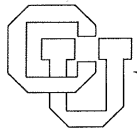
Providing Programmable Relations Over Software Objects in Aspen

D. Baker, D. Heimbigner, S. Sutton, Jr.

CU-CS-350-86 December 1986



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

ABSTRACT

Aspen is a system of managing large collections of software objects. It combines object-oriented and relational concepts as its modelling primitives. Aspen is unique in that it allows its users to program the semantics of relation query and update by associating specific programs with the various query patterns over relations. This allows for a flexible and extensible object manager. A wide variety of access and storage patterns are possible, including simple stored relations, lazy evaluation, forward and backward inferencing, and constraint checking.

1. Introduction

It is recognized that software design environments are characterized by a wide variety of typed objects and an equally wide variety of relationships over those objects. For example, there may be objects of type Source, Binary, Library, Project-plan, and Requirements. There may be relations such as *Compiles_into(Source,Binary)*, which relates a source file to the binary into which it compiles. There may also be a relation *Implements(Requirements,Source)*, which relates a set of requirements to the Source object that implements those requirements. A key problem for such environments is the maintenance of these relationships in the face of evolution of the system. Another major problem is the need to support a wide variety of storage and access mechanisms; many existing programs require data in certain highly encoded formats (files of text, or bit-maps, for example).

Aspen is a system designed to support such large collections of objects, types, and relations. It was developed in conjunction with the Arcadia project [Taylor 86], which has as its goal the development of a software environment for Ada[2] [ALRM 83]. It is unique in providing programmable semantics for various relations. This allows for a wide variety of storage and access mechanisms for relations. In addition, this feature also allows for the inclusion of forward and backward inferencing and constraint monitoring as means of maintaining the relationships over various objects in the system.

2. Aspen Architecture

In terms of implementation, Aspen is intended to serve a role similar to a file system, namely providing a pervasive support for persistent (i.e., long-term) storage of objects. It consists of an object-store that uses non-volatile memory to store the persistent objects. Aspen controls the interface between the

[2] Ada is a registered trademark of the US Government, ADA Joint Program Office.

object-store and the various programs.

In terms of functionality, Aspen may be viewed as a multi-layer system (see figure 1). Layer 1 is the basic operating system upon which Aspen is constructed. This may include an existing file system (such as the Unix file system). Layer 2 is Aspen proper. It uses the facilities provided by layer 1 to implement its functionality. It may best be viewed as an extension/replacement for the underlying file system. Layer 3 is a set of concepts that are recognized as universally important to environments, but whose detailed implementation may vary from environment to environment. Examples of such capabilities might include version control, immutability, and specific software methodologies. The capabilities provided by Aspen are intended to be powerful enough to allow users to implement layer 3 functions using Aspen. A particular software environment is expected to define its various global policies about versions, etc.,

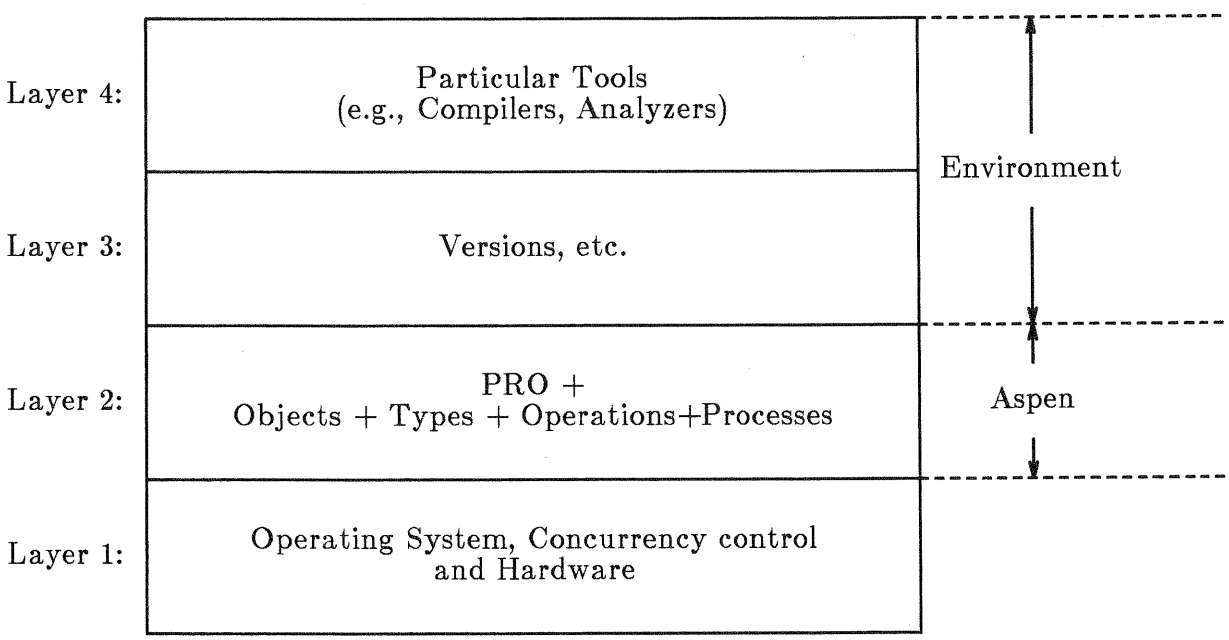


Figure 1. Aspen Functional architecture..

in layer 3. Several different implementations of layer 3 may coexist on a single system. Layer 4 is a set of user defined programs of various kinds. For example, it might contain specific compilers and analyzers. It is in layer 4 that a programmer does his/her actual software design and implementation using the programs provided there.

3. Aspen Modelling Concepts

The modelling concepts in Aspen consist of objects, types, operations, processes, and relations. For purposes of presentation, this set of concepts is described in three parts. Part 1 describes objects, types, and operations. Part 2 describes processes and the programs that implement them. Part 3 describes PRO, which defines the programmable relations concept.

3.1. Basic Modelling Primitives

The basic modelling language most closely resembles Smalltalk [Goldberg 83] or Objtalk[Rathke 83]. It consists of the following concepts:

(1) **Object**: The system to be modelled is assumed to consist of objects. Each object is unique and can be distinguished from all other objects. Objects have an associated state that can be examined and modified. Essentially everything known to Aspen is represented as an object.

All objects known to Aspen are persistent. This means that an object need not vanish as a result of the normal termination of a program execution[3]. As a consequence, persistent objects may have a lifetime longer than the program executions that create or manipulate them. The physical nature of persistence may differ from object to object depending upon its type (see below). Thus, some objects might be made persistent by storing in a

[3] In practice, this means that the object is stored on non-volatile memory.

persistent store [Atkinson 83], some might be stored in a database, some might be stored in files, some might be stored on tape, and some might be stored on paper.

(2) **Type:** All objects are created as instances of specific types. A type consists of a set of object values and a set of operations (see below).

Aspen will enforce strong typing within its confines. Explicit coercion operations must be provided for situations in which it is necessary to change the type of an object (except, of course, from subtype to supertype).

Types may have multiple supertypes (hence multiple subtypes). When T is declared to be a supertype of S, it means that every object of type S is considered to also be of type T. Some supertypes are termed "Mixins" [Rathke 83] because they are used to augment a type with certain kinds of behavior rather than constituting a fully formed type of their own.

(3) **Operations:** Operations define the structure of the state associated with objects of a given type. Operations may be used to read and write the memory of associated objects.

Operations are inherited along the supertype/subtype lines. That is, if type T1 is a subtype of type T2 and O is an operation defined on T2, then O may be applied to objects of type T1 as well as to objects of type T2.

It should be emphasized that the basic set of primitives is intended to be "closed", which means that it can be and is used to represent itself. This kind of closure has been achieved in both Smalltalk and Objtalk. In Aspen types and operations are themselves objects, and there is a type *Type* and a type *Object*, the latter being the supertype for all other types. These objects are often called meta-objects and the collection of meta-objects is called meta-data. Meta-objects, like other objects, will be accessible to programs (subject to

protection restrictions).

Aspen assumes that all other concepts needed for a working design system can ultimately be constructed out of these primitives plus processes and relations.

3.2. Processes and Programs

Processes[4] include both automated and manual procedures. Automated processes may informally be considered to fall into two general classes: traditional tools and software process programs. Both of these types of process are represented in Aspen by executable objects. The execution of these processes occurs under the control of Aspen. Manual processes, to the extent that they are known to Aspen, will be represented by non-executable objects. The execution of manual processes will occur beyond the control of Aspen, but Aspen will still be able to represent these processes in relations with other objects in the system.

Tools are traditional computer programs (or sets of programs), such as compilers and data flow analyzers, written in traditional programming languages.

Software process programs are programmatic expressions of the software development process formulated in a software-process programming language [Osterweil 86]. The purpose of software process programs is to formalize and automate the activities of the software development process in order to make them amenable to the same sorts of programming, analysis, and testing techniques that are applicable to conventional computer programs. Because of the complexity of activities required for software development, some software process programs may be expressible as algorithmic procedures, while others may be represented by logic (Prolog-style) programs that depend on inferencing

[4]The term process is used here in a general sense as opposed to the specific meaning in, say,

mechanisms. Thus, it is expected that software process programs will be composed of other software process programs, manual processes, and traditional tools.

All processes in Aspen will belong to the type *Process*. Automated processes may be represented by the type *Automated_Process*, while manual processes may be represented by the type *Manual_Process*. Both of these would be subtypes of the type *Process*, but the definition of *Automated_Process* may be supplemented by the "mixin" *Executable_Process*. Tools and process programs in particular may be represented as objects of type *Tool* and type *Process_Program*, respectively, which in turn would be subtypes of the type *Automated_Process*, and which would thereby inherit the property of executability.

In the rest of this paper, the terms program, tool, and process may be used interchangeably since all will have an ultimate representation as a computer program.

3.3. *PRO* - Programmable Relations over Objects

In addition to objects, types, operations, and processes, Aspen provides a specialized notion of programmable relations over objects (PRO) as an additional modelling concept. In Aspen a relation is a distinguished type of object that is used to structure the organization of processes and objects in general. All objects exist in the context of one or more relations, and processes are used to define the semantics of relations, as explained below. Thus PRO is built upon and integrates the lower-level modelling concepts.

The semantics of relations are specified in terms of query patterns over the relation. A pattern is an instantiation of the relation with some of the attri-

---

Unix.

butes given and some to be determined. For an N-ary relation, there are 2**N patterns. Consider the binary relation *R(x,y)*. This has four patterns, as follows, where ? indicates an unknown:

R(x?,y?)

R(x,y?)

R(x?,y)

R(x,y)

Associated with each pattern may be a unique process that defines the semantics of queries that match that pattern. That process must take the given attributes and fill in the unknown attributes. For example, the process associated with *R(x,y?)* is given the value x, and must compute a value for y. Processes need not exist for all possible patterns, in which case certain queries cannot be solved, and an error results. Because each query pattern has at most one defining process associated with it, the choice of process for any query is not ambiguous. These processes may be arbitrarily complex and may operate algorithmically or inferentially. They may be tools, manual processes, or software process programs. In general they will make subqueries against relations and thereby invoke subprocesses.

The case of all attributes given is treated specially. It is used to specify the semantics of updates. These specifications are usually only useful for relations whose semantics are extensional (i.e., they actually store the tuple rather than derive it every time it is needed). There are three programs associated with this pattern: one for insertions, one for deletions, and one for modifications. In the case of insertions and deletions, the program is given the values of the attributes specifying the element to be inserted or deleted and it is the program's responsibility to modify the storage supporting the relation.

4. The Programming of Relations

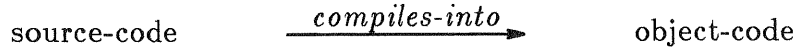
The programming of a relation in Aspen involves the specification of a process to be invoked to satisfy queries against that relation. The programming of these processes is the major task in the construction of an Aspen-based environment. The following examples illustrate the programming of Aspen relations for purposes of compiling a source-code object into an object-code object.

Most of the examples use software process programs. For purposes of characterizing these programs, we have adopted the following assumptions and conventions. The objects involved are declared and typed. The processes involved are named and may make queries against relations. The processes may take parameters, and these parameters may have types in, out, and in out; these are similar to the modes of parameters to subprograms in Ada. The flow of objects through processes is represented graphically, where "O ===> P" represents the flow of object O into process P and "P ---> O" represents the flow of object O out of process P. The expression of queries is patterned after Prolog, and is intended to be suggestive rather than prescriptive of the type of syntax that might be used in any process programming system.

Example 1: Simple compilation processes supported by Aspen

The following discussion illustrates the case in which a simple compiler is used, that is, one that does not generate intermediate objects.

Consider a relation *compiles-into* between objects of type *source-code* and objects of type *object-code.*



One query pattern for this relation is to specify a source-code object and request the related object-code object. The most natural interpretation of this query is as a request for the object-code object compiled from the given source-code object. This semantic interpretation of the query can be supported most simply by associating a compiler directly with that query pattern, so that whenever that query is made the compiler is invoked on the given source-code object to derive the object-code object. Note that in this case the process associated with the query is a traditional "tool". This situation is summarized in the following table.

English query	actual query	program
what is the object, C, of type object-code, that is related to the given object, S, of type source-code via the compiles-into relation?	compiles-into(S, C?)	compile(S: in source-code, C: out object-code)

Suppose instead that the programmers of this relation did not want to invoke a simple tool directly, but instead wished to invoke a process program which, through one or more sub-queries, would effect the compilation. In this case the queries would look just as they do above, but the process invoked would be a process program instead of a tool (and it might be designated "compile_proc" instead of "compile").

The simplest process program that would achieve this would make a single sub-query that would invoke the compiler. There may be little use for such a simple program in a real environment, but it provides a simple case for illustration. Such a program might be described as follows: An object of type *source-code* is input to compile_proc and and an object of type *object-code* is output from it.

S: source-code;
C: object-code;

S ===> compile_proc ---> C

The *compile_proc* program has a single sub-query:

program	is defined by	sub-query structure
compile_proc(S: in source-code, C: out object-code)		compile(S, C?)

Of course, the *compiles_into* relation has another query pattern of interest, which might be represented by the following query:

English query	actual query	program
what is the object, S, of type source-code, that is related to the given object, C, of type object-code via the compiles-into relation?	compiles-into(S?, C)	Lookup(S: out source-code, C: in object-code)

Implementing the lookup program requires that there be stored somewhere a "pointer" from an object code object back to the source code object (or objects) from which it was created.

Example 2: A non-monolithic compilation program supported by Aspen

A more complicated compilation program is possible (even likely) in an environment that supports the composition of tools from smaller tools and in which the intermediate objects produced by a compiler may be saved for reuse by the compiler or other tools. Consider the same binary relation, *compiles_into*, and the same query pattern *compiles_into(S,C?)*, but with more complex semantics because compilation produces a token stream from the source code, then produces a parse tree from the token stream, and then produces the object code from the parse tree. Such a "compile_proc" program may be described as follows:

S: source-code;
C: object-code;

S ===> compile_proc ---> C

L: token-stream;

P: parse-tree;

In this case the intermediate objects (L and P) are declared within compile_proc and are hidden from processes that query *compiles_into*. As described, the program *compile_proc* has three linearly ordered sub-queries, against the relations *lex*, *parse*, and *code*:

program	is defined by	sub-query structure
compile(S: in source-code, C: out object-code)		1. lex(S,L?), 2. parse(L, P?), 3. code(P,C?)

The semantics of these relations are defined by programs in a manner similar to the definitions presented above, and they are available for queries from processes other than compile_proc.

Example 3: Dynamic computation of sub-queries

In the previous examples, the effect of the program invoked in response to a particular query was predetermined. It is possible and often desirable for a process to dynamically compute the sub-queries needed to solve a given query. Continuing with the example of *compiles_into*, the program compile_proc could have a more complex sub-query structure in which it would invoke either the simple compiler (as in example 1) or the more complex compiler (as in example 2). The choice among these two approaches could be made based on the availability of (or need for) intermediate objects, the load on the processing or storage system, or other criteria. If the second choice is made, and any of the intermediate objects are available, then there may also be a choice of starting point in that sequence of sub-queries. In general, the program used to define the semantics of queries can infer the most appropriate sub-queries to make to return the desired results (as discussed in the next section). A given inferencing

program can be attached to and shared by any number of relations, as long as the relations involve types known to the program.

5. Common Relation Semantics

Many of the relations in Aspen will share common semantics. It is expected that many of these semantics will be embedded in programs and then these programs will be attached to any relation that desires to use those specific semantics. At least the following kinds of semantics will be implementable:

(1) Extensional,

(2) Backward inferencing,

(3) Forward inferencing and constraints,

Each is examined in more detail in the following sections.

5.1. Extensional Semantics

Many of the relations in Aspen operate as simple tables that store explicit connections between objects. Such relations are termed extensional. This means that the relation is stored as a set of n-tuples of pointers to related objects. Various flavors of extensional relation may be defined to accommodate various physical representations. For example, such relations might be stored as vectors or heaps, possibly with associated indexing mechanisms (btrees or hash tables). All will have a common interface that allows for the insertion, deletion, and modification of tuples, and the scanning of the relation tuple by tuple.

5.2. Backward-Inferencing Semantics

Backward inferencing is another important kind of semantics. It is invoked as a result of a query pattern in which with one or more known values and one

or more unknown values. Such queries are satisfied by first satisfying a set of sub-queries that together can be used to satisfy the original query. This type of process is often called "derivation" since the result of the original query is derived from the results of the sub-queries. The relation being queried may not exist (in the sense of having an extension), but rather, specific queries against the relation are answered dynamically. The term Backward is used because of the obvious similarity to Prolog inferencing.

The Odin system [Clemm 84] can serve to illustrate this kind of semantics. Odin allows one to record various object types and the tools (programs) that can produce objects of that type. Further, it records the object types that each tool requires as input. Each tool represents a means to derive objects of the output type from objects of the input types. For example, for each source object, there is a corresponding relocatable object derived via the C-compiler. For each relocatable, there is a corresponding executable object[5]. Odin maintains a hyper-graph whose nodes are object types and whose hyper-edges represent the tools that can derive the output from the input.

In Odin, one can make a request for a derived object such as:

test.c : exe.

This command requests the object of type *exe* (executable) that is derivable from the object *test.c*, which is of type *c*. Odin can find the given sequence of tools that must be invoked to produce the type *exe* object by examining the graph and finding a path from the type *c* to the type *exe*. Then by applying the tools specified by that path, Odin can construct the requested derived object.

In Aspen, an inferencing program similar to Odin can be associated with the query patterns of those relations that correspond to the various types of

[5]Handling multiple relocatables and libraries is obviously glossed over in this discussion.

Odin's internal graph. Queries against such relations invoke the inferencing program to actually derive the answer to the query using the internal graph and dynamically invoking the appropriate tools via queries against other relations. For example, the above Odin command would be represented by the query

Exe(test.c,e?).

This might dynamically invoke the following sequence of queries

CC(test.c,o?),Ld(o,e?)

as the means to satisfy the original query.

5.3. Forward Inferencing Semantics and Constraints

Forward inferencing is defined here as the capability to perform an action as a side effect of a change to some object. In Aspen, forward inference is handled by associating programs with the relational update operations: insert, delete, and modify. Whenever a relation is changed, some program will be invoked to decide how to interpret that change.

A major use for forward inferencing is the maintenance of constraints over relationships. A constraint program is intended to verify that some predicate holds on the current state of some set of objects. If the constraint is violated, then the constraint program must take some action to deal with the violation. Possible actions might be to report the violation or to change the state of some objects to repair the violation. It is axiomatic that if a constraint holds over some set of objects, then it can only be violated in the future by changes to the objects. Thus constraint checking can be initiated at the time that objects are changed.

A major problem for forward inferencing is to determine when to execute the associated program. It must be done on a transaction basis, which means that changes are grouped into sets of related changes. Forward inference

programs may be defined to be performed before the changes are made permanent or after they are made permanent. Checking before the changes are made is useful for verification of the correctness of changes. Checking after is useful for propagating changes to other objects and relations.

6. Higher-Level Modelling Concepts

The role of Aspen is to provide essential low-level modelling elements and principles. It is clear, however, that any useful environment must provide many higher-level capabilities, such as version control, packaging, control over the persistence and mutability of objects, configuration management, tasking, management of metadata, and others. Aspen is designed to support such features in a natural way. Limitations on space preclude extensive examination of all of these. but support for version control and packaging (especially directories) is briefly discussed below.

6.1. Version Control

Versions of objects may be assumed to form a branching tree. "Vertically" ordered objects are assumed to represent time-successive versions. "Horizontally" ordered objects are assumed to represent alternative (time-concurrent) versions. Several objects in the tree may be actively in use simultaneously.

By default a new version is assumed to be the successor of the old version from which it was implicitly created, that is, the new and old versions are related vertically in the tree. Formation of an alternate version is usually accomplished by an explicit "branching" instruction to establish a new horizontal relationship in the tree. Creation of new horizontal relationships implicitly establishes new vertical relationships as a side effect. In practice, it is often desirable to terminate branches and merge them into other branches (usually some designated main line). This merging process is very difficult to automate

and so few version control systems address it.

Historically, version control systems have taken two different approaches to the storage and management of versions of an object. In some systems, like RCS [Tichy 82], and SCCS [Rochkind 75], all version objects are physically packaged into a special version control object (for example, an RCS ",v" file). In such systems particular version objects are inserted into and extracted from the version control object by special user tools. In other systems, like Cedar [Lampson 83], each version is maintained as a separate, independent object, and the version relation is realized through a logical packaging mechanism provided by the operating system. The particular version objects may be immutable, as in Cedar, or mutable. In these systems existing versions are accessed just like other objects (without special tools), and an attempt to change a version object results in the creation of a new version object that incorporates the changes.

Aspen takes the view that version control is not primitive, and that various version control policies and mechanisms can be implemented using constraints and forward propagation. When an attempt is made to create a new version object, a version control process will be invoked to ensure that the previous version is preserved and that the new version is recorded according to the version control scheme in use. In an RCS-like system, the proper tool would be invoked to physically insert the new version into the version control object. In a Cedar-like system, the new version would have to be logically inserted into the version control relation. (Note that in the most general case version objects can also be used in the creation of new objects that are not new versions. Consequently the creation of new versions must be distinguishable from the creation of new objects in general.)

One unsolved problem with version control concerns the granularity of differences between versions. That is, how many changes must be made before

a new version is created. In Emacs, for example, it is possible to save a version every n keystrokes. For RCS, an explicit user command (check-in) controls the granularity. This issue is closely tied to the notion of transaction since the creation of versions is often naturally associated with the completion of atomic transactions.

6.2. Packaging and Directories

In any software environment, the organization of the logical "space" of objects is a central issue. It affects the naming of objects and the ways in which objects may be logically contained or "packaged" in one another. The simplest logical organization is a "flat" object space, in which all objects are contained in single-level name space and nesting of objects (such as files in directories) is precluded. Experience with the old IBM-360 operating systems shows that this is untenable. It has proven more useful to organize the object space into hierarchical "packages" or "directories" of related objects, such as trees and other forms of directed graphs (acyclic or otherwise). The Unix directory system clearly shows the utility of such an organization.

Different notions of packaging will be supported at different layers in an Aspen environment. At the lowest layers of the environment proper, immediately above Aspen (Figure 1, Layer 3), packaging will be provided by *directories*. At higher layers, additional logical and physical packaging schemes can be constructed by users.

6.2.1. Directories

Aspen *directories* are a special type of relation between names and objects. Directories define logical containment of objects (as opposed to physical containment). The type of the contained objects is unrestricted; they may include

other directories and relations. Objects (logically) contained in a directory may also participate in other relations with objects that are outside the directory. This concept is a direct generalization of a Unix directory to include objects of arbitrary types. It is also a generalization of the relations in CAIS [CAIS 85], where objects can participate in multiple relations, but only one primary relation.

Since directories are relations, they have associated processes that can interpret queries and respond to changes. These processes are responsible for interpreting names and checking constraints. User processes may utilize directories as structures to hold named objects; these processes may then retrieve objects by querying the directory.

A query against a directory relation is similar to traversing one step of a path specification in Unix. Thus it is possible, unlike Unix, to have various flavors of directory that interpret paths in different ways. For example, one kind of directory might implement an export interface facility so that not all of its objects are accessible through the directory; another kind may be used in a distributed system to retrieve objects from remote sites.

7. Implementation Issues

In order to work in a multi-user environment, Aspen will need to address a number of implementation issues. Garbage collection, concurrency control and atomic transactions are three obvious examples, although not exhaustive. Initially, and where possible, known solutions will be used and later extended as needed. Thus reference counting (with all of its flaws) may be used for garbage collection. Similarly, locking may be used for concurrency control. Atomic transactions can initially be implemented using explicit commit and abort operations. This is inconvenient for the user, it is not expected that this solution will be adequate beyond the prototyping stage. Transaction specification

will be very important in Aspen since much of the constraint semantics depends on the form of transactions that are implemented. It will also affect issues such as version control. Eventually some sort of user-controllable transaction management must be provided, but the details are as yet unspecified.

8. Comparison of Aspen to Related Systems

In the past few years there has been much research on systems that may be used for managing design processes and objects. These include CAD/CAE systems, [Katz 82, McLeod 83, Hollaar 84, Zara 85], as well as programming and software development environments [Teitelman 81, Ceri 83, Goldberg 83, Powell 83, Clemm 84, LeBlang 84, CAIS 85, Reiss 85, taylor 86]. Aspen is intended to support software development, especially through the programming of software processes as proposed by Osterweil [Osterweil 86] and the inferencing of process invocation as generalized from the Odin system [Clemm 84]. Two of the systems that have the most significance for Aspen in this regard are POSTGRES [Stonebraker 86] and Encore [Zdonik 85]. Aspen is briefly compared to each of these below.

8.1. Comparison to POSTGRES

POSTGRES is a database management system that is being developed as the successor to the INGRES relational database system [Stonebraker and Rowe 86]. Aspen shares many of the same goals as POSTGRES, but differs from it in important ways with respect to the data model, process invocation, and features provided.

Among the design goals of POSTGRES are

(1) 1. to provide better support for complex objects;

(2) 2. to provide user extendibility for data, operators, and access methods; and

(3) 3. to provide facilities for active databases (using alerters and triggers) and inferencing (both forward- and backward-chaining).

These features will be important in any software engineering environment, and Aspen has been designed to support them as well.

POSTGRES preserves the traditional relational data model used in INGRES. In this model, objects are not shared among relations and relations themselves are not a data type. Thus, in POSTGRES relations are not composable, and the type system is not closed. Aspen adopts a combination of the relational and object-oriented data models. In Aspen particular objects can participate in multiple relations. Relations themselves are typed objects and as such can participate in other relations. Thus, in Aspen the type system is closed. By including relations as objects that may be managed in the context of other relations Aspen supports "meta-programming" of the software process in addition to the traditional programming of the software product.

POSTGRES supports the programming of processes in both a query language and in general programming languages (procedures written in general programming languages are a special data type). Processes may be invoked in response to queries within the system, or in response to a specific "execute" command. Processes of the latter type may make POSTGRES queries or may operate independently of the system and act on objects not in the POSTGRES database. In Aspen, processes may be programmed in a general programming language or in a special software-process programming language (of the type proposed by Osterweil [Osterweil 86]). All processes are invoked directly or indirectly in response to queries, under the control of Aspen[6]; this ensures that

[6]Unix processes, as opposed to Aspen processes, may be invoked directly.

Aspen retains control of access to persistent objects.

One other major difference between POSTGRES and Aspen is in the range of features included. POSTGRES provides for forward- and backward-inferencing by means of triggers, alerters, and virtual columns. Aspen provides for backward inferencing through the processes that users may associate with relations to define the semantics of queries. Aspen supports forward inferencing by enabling the user to specify the conditions under which forward-inferencing processes are to be invoked (for example, in response to particular types of queries or changes to objects). POSTGRES also provides version control and history management, whereas Aspen provides mechanisms that enable users to incorporate alternative version control and history management methods at a higher level. In general, Aspen provides low-level mechanisms on top of which users may build higher-level features that are adapted to the needs of a particular project or methodology. In contrast, POSTGRES is a more complete but less flexible system in which many features are built in.

8.2. Comparison to Encore

The Encore system [Zdonik 85] provides an object-oriented database approach to programming environments. In particular, Encore provides an object-oriented database language for a database in which the objects are described by types with operations, properties, and type inheritance. Aspen shares much of the object-orientation of Encore, but provides more general facilities for dealing with inferencing and the relationships among objects.

The type system in Encore is closed in the sense that everything known to Encore is an object of some type. All objects have a type, and types themselves are included as objects. Binary relations among objects are represented as properties of the objects, and these properties are also objects; relations apart from

properties are not included as a predefined type within the system. All processes in Encore are associated with operations on objects. Aspen adopts a similar object-oriented perspective. All objects have a type, the types are themselves objects, and the type system is closed. One minor distinction is that types in Aspen are represented as abstract data types defined in terms of a domain of values and operations on those values. A more important difference is that while objects in Aspen have a state (analogous to properties in Encore), Aspen provides for n-ary relations over objects. Processes in Aspen are associated with relation objects in particular; the relations provide a structured framework for the action of processes on objects.

Encore supports backward inferencing through associative retrieval, and it supports forward inferencing through triggers that may be linked to the operations on an object. As discussed above, Aspen supports inferencing through processes that may define the semantics of queries or that may be invoked in response to user-specified conditions. Thus Aspen can thus support Encore-style inferencing and also other types of inferencing.

9. Conclusions

The programmable relation concept of Aspen seems to provide a powerful means for defining and controlling relationships over objects. This allows for a flexible and extensible object manager. A wide variety of access and storage patterns are possible, including simple stored relations, lazy evaluation, forward and backward inferencing, and constraint checking.

References

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

[Atkinson 83] Atkinson, M. P., et al, "An Approach to Persistent Programming", *Computer Journal* 26(4):360-365, 1983.

[CAIS 85] Ada Joint Program Office, U.S. Department of Defense, *Common Ada Programming Support Environment Interface Set,* Proposed MIL-STD CAIS, 1985.

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

[Clemm 84] Clemm, G. M., "ODIN - An Extensible Software Environment Report and User's Manual", University of Colorado at Boulder, Computer Science Department Technical Report CU-CS-262-84, (May 1984).

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

[Hollaar 84] Hollaar, L., Nelson, B., and Carter, T., "The Structure and Operation of a Relational Database System in a Cell-Oriented Integrated Circuit Design System", *Proceedings of the 21st Design Automation Conference* (1984), pp. 117-125,

[Katz 82] Katz, R. H., "A database Approach for Managing VLSI Design Data", *Proceedings of the 19th Design Automation Conference,* (June 1982), pp. 274-282.

[Lampson 83] Lampson, B. W., and Schmidt, E. E., "Organizing Software in a Distributed Environment", *Proceedings of the ACM Symposium on Programming Languages Issues* in Software Systems, San Francisco, (June 1983), pp. 1-13.

[Leblang 84] Leblang, D. B., and Chase, Jr., R. P., "Computer-Aided Software Engineering in a Distributed Workstation Environment", *Proceedings of the ACM Symposium on Practical Software Development Environments,* Pittsburgh, (April 1984), pp. 104-112.

[McLeod 83] McLeod, D., Narayanaswamy, K., and Bapa Rao, K. V., "An Approach to Information Management for CAD/VLSI Applications" *Proceedings of the ACM SIGMOD International Conference on Databases* for Engineering Design, San Jose (May 1983), pp. 39-50.

[Osterweil 86] Osterweil, L., "A Program-Object Centered View of Software Environment Architecture," University of Colorado, Department of Computer Science Technical Report CU-CS-332-86, (May 1986).

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

[Rathke 83] Rathke, C., and Laubsch, J. H., "OBJTALK: Eine Erweiterung von LISP zum objektorientierten Programmieren", In *Objektorientierte Software- und Hardwarearchitekturen,* Teubner Verlag, 1983, H. Stoyan and H. Wedekind, ed., pp. 60-75,

[Reiss 85] Reiss, S. P., "PECAN: Program Development Systems that Support Multiple Views" *IEEE Transactions on Software Engineering SE-ll,* 3, (March 1985), pp. 276-285.

[Rochkind 75] Rochkind, M. J., "The Source Code Control System", *IEEE Transactions on Software Engineering SE-1,* 4, (December 1975), pp. 364-370.

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

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

[Teitelman 81] Teitelman, W., and Masinter, L., "The Interlisp Programming Environment", *Computer 14,* 4 (April 1981), pp. 25-34.

[Tichy 82] Tichy, W. F. "Design, Implementation, and Evaluation of a Revision Control System," *Proceedings of the Sixth International Conference on Software Engineering,* pp. 58-67, 1982.

[Zara 85] Zara, R. V., and Henke, D. R., "Building a Layered Database for Design Automation", *Proceedings of the 22nd Design Automation Conference,* (1985) pp. 645-651.

[Zdonik 85] Zdonik, S. B., and Wegner, P., "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.