set of requirements for object management. In this section, we briefly review the most important of those requirements. This list is taken from published lists [5,10] combined with initial experiences in managing software products [25]. Some possible requirements have intentionally been ommitted from this list because we do not believe they are fundamental. Evidence from the configuration management community [41] suggests that versioning, for example, must be built over an object manager to support semantic specification of versions.

2.1 Data Model

By "data model" we mean the general way in which objects are represented and classified. We begin with the assumption that the software product consists of objects; the data model must accommodate these as well as other objects that will occur in an environment. Objects have identities, by which they can be referenced, a state, which represents its value and can be operationally accessed, and a type. A type is usually defined in terms of a set of operations and a set of values. The operations are applicable to all objects of the type, and the values can be taken on by any object of the type.

Typing is useful because it imposes some organization on objects, provides a way to summarize their properties, and serves as a basis for their interpretation and use. Because of these benefits it is often useful to strictly enforce the typing of objects. However, in a software environment there are circumstances in which it is expedient (if not necessary) to violate the typing of objects, for example, to treat an object of one type as if it were of another, to access the implementation of an object, or to modify the definition of a type. Consequently, mechanisms by which typing constraints can be circumvented are also important. In order to avoid a breakdown of the typing system, however, these mechanisms should be well integrated, explicit, and systematic, and their use should be controlled insofar as possible.

Typing systems typically provide a set of predefined "base" types plus type constructors that enable new types to be defined from existing types. In a software environment the base types and type constructors should be appropriate for software products and processes. Aggregations of objects and relationships among objects seem particularly important.

The set of types in an environment must be easily extensible. The types required for a software product vary from project to project and evolve over time within a project. The extensibility of a type system is facilitated by a rich set of base types and type constructors plus mechanisms that allow type definitions to be reused.

2.2 Persistence

The persistence of objects is a measure of their lifetime. In software environments the persistence of objects is usefully measured relative to the scopes in which they are used. A range of persistence of objects can be distinguished [4]. Of particular importance for software development is the persistence of objects beyond the lifetimes of processes that use them.

Conventional programming languages are concerned primarily with objects that persist within a single program execution. They commonly use external systems, such as files or databases, for storage of more persistent objects. However, these systems are limited in some ways, their use adds syntactic and semantic complexity to a language, and it weakens the programmer's control over stored objects.

So called "persistent programming languages" provide better integration of persistence with the language than is available through file systems [4,27,35,39,28]. Many of them enable persistent objects to be treated more like "transient" objects whose lifetimes are limited to a single program execution. They also provide *type safety*; that is, they preserve typing information for objects that persist between programs.

Otherwise, these languages differ widely in their approaches, and there is no consensus as to which of the approaches is best. In any case, programmable control over the persistence of objects is particularly important in a language for the programming of object management systems.

2.3 Constraints and Inconsistency

Constraints are conventionally regarded as predicates on data that must be satisfied whenever the data are generally accessible. Constraints in this sense are useful in many contexts in software environments. They may be applied to individual objects, aggregates of objects, and relationships among objects.

The conventional view of constraints is too restrictive for software processes, however. It assumes that constraints can be specified in advance of processing and that constraint evaluation and maintenance are not prohibitively costly. However, during software development not all constraints are known in advance, and constraint processing may be long and complex. Although there is a high potential for violation of constraints in a software environment it may not be practical or necessary to enforce all constraints at all times.

If constraints are not always enforced then an environment may be in an inconsistent state. While inconsistency is intolerable in some contexts it may be tolerable in others. If inconsistency can be accommodated then it may be easier to add constraints dynamically and the extent to which processes are blocked by constraint maintenance may be reduced. Thus software environments should facilitate the constructive management of inconsistency rather than simply prohibiting it.

2.4 Event Monitoring

The execution of many processes in a software environment will be conditional on events that are generated by other processes. An event can be defined as an arbitrary signal that is generated a some selected point in a process. Similar concepts include *demons*, *triggers*, and *event monitoring*. In conventional languages the conditional process must be invoked directly by the process that generates the event. This approach is useful in some cases but it is limited in that the invoking process must know about all of the processes interested in the events it generates.

In order to avoid this knowledge of other processes, a *reactive* mechanism can used. Such a mechanism enables processes to respond independently and unobtrusively to events in the environment. This approach is useful because the process that generates

an event does not need to know about the processes that respond to that event; thus it enhances the extendibility of environments. Reactive processes can be used for many purposes, including performance measurement, user-interface management, constraint maintenance, object derivation, and general monitoring of resource usage.

2.5 Cooperative Sharing

Except for small projects, the development of software is inherently a cooperative activity. Many people and processes are involved in the design, construction, and testing of software. Thus, any software environment must support sharing of data among multiple users.

There are several interrelated issues surrounding shared access to data. One problem is concurrency control: ensuring that two or more processes that simultaneously access an object do not interfere with one another. In particular, no process should find an object in an inconsistent state as a consequence of concurrent access by another process. The database concepts of transaction, atomicity, and logging are examples of solutions to concurrency control.

Another issue is control of visibility of shared objects. Objects that are shared must be visible to the sharing processes. This implies that the objects and processes must exist in some encompassing context in which the sharing occurs. A related issue is control of persistence (see Section 2.2). If objects are to be shared in sequence by processes then the objects must be persist between accesses in some scope within which the processes execute. The visibility and persistence of objects with respect to separate threads of control within a single process are typically governed by definition in a language. The visibility and persistence of objects with respect to multiple separate processes within a larger system, such as an operating system or database, are typically governed by the rules of that system and are beyond the control of programming languages. The visibility and persistence of shared objects are central concerns of object management systems for software environments.

2.6 Other Capabilities

There are several additional capabilities which we believe place important requirements on object management systems for software environments. These include:

- **Active Objects**: We believe that not all objects should be regarded as passive entities upon which processes act. Rather, some objects should encapsulate

independent processes as well as state. In this way much of the processing that relates to an object can be associated directly with the object instead of being dispersed through application and system code. The activity of objects can be used to dynamically and independently manitain constraints, compute values, manage storage, and invoke other processes in general.

- **Derived Objects:** Software development can be viewed as the derivation of a product from requirements, and derived objects of many kinds are important any software environment. One of the purposes of a software environment is to support the derivation of objects. This should include support for the specification and maintenance of derivation relationships and control over derivation processes and derived objects.

- **"Foreign" Tools and Objects:** Software environments in general are not isolated systems. An environment may benefit from tools and objects developed in and imported from other environments. A problem with the use of such tools and objects, though, is that they may be based on different abstractions and implementations. To accommodate foreign tools and objects most effectively an object management system must be able to incorporate and take advantage of alternative abstractions and implementations while at the same time isolating the overall environment from potentially disruptive differences.

- **Programmable Implementations:** Many software environments provide fixed implementations for various aspects of object management. These may include the storage system, computation and caching strategy for derived objects, and inferencing mechanisms. While these may suffice for particular circumstances, we believe that fixed implementations are inadequate in general. A language for the programming of object management systems should instead allow programmable implementations for storage and related capabilities. This would promote the adaptability, portability, and extensibility of object management systems, and it would also enable an implementation to take advantage of existing support services where those can be of use.

- **Query:** The ability to query the object manager is often neglected in existing environments. Yet from databases, we know that set-oriented queries are a major source of efficiency because they provide opportunities for optimization that navigational queries cannot provide. In addition, an ad-hoc query mechanism is useful to project managers because it allows them to browse the system and

efficiently examine the status of the project. Managers do not want to write new programs for every question they might ask. This would seriously impede their ability to manage even moderate scale projects.

The capabilities outlined in this section do not comprise an exhaustive list of requirements for object management in software environments. Nevertheless, these capabilities should provide an effective basis for addressing other requirements, for example, configuration management, version control, and administrative support.

3 Active Relations as an Organizing Principle

Aspen is not an actual language; rather it is a set of concepts that must be made concrete in some actual language. An analogy may be drawn from operating systems. The concept of a "monitor" [18] defines an approach to designing operating systems, but it must still be embodied in a language such as Concurrent Pascal [6] before it can be used. Similarly, Aspen provides a set of concepts for languages that are intended to be used to construct object managers for software environments. Section 4 shows these abstract concepts made concrete in the context of the programming language APPL/A.

The principal tenet of Aspen is that relations are a fundamental concept for software object management. We believe that this approach distinguishes Aspen. With few exceptions [34,19], relations have been deprecated because of efficiency problems and problems with the use of flat relations [24,33,8]. We believe that the Aspen approach will avoid these problems without sacrificing the benefits of a relational approach.

It is important to note what our relations are not; they are not the flat relations used in traditional relational databases. Our approach combines relations with composite objects to provide the capabilities missing from simple relations. We also differ from Entity-Relationship models such as PCTE [14] and CAIS [1] in that we associate information about the semantics and implementations of the objects and processes with the relations rather than the objects.

Our justification for using relations comes from the observation that much of the difficulty of dealing with software products derives from the magnitude and complexity of the relationships among their constituent objects. The variety of these relations is poorly understood and is a major source of difficulty in developing and maintaining software products. A successful software engineering environment must be capable of managing not only objects but also the complex interrelationships among them.

Moreover, we believe that management of relations is facilitated if the relations are made explicit and are themselves raised to the status of objects.

Aspen does not yet address all of the requirements of Section 2. The key requirements that we have chosen to focus on initially are activity, constraints, monitoring, queries, derived data, and programmable implementations. It addresses persistence through the use of relations as persistent stores. It addresses typing to the extent that relations are part of the type system. Otherwise it assumes the existence of a pre-existing type system.

3.1 The Role of Types and Objects in Aspen

Notions of *object* and *type* have been well developed in programming languages and databases [2,15,22,16]. Since the world to be modeled consists of software objects, approaches from object-oriented languages seem especially appropriate to this domain, and so we have adopted the following principles as part of Aspen.

Each software object is unique and distinguishable and is an instance of an object type. A type characterizes its instances by restricting the set of operations that reference those instances. The operations define the structure of the state of objects belonging to the type, and through these operations the state of an object can be examined and modified. The "typing" of objects is "strong" in that an object can only be manipulated in terms of the operations associated with the type to which it belongs. All types are organized into a comprehensive type system. This system comprises some number of predefined types plus mechanisms for the definition of new types from existing types.

3.2 Relations

The concept of relations used in this model is is based on the mathematical definition of a relation as a subset of the Cartesian product of one or more domains (i.e. object types). As such, it is more general than the notion of relation as defined in most relational database systems. Those systems typically limit the attributes to non-structured types such as integers and strings. Our relations have no such restriction; they may include complex objects and even other relations. Additionally, our relations differ from those of conventional relational systems in terms of operations, semantics, and the range of available implementations.

Each occurrence of a domain in the cross product is known as an *attribute* of the relation; these attributes are both ordered and named. The elements in a relation

are known as *n-tuples* (for a relation over n attributes) or simply *tuples*. Tuples are unordered within a relation. Each tuple comprises an ordered list of values, one for each attribute; each value belongs to the type of the corresponding attribute and is identified within the tuple by reference to that attribute. Our relations are multi-sets, so it is possible to have two tuples with identical sets of values for all attributes.

The operations inherently associated with relations include create and destroy (for relations) and insert, update, delete, and selective-retrieve (for tuples in relations). More powerful operations such as those of the relational algebra can be programmed in terms of the basic operations. However, it is also possible to restrict the basic operations associated with a relation to a proper subset of those listed above (and thereby also restrict the additional operations that can be programmed from them). A restricted set of operations can be used, for example, to define 'views' on which the ability to retrieve or update data is limited.

3.3 Active Relations

While we believe that a relational interface is appropriate for users who wish to access software objects, we also believe that the semantics and implementation of relations should be programmed by their creators and that few restrictions should be placed on the complexity of these programs. This allows relations to serve as abstract interfaces to existing data structures, to trigger the execution of software tools, to implement choices about how and whether various relations are to be stored, and to determine what kinds of evaluation strategies (lazy versus eager, cached versus uncached) should be pursued. Indeed, relations seem to provide a unifying mechanism capable of supporting all the requirements for software object management. We will use the term "active relation" for such programmable relations. The term "active" refers to the fact that we expect the programs associated with relations to act concurrently and that processes associated with the relations will contribute to this concurrency. We feel that concurrent activity is critical in addressing the requirements enumerated in Section 2.

Active relations may be contrasted with the capabilities of a traditional relational database in which relations are passive data structures and in which the storage, indexing, and accessing mechanisms for relations are limited to a few fixed, implementations [37]. This is too limiting for most software engineering applications such as the creation of a software environment, where the choice of implementations must evolve over time to meet performance criteria or to incorporate externally developed software tools.

3.4 Common Relation Semantics

Aspen relations are programmed in the sense that the operations on relations, i.e. insert, delete, update, and retrieve, are defined by programs provided by the creator of the relation. In practice, this degree of freedom must be constrained in order to be tractable. Many of the relations in Aspen will share common semantics. It is expected that these semantics will be embedded in programs which will be attached to any relation that desires to use those semantics. We expect any particular realization of Aspen to provide some means of combining programs (e.g., inheritance) to provide the required semantics for any specific relation. Below, we show how some typically desired relation implementations can be programmed.

3.4.1 Extensional Data

Extensional data is the conventional data that is physically stored in some persistent medium. Most relational systems provide this class of data. A programmable system may provide extensional data by programming the insert operation to physically store the inserted data into some form of low-level persistent store such as a file system or a data base [38,21].

3.4.2 Derived Data

Derived data is information that is not physically stored; rather it is computed as needed from other data. This may be simulated by programming the retrieve operator on a relation to compute the derived values at the point when the data is requested. A number of storage and evaluation strategies can be used around derived data:

Caching It is possible to cache derived data, which means that once they is computed, it is saved in persistent storage as long as the data from which it was derived is not altered.

Eager versus Lazy Evaluation If derived data are cached and the data from which it is derived is altered, the derivation could be computed immediately, or it can be delayed to when the derived data is retrieved. Immediate computation is often termed "eager" evaluation and delayed computation is called "lazy" evaluation.

Any of these strategies can be incorporated using programmable relations.

4 APPL/A

APPL/A [39] is an extension to Ada based on the principles of Aspen. APPL/A supplements Ada with a relation unit, a tuple type, and related control constructs including a 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 an APPL/A relation specification appears at the end of this section.

4.1 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.

4.2 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 the 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.

4.3 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 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. This flexibility is acquired at the cost of programming the implementations, but this programming can be simplified by mechanisms, such as Ada generic units, that enable implementations to be reused, and by tools which can automate much of the work.

4.4 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.

4.5 Relation Entries

Operations on relations are effected through entries that are syntactically similar to task entries. Relation entries 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.

4.6 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.

4.7 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.

4.8 "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 a insert event .

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. If this condition is not satisfied then control is logically 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.

During execution a given invocation event is considered to occur when a call to the corresponding relation entry returns successfully, that is, without propagating an exception. Upon completion, the event is appended to all event queues that are interested in it. Thus, the occurrence of an invocation event does not synchronously trigger any **upon** statements that reference it; no **upon** statement is triggered if it is not ready to execute (i.e., available to examine it event queue).

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.

5 Status

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] and to construct a design support system based on the Rational Design Methodolgy of Parnas[31].

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.

```
with wc;      - represents UNIX wc tool
with documents; - another relation

relation Word_Count is
- Relates input and output of the UNIX wc tool
    type wc_tuple is tuple
        text: in text_name;
        lines, words, characters: out natural;
    end tuple;
entries
    insert(text: in text_name);
    delete(select_text: in boolean; given_text: in text_name);
    find(first: in boolean;
        select_text: in boolean; given_text: in text_name;
        select_lines: in boolean; given_lines: in natural;
        select_words: in boolean; given_words: in natural;
        select_characters: in boolean; given_characters: in natural;
        t: out wc_tuple;
        found: out boolean);
    selected(t: in wc_tuple);
dependencies
    text determines lines, words, characters
        by wc(text, lines, words, characters);
constraints
    - each tuple is unique
    all t1 in word_count satisfy
        no t2 in word_count satisfies
            t1.text = t2.text and t1 /= t2;
        end no;
    end all;
    - each text occurs in relation "documents"
    all t1 in word_count satisfy
        some t2 in documents satisfies
            t1.text = t2.text
        end some;
    end all;
End Word_Count;
```


Figure 1: Example APPL/A Relation Specification

- The relational model proved useful for defining abstract types for requirements and simplified extension of those types.
- Particular relationships modeled included logical containment, functional decomposition, derivations between requirements elements, and other associations based on requirements semantics.
- Both intra- and inter-relational constraints were generally useful.
- The activity and reactivity of relations was helpful in maintaining explicit constraints and in propagating updates to the requirements; this simplified the application-level code, which was effectively relieved of these responsibilities.
- The separation of relation specifications from bodies enabled application-level code to be developed independently of the underlying storage system; in REBUS both Ada direct I/O files and the Cactis [21] semantic database were used for storage.
- The opportunity to regard many predicates on requirements specifications as goals to be satisfied eventually rather than as constraints to be met immediately strengthened our belief that the toleration and management of inconsistency is essential.

Ongoing research includes the development of additional software systems based on APPL/A, particularly systems for requirements specification, design, testplan generation, and maintenance. This will enable us to continue to evaluate APPL/A and the concepts behind it. As the ASPEN principles evolve and the APPL/A mechanisms are updated we expect to gain additional insights into the requirements for software object management systems.

6 Related Work

APPL/A is being developed as part of an ongoing program of software environments research at the University of Colorado. The earliest effort, TOOLPACK [30], focused on collections of tools. Within TOOLPACK the Odin system [9] explored the idea of an object base as the integrating element of an environment. Odin also supported a system of inferencing over derivation relations. In conjunction with the Arcadia

project [40], APPL/A is being developed to extend the ideas of Odin to support general, programmable relations over objects in an environment.

APPL/A is not the first system to propose some form of relation as a structuring concept. In [8,24,33] relations are proposed for the storage of complex software structures such as source code, parse trees, and execution states. These systems rely on traditional relational databases (such as INGRES [38]); as a consequence their performance is prohibitively slow.

POSTGRES [36], is an ambitious attempt to extend the relational data model to support software environments. Specifically, POSTGRES extends INGRES with domains over abstract data objects and procedures and supports both forward and backward inferencing. Consequently, POSTGRES offers many of the capabilities of APPL/A, and since POSTGRES is based on INGRES, it is a more mature system. The major flaw in POSTGRES is its relative inflexibility. While individual domain values may be programmable, the implementation of a whole relation cannot be specified. In addition, the relations themselves are not considered objects, and so meta-relations over existing relations cannot be constructed.

Increasingly research environments are using various kinds of object-oriented models to overcome the limitations of relational databases [15,16,22,35]. APPL/A can be considered such a system since it explicitly uses objects as the domains for its relations.

The Encore system [42] is one example of an object-oriented approach. Encore provides an object-oriented database language for a database in which the objects are described by types with operations, properties, and type inheritance, much like Smalltalk [15]. Encore has no built-in notion of relation; rather it must be built up from more primitive concepts. Also, programmability is associated with types rather than with relations.

Other systems that can be considered in this category are DAMOKLES[12], CAIS[1], NuMIL[26]. These offer some interesting features but lack the combination of high-level relations, strong typing, and programmability that is important in APPL/A.

PS-ALGOL [4] represents a third modelling approach. It uses a persistent heap for object storage and uses the language typing system for persistent objects. This allows it to closely control some aspects of the storage of data, but this model is less flexible than the relational model for queries.

Horwitz [19] has proposed an approach to relations that is very similar to APPL/A. She proposes a relational interface to existing data structures, specifically software data, and shows that a properly defined interface allows for certain classes of query

optimization over such interfaces. The basic operations of APPL/A relations are comparable to those she recommends, but with modifications to handle iterators and the *upon* operator.

AP5 [11] is closest to APPL/A in spirit. It too has programmable relations as its primary modelling concept. These relations can support constraints and inferencing and AP5 has a very powerful query language. AP5 is embedded in Lisp as opposed to Ada.

APPL/A has some points in common with Adaplex [35]. Adaplex attempts to embed an object oriented database into Ada by extending the Ada language. Adaplex uses types with attributes as opposed to the n-ary relations favored by APPL/A. Adaplex also does not allow close control over implementation of its modelling structures.

Finally, the notion of using programming languages such as APPL/A to express software relations fits nicely with the notion of process programming [29]. Process programming suggests that the way in which software is developed and maintained can be captured as software itself, and that environments can support and effect the development and maintenance of software by interpreting software process programs. Clearly such software process programs must operate on software objects, and must therefore incorporate specifications of the type systems used to structure environment object stores. Going further, however, process programs must also embody and implement the relations among these objects. Thus process programs must effect software object relation management. From that perspective, a relation management language such as APPL/A is a component of a process programming language.

7 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] *Military Standard Common APSE Interface Set (CAIS). Proposed MIL-STD-CAIS*, Department of Defense, January 31 1985.

[2] U.S. Department of Defense Ada Joint Program Office. *Reference Manual for the Ada Programming Language. ANSI/MIL-STD-1815A-1983*, Department of Defense, Washington D. C., 1983.

[3] M. W. Alford. "A Requirements Engineering Methodology for Real-Time Processing requirements". *IEEE Transactions on Software Engineering*, 3(1):60–69, January 1977.

[4] Malcolm P. Atkinson, Peter J. Bailey, K. J. Chisholm, W. P. Cockshott, and Ronald Morrison. "An Approach to Persistent Programming". *The Computer Journal*, 26(4):360–365, 1983.

[5] Philip Bernstein. "Database System Support for Software Engineering". In *Proceedings 9th International Conference on Software Engineering*, Monterey, California, March 1987.

[6] Per Brinch Hansen. "The Programming Language Concurrent Pascal". *IEEE Trans. on Software Engineering*, 1(2):199–207, 1975.

[7] J. Buxton. *Requirements for ADA programming support environments: Stoneman.* US Department of Defense, OSD/R&D, Washington, D. C., Febuary 1980.

[8] Ceri. "Relational Data Bases in the Design of Program Construction Systems". *SIGPLAN Notices*, 18(11):34–44, November 1983.

[9] 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.

[10] CLF Project. CLF Overview. Univ. of Southern California, Information Sciences Institute, 1985.

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

[12] Klaus Dittrich and Peter Lockmann. "DAMOKLES - A Database Syetem for Software Engineering Environments". In *IFIP WG2.4 International workshop on Advanced Programming Environments*, Trodheim, Norway, June 1986.

[13] 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.

[14] F. Gallo, R. Minot, and I. Thomas. "The Object Management System of PCTE as a Software Engineering Database Management System". In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 12–15, Palo Alto, CA, January 1987.

[15] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[16] M. Hammer and D. McLeod. "Database Description with SDM: A semantic database model". *ACM Transactions on Database Systems*, 6(3):351–386, September 1981.

[17] Dennis M. Heimbigner, Deborah A. Baker, and Stanley M. Sutton, Jr. *Providing Programmable Relations over Software Objects in ASPEN.* Technical Report CU-CS-350-86, Univ. of Colorado, Dept. of Computer Science, Boulder, Colorado 80309, 1986.

[18] C. A. R. Hoare. "Monitors: An Operating System Structuring Concept". *Comm. ACM*, 17(10):549–557, 1974.

[19] S. Horwitz. *Adding Relational Databases to Existing Software Systems: Implicit Relations and a New Relational Query Evaluation Method.* Technical Report 674, University of Wisconsin Computer Science Dept, Madison, Wisconsin, November 1986.

[20] S. Hudson and R. King. "Object-Oriented Database Support for Software Environments". In *ACM SIGMOD International Conference on Management of Data*, pages 491–503, 1987.

[21] Scot Hudson and Roger King. "CACTIS: A Database System for Specifying Functionally-Defined Databases". In *Proc. of the Workshop on Object-Oriented Databases*, pages 26–37, September 1986.

[22] R. King. "Sembase: A Semantic DBMS". In *Proceedings of the 1st International Workshop on Expert Database Systems*, pages 151–171, Kiawah Island, South Carolina, October 24–27 1984.

[23] Leblang. "Computer aided software enginerring in a distributed workstation environment". In *SIGPLAN notices*, May 1984.

[24] M. A. Linton. "Implementing Relational Views of Programs". In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 132–140, Pittsburgh, PA, May 1984.

[25] M. Maybee. *Lessons Learned From Rebus.* Arcadia Document CU–88–01, University of Colorado Computer Science Dept., January 1988.

[26] K. Narayanaswamy, W. Scacchi, and D. McLeod. *Information Management Support for Evolving Software Systems.* Technical Report 85–324, University of Southern California Computer Science Dept, March 15 1985.

[27] 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.

[28] Jack A. Orenstein, Sunil K. Sarin, and Umeshwar Dayal. *Managing Persistent Objects in Ada: Final Techincal Report.* Technical Report CCA–86–03, Computer Corporation of America, Cambridge, Massachusetts, May 1986.

[29] L. J. Osterweil. "Software Processes are Software Too". In *Proceedings 9th International Conference on Software Engineering*, pages 2–13, Monterey CA, March 31–April 2 1987.

[30] L. J. Osterweil. "Toolpack – An Experimental Software Development Environment Research Project". *IEEE Transactions on Software Engineering*, 13(9):673–685, November 1983.

[31] D. L. Parnas. "A Rational Design Process: How and Why to Fake it". *IEEE Transactions on Software Engineering*, 12(2):251–257, February 1986.

[32] M. H. Penedo. "Prototyping a Project Master Database for Software Engineering Environments". In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 1–11, Palo Alto, CA, January 1987.

[33] M. L. Powell and M. A. Linton. "Database Support for Programming Environments". In *Proceedings of the ACM SIGMOD International Conference on Databases for Engineering Design*, pages 63–70, San Jose, CA, May 1983.

[34] James Rumbaugh. "Relations as Semantic Constructs in an Object-Oriented language". In *OOPSLA '87*, pages 466–481, Orlando, Florida, December 1987.

[35] John M. Smith, Steve Fox, and Terry Landers. *Reference Manual for ADAPLEX.* Technical Report CCA–83–08, Computer Corporation of America, May 1981.

[36] M. Stonebraker and L. A. Rowe. "The Design of Postgres". In *Proceedings of the ACM–SIGMOD International Conference on Management of Data*, pages 340–355, Washington, D.C., May 28–30 1986.

[37] Michael R. Stonebraker, editor. *The INGRES Papers: The Anatomy of a Relational Database Management System.* Addison–Wesley Publishing Company, 1985.

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

[39] 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.

[40] R. N. Taylor, L. A. Clarke, L. J. Osterweil, J. C. Wileden, and M. Young. "Arcadia: A Software Development Environment Research Project". In *Second International Conference on Ada Applications and Environments*, pages 137–149, April 1986.

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

[42] S. B. Zdonik and P. Wegner. "A Database Approach to Languages, Libraries and Environments". In *Proceedings of the Workshop on Software Engineering*

Environments for Programming-in-the-Large, pages 89–112, Harwichport, Massachusetts, June 1985.