APPL/A: A Language for Managing Relations Among Software Objects and Processes

Dennis Heimbigner
Stanley Sutton, Jr.
Leon J. Osterweil

CU-CS-374-87 September 1987

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

Abstract

Software products and the processes that create them are characterized by use of a wide variety of relationships among software objects. We describe a research project whose goal is to enable more effective management of software relations in the context of a software process programming language. Our approach is to describe a model based on objects, relations, and processes. Our model differs from most other attempts to use relations by providing programmable semantics for relations and separating their implementation from logical specification. Programmability may be used to implement a variety of storage structures, constraints, and inferencing mechanisms. We introduce an extension of Ada, called APPL/A, to enable programmers to define and manage relations within programs. The extensions to Ada include a relation library unit with separate specification and body, a tuple type, an iterator for relations, and a new select alternative called the *upon* statement. We are defining an automatic translation of APPL/A into Ada, and are applying the language to a prototype process programming system for requirements, design, and testing.

Keywords: Process programming, relations, software environments, object management, Ada.

Table of Contents

1. Introduction

Software products are generally understood to be very complex aggregates of typed objects such as source text, executable code, testcases, requirements specification, documentation, and design element. We believe that good progress is being made toward understanding the components of software products. Far less attention has been devoted to understanding the interrelations among these components. Most large software objects such as source code and designs are, at least implicitly, organized into graphs of component parts. Source code modules are related to corresponding test case modules by complicated derivation processes. Design specifications are related to requirements specifications by complex consistency relations which may not be well understood. It is becoming increasingly clear that much of the complexity of software products derives not simply from the diversity of objects comprising them, but also from the intricate ways in which these objects must be related to each other.

This slighting of relations is unfortunate because their effective management is important to the development of high-quality software products and essential to the maintenance of these products. Relations should guide both developers and maintainers to the precise locations in the particular objects which need alteration and enhancement. Consequently, the explicit recognition of complex relations among software objects is an important element in the specification of software development and maintenance processes. This implies that relation management is especially useful as an adjunct to software process programming ([Osterweil 87]).

In this paper we describe a research project whose goal is to enable more effective management of software relations in the context of a software process programming language. Our premise is that existing languages do not enable software engineers to create and manage relations of sufficient power to model actual software products. We believe that by raising the visibility of relations we can make software more comprehensible, and therefore more maintainable and robust. Our approach is to describe a conceptual formalism and to introduce a programming language to enable programmers to define and manage relations within programs. This language is called APPL/A[1]. APPL/A differs from most other attempts to use relations in software engineering by providing programmable semantics for relations and separating their implementation from logical specification.

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. For example, the relation between two objects may be defined in terms of the execution of an entire sequence of complex programs (eg. software tools), each of which can be implemented using the storage structures best suited for it.

This approach is in contrast to the capabilities of a traditional relational database, in which the storage, indexing, and accessing mechanisms for relations are limited to a few fixed, implementations. This is too

---

[1] "APPL/A" stands for "Ada Process Programming Language with Aspen," where "Aspen" is an earlier formulation of the data model [Heimbigner 86]

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.

The rest of this paper is organized as follows. We first describe our relational models for software objects and software processes and show how these are interrelated. We then present an overview of the language APPL/A, followed by examples of the use of APPL/A to represent selected software relations. Finally we discuss related work and report the status of our work.

2. Models for Software Objects and Processes

We believe that both software products and the processes that create them are characterized by use of a wide variety of relationships among software objects. One aspect of this belief is that relations should be programmable to enable specification of the precise semantics required by particular products and processes. Consequently, relations provide a logical structure in which object and process management can be integrated.

2.1. The Object Model

The object model we use in this work is based on a view of the software product as a large collection of objects of many types, for example, requirements, design, code, tests, analyses, and documentation. However, these objects are logically and physically interconnected by a complex network of relations of many kinds, including derivation, consistency, version, and configuration relations. Some of these relations are imposed by product requirements, while others support the software processes by which the product is developed, evaluated, and maintained.

2.1.1. Objects and Types

Notions of *object* and *type* have been well developed in programming languages. Because the world to be modeled consists of software objects, approaches from object-oriented languages seem especially appropriate. We have adopted the following principles.

Each software object is unique and distinguishable and is an instance of one (or more) object types. A type comprises a set of object values and a set of operations that reference those values. 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(s) 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. A wide variety of type definition mechanisms exist but we have restricted ourselves to using the type definition facilities of Ada [Ada 83] for reasons described in section 3.

2.1.2. 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 [Codd 70]. 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. Thus they are akin to various kinds of object models [Goldberg 83, Hammer 81, King 84, Smith 81]. Additionally, our relations differ from those of conventional relational systems in terms of operations, semantics, and approach to implementation, which are described below.

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 and unique 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.

The operations inherently associated with relations include create and destroy (for relations) and insert, update, delete, and selectively 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.

As stated above, many kinds of relation are important for object management in software engineering; these include at least derivation, consistency, organizational, and temporal relations (and possibly others). Constraints on data, which may also be expressed in relational terms, are also important. The specific relations and constraints appropriate to an environment or project will depend on the requirements of the software product and processes. To be effective an environment must allow the specification of relations with a wide range of semantics. In our model, in addition to the signature[2], one can specify the following semantic properties:

- **Computational dependencies among attributes**: Attributes can be designated as "computed", and a computation can be specified in terms of specific algorithms and inputs; these algorithms can be represented by processes in an environment, and the inputs can include values of other attributes in the relation. In this way it is possible to express specific computational dependencies between attributes of a relation. In software engineering terms, these specifications can be used to define derivation relations among objects; in mathematical terms, these specifications represent a class of functional dependencies among the values of attributes in a relation.
- **Constraints on data**: Computable constraints can be associated with relations; these can be used to restrict the values of individual objects, tuples of objects, and whole relations. In this way it is possible to stipulate existential, referential, and other integrity constraints and to specify functional, multi-valued, and other dependencies in the mathematical sense. Constraints can be represented by programs in an environment, thus the specification of arbitrary programmable constraints is possible.

The model must also allow the specification of the semantics of sets of relations. A software product will be defined in terms of multiple relations and software processes will make use of additional relations; the effectiveness of the processes and the acceptability of the product will depend on the consistency of these relations as a group. Consequently, *inter-relational* dependencies are important. An approach to achieving consistency between relations is *inter-relational inferencing* in which operations on one relation

[2]The combination of relation name, attribute names, and domain types is the signature.

trigger operations on another. For example, triggered operations can retrieve data from one relation to use in the computation of a value in another, and they can propagate the effects of changes in one relation to other, dependent relations.

2.2. The Process Model

Our process model is based on the notion of Process Programming as described in [Osterweil 86, Osterweil 87]. This model suggests that complex activities such as software development, software maintenance and all of their subactivities (eg. requirements specification and testing) can be modelled on computer processes and that such software processes should be described formally by means of computer programming techniques. Process programs should be written to describe the way in which that activities of software workers are to be coordinated with those of automated tools to produce lower level software objects and to integrate them into software products.

Analogous to a computer program, a software process program consists of code and data. A software process is the execution of a software process program by an interpreter. The interpreter may be either a computer or a person, depending on the nature of the process. Software tools function as operators in the code and software objects serve as the operands for these operators. Software processes execute in the context of a software environment, which contains the supporting interpreters, tools, and objects.

2.3. Interrelation of the Object and Process Models

Neither the object model nor the process model alone is sufficient for a software environment. The object model emphasizes the importance of relations with programmable semantics and implementation. Strictly speaking, it says nothing about how the semantics and implementation are to be programmed and it says nothing about how the relations are to be used. Similarly, the process model emphasizes that software processes should be formally specified in a programming language, but it says nothing about the algorithms for those processes or the data structures on which they operate.

To resolve this dilemma, we propose that software processes should define the semantics for relations and relations should define the mechanism through which processes access data. In this way, relations and processes complement each other and provide an integrated model for software process programming. Thus the notion of relation provides the logical structure which serves to unify the object and process models.

Relations incorporate software processes through programmable semantic specifications. Computational dependencies and constraints are defined in terms of software processes, and software processes are triggered to effect interrelational inferencing. Additionally, software processes define the implementation of relations in the environment.

Conversely, a collection of software processes characterizes a software product and supports its development, and the processes do this principally by elaborating and maintaining relations over software objects. Consequently, it is appropriate to express software processes directly in terms of operations on relations. Moreover, since the schema and semantics of defined relations will depend in part on the requirements of software processes, software processes can be simplified by the use of appropriate relations.

The above considerations suggest that programmable relations provide a coherent framework for the management of objects and processes in a software process programming environment. A key obstacle to writing process programs at present is the lack of a language which is capable of supporting the precise encoding of process programs. We believe that any such language must support relations with a wide range of semantics and implementations. We view the inability of most languages to deal effectively with such relations as a very serious deficiency. The work described in the next section is an attempt to remedy this deficiency.

3. Development of a Process Programming Language Based on Relations

In order to experiment with and evaluate the process and object models described above we are designing a programming language that includes a "relation" construct based on those ideas. The basic goals of the language are

- support the relation-based object and process models described above;
- separate the specification of a relation from its implementation in a storage system;
- facilitate the programming of software processes.

Our approach to the design of a process programming language to meet the above goals has been to extend an existing language. This has enabled us to concentrate on the features of special interest. We selected Ada as our starting point. Ada has several advantages, including support for abstraction of processes and data, multi-process programming through tasking, deterministic and non-deterministic control constructs, and multiple types of program unit. Additionally, several of the features of Ada provide useful models for the relational constructs to be added.

The main points of our approach to extending Ada are as follows:

- *Relations* are added as a new kind of library unit and *tuples* are added as a new type constructor.
- New statements are added to iterate over relations and support interrelational inferencing.
- The syntax and semantics of Ada constructs were used as models for those of the new features wherever appropriate.

As a result, the new relational features resemble Ada constructs, and programs that make use of these features resemble Ada programs.

4. APPL/A Syntax and Semantics

This chapter discusses the syntax and semantics of selected features of APPL/A as extensions to Ada. A formal syntax for APPL/A has been prepared, but it is omitted here for brevity; the syntax is illustrated in examples in section 5.

4.1. Relations

APPL/A incorporates *relations* as an new kind of library unit. Unlike other library units, but like tasks, relation units may represent instances or types. Relations can also be declared in basic declarations. Relations have specifications and bodies comparable to those of packages and tasks.

A relation specification includes (in order)

- A mandatory tuple type definition.
- Optional relation entry declarations, which represent operations on the relation.
- Optional computational dependency specifications, which may be used to indicate the ways in which attribute values are computed.
- Constraint specifications, which indicate conditions that must be satisfied by attribute values within tuples and by tuples within the relation. Constraints should be specified where appropriate.

Each of these elements is discussed in a following section.

The relation body is syntactically similar to a package or task body; because operations on relations are effected through entry calls, relation bodies tend to resemble task bodies in practice. In general the body of a relation will contain accept statements for the entries of the relation, which in turn will reference an underlying storage system. The choice of a storage system and the implementation of the operations is not constrained by the language. On the contrary, the programming of the body of a relation affords the opportunity to implement alternative strategies for the storage and computation of attributes and for inferencing between relations (as shown in the examples).

4.2. Declaration of Tuples and Attributes

Tuple declarations are analogous to record declarations. The principal difference between tuple and record types is that tuple attributes have modes whereas record components do not. The attribute modes are just those for subprogram and entry parameters: **in**, **out**, and **in out**; the default attribute mode is **in**. The purpose of the modes is to indicate the ways in which attributes can receive values. Attributes of mode **in** must be given values by the user through calls to relation entries. Attributes of mode **out** may not be given by the user but must be computed within the body of a relation. Attributes of mode **in out** can take on given and computed values in sequence. The modes of attributes also constrain the parameter modes of relation entries.

Operations on tuples and attributes are restricted to help ensure that the attribute values of any tuple object are consistent with the state of the relation from which it was retrieved. Attribute values within a tuple can only be assigned in the body of a relation, although they can be referenced anywhere. Tuple values as a whole can be tested for equality and assigned.

4.3. Relation Entries

The syntax of relation entry declarations is a subset of that for task entry declarations. Relation entries fall into two groups: declarable and non-declarable (or implicit). The declarable relation entries are restricted to "insert", "delete", and "update"; a relation specification may include declarations for any subset of these. They are used to effect the corresponding operations on the relation, and they are directly callable from user programs. The implicit entries are "find" and "selected"; these are not declared but are assumed to be available for any relation. They are used during retrieval from relations. They are not directly callable by user programs; instead they are invoked indirectly during the iterative retrieval process (as described in section 4.5).

The formal parameters of relation entries must conform to rules that ensure consistency with the corresponding tuple type definition and that provide a comparable interface from relation to relation. For example, the "insert" entry may take one **in** or **in out** parameter for each attribute of that mode in the

definition of the tuple type for the relation (and no others), and the delete entry must take a single **in** parameter of the tuple type for the relation. Similar rules apply to the other entries, both declarable and implicit.

Relation entries provide a standardized interface to a relational abstract data type. This interface is independent of the underlying storage structures, which may be relational or non-relational. Consequently, the storage structures can be changed without affecting programs that make use of relations, so long as the relation specifications remain unchanged.

APPL/A requires no extensions to the syntax or semantics of Ada **accept** statements, except that they can be used to accept calls to entries for both tasks and relations. The **accept** statement for relation entries will implement the operations on the relation in terms of programmed strategies for the storage and computation of attributes and inferencing between relations. For example, the values of **out** attributes can be computed upon insert (an "eager" strategy) or upon retrieval (a "lazy" strategy); once computed these values may be stored for future reference, and they may be propagated to other relations.

4.4. Dependency and Constraint Specifications

Dependency and constraint specifications provide a means to stipulate the semantics of relations. The purpose of a dependency specification is to indicate the way in which attribute values are computed; an especially important case of this is computational dependencies among attributes of a single relation. Computations are specified by reference to named subprograms or entries. For example, for a relation R, with attributes A1 and A2, and procedure P, which computes a value of type A2 from one of type A1, a dependency specification may stipulate that "A determines A2 by P(A1, A2)" (see the examples). Reference to a specific computation may be omitted (for example, "A1 determines A2"). A relation may have multiple dependency specifications; alternatively, the dependency specification may be omitted entirely even when the values of some attributes are computed. The values of the computed attributes of a relation must be determined in accordance with any relevant dependency specifications.

The purpose of constraint specifications is to stipulate predicates that must be true of the relation as a whole or of tuples or attribute values within it. The predicates include Ada boolean expressions supplemented by special forms of *relational predicate*. These are based on constructs suggested by Abbott [Abbott 86], and include conditional and quantified expressions (the details of which are omitted here). The constraints for a relation must be characteristic of all tuples, individually or in combination, that are retrievable from that relation.

4.5. Iteration over Relations

Retrieval of the tuples of a relation is provided through an iterative construct in the form of a loop statement:

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

where R is a relation, t is a variable of the tuple type of R, P is an optional selective predicate (a list of values to be matched by tuples selected from R), and S is a (possibly compound) statement. S is executed once for each tuple in the relation with values that match P. The order in which tuples are

retrieved is not defined by the language.

Retrieval is effected through calls on the implicit "find" and "selected" entries for the relation. The semantics of iterative retrieval is defined in terms of a "while" loop and these entries. A statement of the above form is equivalent to

```
R.find(first, P, t, found);
while found loop
   R.selected(t);
   S;
   R.find(next, P, t, found);
end loop;
```

where "R.find(first, P, t, found)" is a shorthand notation (for purposes of discussion) that means "search for the first tuple in R that satisfies P; if one exists then return it in t and set 'found' to true, otherwise set 'found' to false", and where "R.find(next, P, t, found)" has an analogous meaning. The purpose of the "find" entry is to retrieve stored data for the tuple and allow the computation of any attribute values that remain to be determined. The purpose of the "selected" entry is to signal that the given tuple has been selected and thereby trigger actions that are dependent on the selection.

4.6. Upon and Select Statements

The **upon** statement is used to specify the conditional triggering of a sequence of statements by operations on relations. Its purpose is to enable a relation to respond dynamically to events in the environment. For example, an **upon** statement can be used to trigger the transfer of data between relations. Thus, it can be used to effect interrelational inferencing.

The general form of an **upon** statement is

```
upon invocation_condition invoke
        S;
end upon;
```

where the *invocation_condition* identifies an operation on a relation.

The **upon** statement is implemented logically as follows. An *event queue* is associated with each process that may execute an **upon** statement. The event queue will contain records of events that are referenced in invocation conditions in **upon** statements in the process. When an event occurs a record for it is placed in the event queue of each process that has referenced it. These records are queued in the order in which the events occur. The records are then available to trigger the the corresponding **upon** statements in the processes, thus enabling the processes to respond asynchronously to the events.

Upon statements are somewhat analogous to Ada **accept** statements. When a process executes an **upon** statement it is suspended until the "invocation condition" in the statement is satisfied (if ever); at that time the record for the corresponding event is removed from the event queue for the process, and the sequence of statements in the body of the **upon** statement is executed. This is not a "rendezvous" in the Ada sense, however: There is no necessary synchronization of processes across an **upon** statement. The process that establishes the invocation condition is not suspended until the **upon** statement is completed; moreover, all processes waiting on an invocation condition are triggered by the realization of that condition. (In effect a notice of the event is "broadcast".) The **upon** statement is like the Ada **accept** statement in one other respect, however. No **upon** statement is triggered if it is not ready to execute, just

as no **accept** statement can receive an entry call if the **accept** is not ready to execute.

Upon statements can appear in any sequence of statements; they are not restricted to relation bodies. In particular **upon** statements may be used in **select** statements.

The APPL/A **select** statement is identical to the Ada **select** statement with the addition of the "upon_alternative" to the "selective_wait_alternative". This enables the selective-wait statement to be used to express the response of a relation to both entry calls and invocation events.

5. Examples

This chapter presents examples of two associated APPL/A relations. The specifications of the relations are shown in detail, including tuple type definitions, computed attributes, and computational dependencies. The high-level structures of the bodies are outlined; the implementations of the relations, as programmed in the bodies, differ with respect to computation and storage strategies and use of interrelational inferencing.

5.1. Relation C_to_Obj

Relation C_to_Obj (see figures 1 and 3) relates C source code to the object code that is compiled from it. The relation references two other library units, a procedure "invoke_cc", which compiles the source code, and a relation "Obj_to_Exe", which relates object code to the executable code linked from it. (The relation Obj_to_Exe is not included here because of space limitations; it has a specification and implementation that are directly analogous to those of C_to_Obj.)

C_to_Obj has a tuple type with four attributes: two "in" attributes, the C code and a "debug" flag for the complier, and two "out" attributes, the object code and a diagnostic report. The debug attribute has a default value of "false". The object code and diagnostic report are computed automatically from the C code and debug flag using the procedure "invoke_cc", which invokes the C compiler (this computation is programmed in the body of the relation). The relation includes entry declarations for insert, update, and delete operations; it has no constraints.

The sequence of statements in the body consists of a single, non-terminating loop statement over a single **select** statement. The **select** statement includes five alternatives, one **accept** statement for each of the three declared entries and one for each of the implicit entires "find" and "selected".

The low-level code within the select alternatives is omitted for the sake of simplicity since it depends on the storage system used and since many of the details are uninteresting. The intended behavior of each alternative is described by comments that indicate the computation, storage, and inferencing strategies to be used. For example, the object code and diagnostics are computed in the "insert" entry as soon as the C code and debug parameter are available, thus effecting an "eager" evaluation strategy, and the computed values are stored immediately for future retrieval.

5.2. Relation C_DFA

Relation C_DFA (see figures 2 and 4) relates C code to a dataflow analysis that is derived from it by means of a dataflow analysis tool. The relation references a subprogram that invokes the analysis tool and the relation C_to_Obj.

The relation has a tuple type with one "in" attribute for the C code and one "out" attribute for the analysis. The analysis is computed automatically from the C code using the procedure "invoke_c_dfa". For the sake of simplicity the relation has only insert and delete entries; no update operation as such is possible. The relation has no constraints.

The gross structure of the body of C_DFA resembles that for C_to_Obj: a non-terminating loop over a **select** statement. The **select** statement has seven alternatives, one **accept** statement each for the entries "insert", "delete", "find", and "selected", and one **upon** statement each for invocations of "C_to_Obj.insert", "C_to_Obj.update", and "C_to_Obj.delete." As in the body for C_to_Obj, the details of the select alternatives have been omitted but the intended effect of each alternative has been described by comments. For example, C code values are stored upon insertion without the corresponding dataflow analysis, and the dataflow analysis is computed upon retrieval instead of insertion, thus effecting a kind of "lazy" evaluation.

Upon statements are used to respond to operations on the C_to_Obj relation, in particular to ensure that C code that is represented in C_to_Obj is also represented in C_DFA. In this way data are propagated between relations by a kind of "backward" inferencing.

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 [Osterweil 83], focused on collections of tools. Within TOOLPACK the Odin system [Clemm 86] 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 [Taylor 86], 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 [Ceri 83, Linton 84, Powell 83] 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 [Stonebraker 76]); as a consequence their performance is prohibitively slow.

POSTGRES [Stonebraker 86], 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 [Goldberg 83, Hammer 81, King 84, Smith 81]. APPL/A can be considered such a system since it explicitly uses objects as the domains for its relations.

The Encore system [Zdonik 85] 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 [Goldberg 83]. 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 [Dittrich 86], CAIS [CAIS 85], NuMIL [Narayan 85]. 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 [Atkinson 83] 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 [Horwitz 86] 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 [Cohen 87] 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.

Finally, APPL/A has some points in common with Adaplex [Smith 81]. 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.

7. Project Status and Future Work

The definition of the syntax and semantics of APPL/A is complete. We have found that it supports the major features of our models. As we gain experience with the language we expect that the syntax and semantics will be refined and extended.

We have begun to define automatic translators of APPL/A constructs into Ada. A relation is translated into an Ada package. The tuple type definition is translated into a record type definition within the package specification, and the relation entries are translated into a task specification within the package specification. The relation body is translated into the task body within the package body. The "for" loop to iterate over relations is translated into a "while" loop as described above. The **upon** statment is translated into various conditional constructs, depending on the context. The signalling required for **upon** statements is implemented by supplementing an APPL/A system with additional tasks for this purpose. An "Event_Queue" task is associated with each relation to represent the logical event queue in terms of which the **upon** statement is defined. Each system is also supplemented with a single "Event_Monitor"

task. Each relation body is instrumented to signal the system Event_Monitor each time an entry is called; the event monitor then signals the Event_Queue of each task with an **upon** statement that references that event.

Using these translations we have been able to program a simple APPL/A system comprising a main procedure, two relations, and two tasks which we have translated into a single Ada program and executed successfully. The main procedure provides a simple interface to the relations that enables a user to insert, delete, and retrieve data; among other things this procedure makes use of the iterative construct. A "Squares" relation associates integers with their squares. The integers are constrained to be non-negative. The squares are computed from given integers upon insertion and the integer-square pairs are then stored; thus this relation uses an eager evaluation strategy with caching of computed data. Insertion of an integer into this relation triggers insertion of the same integer into the other relation, a "Cubes" relation, thus effecting a kind of forward inferencing. The Cubes relation associates integers with their cubes. The cubes are computed upon retrieval but are not stored; thus this relation uses a lazy evaluation strategy without caching of computed data. The tasks include **upon** statements that enable them to respond to operations on the relations. The implementation of this system was simplified in some respects because it was a single program. We are generalizing this approach to multi-program systems.

We are also using APPL/A in a prototype software process program for a "requirements builder", a program that coordinates the efforts of workers in developing a requirements specification. The requirements specification is regarded as a DAG of requirements elements some of whose values are entered manually and some of whose values are computed. Values of both types may be constrained. APPL/A relations are used to model, support, and enforce the DAG's lattice structure as well as relations among the values of the elements. This prototype is being generalized to encompass other phases of the development process, including design, testing, and maintenance; all phases will be supported by APPL/A relations. Finally, the phases will be combined into a programmable system for the specification of software processes; relations will be an important integrating mechanism for this system. Additionally, relations will be used to support dynamic typing for the system.

We also expect to make use of APPL/A to experiment with alternative storage systems and with strategies for computation, storage, and inferencing. In early work we have simply used Ada direct I/O files for storage. We are incorporating CACTIS [Hudson 87], a semantic database, as the storage system in ongoing work. As the process programming prototype is developed and used it will afford the opportunity to experiment with different approaches to the computing and caching of data and different schemes for interrelational inferencing. In this way we will learn more not only about object management for software processes but also about the processes themselves and the interrelation of object and process management.

Acknowledgements

The authors gratefully acknowledge the support of the National Science Foundation cooperative agreement #DCR-8420944, National Science Foundation grant #DCR-8745444 in cooperation with the Defense Advanced Research Project Agency, Department of Energy grant #15537612, and the American Telephone and Telegraph Company. In addition, the authors wish to thank Deborah Baker, Roger King, Shehab Gamalel-din, and Mark Maybee for their advice and encouragement. The comments of the members of the Arcadia consortium were also important in clarifying the issues surrounding APPL/A.

References

[Abbott 86] R. J. Abbott, *An Integrated Approach to Software Development*, John Wiley & Sons, New York, 1986.

[Ada 83] Ada Joint Program Office, U.S. Department of Defense,*Reference Manual for the Ada Programming Language*,ANSI/MIL-STD-1815A-1983, U. S. Government, 1983.

[Atkinson 83] M. P. Atkinson, et al,"An Approach to Persistent Programming," *The Computer Journal* 26(4):360-365.

[CAIS 85] *Military Standard Common APSE Interface Set (CAIS)*, Proposed MIL-STD-CAIS, January 31, 1985.

[Ceri 83] "Relational Data Bases in the Design of Program Construction Systems," *SIGPLAN Notices* 18(11):34-44 (November 1983).

[Clemm 86] G. Clemm, *The Odin System: An Object Manager for Extensible Software Environments*, University of Colorado Ph. D. Thesis, also Computer Science Technical Report CU-CS-314-86, February 1986.

[Codd 70] E. F. Codd, "A Relational Model for Large Shared Data Banks," *Communications of the ACM* 13(6):377-387.

[Cohen 87] D. Cohen, *AP5 Manual*, March 27, 1987.

[Dittrich 86] K. R. Dittrich, W. Gotthard, and P. C. Lockemann, "DAMOKLES - A Database System for Software Engineering Environments," *Proceedings of the International Workshop on Advanced Programming Environments*, IFIP WG2.4, Trondheim Norway, June 1986, pp. 353-371.

[Goldberg 83] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison Wesley, 1983.

[Hammer 81] M. Hammer and D. McLeod, "Database Description with SDM: A semantic database model," *ACM Transactions on Database Systems* 6(3):351-386 (September 1981).

[Heimbigner 86] D. Heimbigner, D. Baker, and S. M. Sutton, Jr., "Providing Programmable Relations over Software Objects in Aspen," University of Colorado Technical Report CU-CS-350-86, University of Colorado, Boulder, 1986.

[Horwitz 86] S. Horwitz, "Adding Relational Databases to Existing Software Systems: Implicit Relations and a New Relational Query Evaluation Method," University of Wisconsin, Madison, Computer Sciences Technical Report 674, November, 1986.

[Hudson 87] S. Hudson and R. King, "Object-Oriented Database Support for Software Environments," *ACM SIGMOD International Conference on Management of Data*, 1987, pp. 491-503.

[King 84] R. King, "Sembase: A Semantic DBMS," *Proceedings of the 1st International Workshop on Expert Database Systems*, Kiawah Island, South Carolina, October 24-27, 1984, pp. 151-171.

[Linton 84] M. A. Linton, "Implementing Relational Views of Programs," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, PA, May 1984, pp. 132-140.

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

[Osterweil 83] L. J. Osterweil, "Toolpack -- An Experimental Software Development Environment Research Project," *IEEE Transactions on Software Engineering* SE-13(9):673-685 (November 1983).

[Osterweil 86] L. J. Osterweil, "Software Process Interpretation and Software Environments," University of Colorado, Boulder, Department of Computer Science Technical Report

CU-CS-324-86, May 1986.

[Osterweil 87] L. J. Osterweil, "Software Processes are Software Too," *Proceedings 9th International Conference on Software Engineering*, Monterey CA, March 31-April 2, 1987, pp. 2-13.

[Powell 83] M. L. Powell and M. A. Linton, "Database Support for Programming Environments", *Proceedings of the ACM SIGMOD International Conference on Databases for Engineering Design*, San Jose, CA, May 1983, pp. 63-70.

[Smith 81] J. M. Smith, S. Fox, and T. Landers, *Reference Manual for Adaplex*, Computer Corporation of America, Cambridge, Massachusetts, January 1981.

[Stonebraker 76] M. R. Stonebraker, E. Wong, P. Kreps, and G. D. Held, "The Design and Implementation of INGRES," *ACM Transactions on Database Systems* 1(3) (September 1986).

[Stonebraker 86] M. Stonebraker, and L. A. Rowe, "The Design of Postgres", *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, Washington, D.C., May 28-30, 1986, pp. 340-355.

[Taylor 86] R. N. Taylor, L. A. Clarke, L. J. Osterweil, J. C. Wileden, and M. Young, "Arcadia: A Software Development Environment Research Project", *Second International Conference on Ada Applications and Environments*, April 1986, pp. 137-149.

[Zdonik 85] S. B. Zdonik, and P. Wegner, "A Database Approach to Languages, Libraries and Environments," *Proceedings of the Workshop on Software Engineering Environments for Programming-in-the-Large*, Harwichport, Massachusetts, June 1985, pp. 89-112.

```
with invoke_cc;
with Obj_to_Exe;

relation C_to_Obj is
--
   type c_to_obj_tuple is tuple
      c: in c_code;
      debug: in boolean := false;
      obj: out object_code;
      diagnostics: out diagnostic_text;
   end tuple;

   entry insert(c: in c_code; debug in boolean := false);
   entry update(t: in c_to_obj_tuple; c: in c_code := t.c;
      debug in boolean := false);
   entry delete(t: in c_to_obj_tuple);
--
dependencies
--
   c, debug determine obj, diagnostics
      by invoke_cc(c, debug, obj, diagnostics);
--
end C_to_Obj;
```

Figure 1: Specification for Relation C_to_Obj

```
with invoke_c_dfa;
with C_to_Obj;

relation C_DFA is
--
   type c_dfa_tuple is tuple
      c: in c_code;
      dfa: out c_dfa_text;
   end tuple;

   entry insert(c: in c_code);
   entry delete(t: in c_dfa_tuple);
--
dependencies
--
   c determines dfa
      by invoke_c_dfa(c, dfa);
--
end C_DFA;
```

Figure 2: Specification for Relation C_DFA

```
relation body C_to_Obj is
--
   t: c_to_obj_tuple;
--
begin
--
   loop
      select
         accept insert(c: in c_code; debug in boolean := false) do
         -- Initialize a new tuple with the given values of c and debug,
         -- invoke the compiler to compute values for obj and diagnostics,
         -- and store the result. If compilation is successful then
         -- insert obj and debug into Obj_to_Exe.
         end insert;
      or
         accept update(t: in c_to_obj_tuple; c: in c_code := t.c;
            debug in boolean := false)
         do
         -- Update the given tuple with the given values of c and debug,
         -- then proceed as above for insert.
         end update;
      or
         accept delete(t: in c_to_obj_tuple) do
         -- Delete the given tuple from storage; find the corresponding
         -- tuple in Obj_to_Exe and delete it as well.
         end delete;
      or
         accept find(first: in boolean := false;
                  c: in c_code := NULL; match_c := false;
                  debug: in boolean := false; match_debug := false;
                  obj: in object_code := NULL; match_obj := false;
                  diagnostics: in diagnostic_text := NULL;
                        match_diagnostics := false;
                  t: out c_to_obj_tuple) do
         -- Simply retrieve the first (or next) tuple that matches
         -- the given values, if any, and return it.
      or
         accept selected(t: in c_to_obj_tuple);
         -- Null
      end select;
   end loop;
--
end C_to_Obj;
```

Figure 3: Body for Relation C_to_Obj

```
relation body C_DFA is
begin
--
   loop
      select
         accept insert(c: in c_code) do
         -- Initialize a new tuple with the given value of c; do
         -- not compute the corresponding value of dfa at this time
         -- but store the tuple in an "incomplete" form.
         end insert;
      or
         accept delete(t: in c_dfa_tuple) do
         -- Delete the given tuple from storage.
         end delete;
      or
         accept find(first: in boolean := false;
                     c: in c_code := NULL; match_c: in boolean := false;
                     dfa: in c_dfa_text := NULL;
                        match_dfa: in boolean := false;
                     t: out c_dfa_tuple) do
         -- Retrieve the first (or next) tuple that matches (or may match)
         -- the given values. If the value of dfa has not been computed
         -- for the retrieved tuple then compute it at this time. If the
         -- the tuple does match the given values then return it.
         end find;
      or
         accept selected(t: in c_dfa_tuple)
         -- If the dfa value has just been computed then store the tuple
         -- with this new value.
         end selected;
      or
         upon C_to_Obj.insert(c, debug) invoke
         -- Initialize a C_DFA tuple with the value of c just inserted
         -- into C_to_Obj and store the tuple as for insert.
         end upon;
      or
         upon C_to_Obj.update(c_to_obj_t, c, debug) invoke
         -- If the value of c is updated for a tuple in C_to_Obj
         -- then delete the corresponding tuple in C_DFA and
         -- create a new one with the updated value of c.
         end upon;
      or
         upon C_to_Obj.delete(c_to_obj_t)
         -- Delete the corresponding tuple from C_DFA.
         end upon;
      end select;
   end loop;
--
end C_DFA;
```

Figure 4: Body for Relation C_DFA