Object-Oriented Database Support for Software Environments

Scott E Hudson

Department of Computer Science
University of Arizona
Tucson, Arizona 85721

Roger King

University of Colorado
Department of Computer Science
Boulder, Colorado 80309

Abstract

Cactis is an object-oriented, multi-user DBMS developed at the University of Colorado The implementation is self-adaptive and concurrent, and runs in the Unix/C Sun workstation environment A central, unique focus of Cactis is the support of functionally-defined data in a manner which provides good performance Cactis is intended for use in applications which are conducive to an object-oriented approach and involve derived data Such applications include software environments

Cactis supports the construction of objects and type/subtype hierarchies, which are useful for managing the complex and highly-interrelated data found in software environments Such data types include programs, requirement specifications, milestone reports, configurations, documentation, and many others Cactis uses techniques based on attributed graphs to ensure that functionally-defined attributes of objects, such as compilation dependencies, cost calculations, and milestone dependencies can be maintained efficiently Since it is necessary to dynamically add new tools (such as debuggers and compilers) to a software environment, the DBMS allows the user to extend the type structure The system also supports an efficient rollback and recovery mechanism, which provides the framework for a software version facility

1 Introduction

A software environment provides a facility for managing the design, construction, testing, use, and eventual reuse of software There is currently a very active research area focussed on the development of techniques for constructing software environments One major effort is described in [Tay86] Cleary, one of the most important requirements of a software environment is that it have a central store for managing the myriad of objects which make up a software project A DBMS structures an otherwise chaotic system of files, provides a framework for specifying their interrelationships and dependencies, and for defining the precise effects of the programs which act on these files In short, a DBMS can provide a powerful abstraction, allowing a system of files to be viewed as an organized collection of objects and tools which use these objects

In this paper, we argue that conventional database technology is not sufficient for performing these tasks We take the view that an object-oriented approach is the appropriate database formalism for constructing the central store of a software environment We discuss an object-oriented DBMS called Cactis, and describe the mechanisms is uses to efficiently support software environment applications The unique contributions of Cactis are centered around its ability to effectively manage constructed objects and functionally-defined data, and perform rollback

1.1. Limitations of Current Database Technology

Current database technology, such as hierarchical, network, and relational DBMS's are limited in their data abstractions and representational power Briefly, we view the following four capabilities as being central to supporting software environments

1 The construction of recursively-defined objects and type/subtype hierarchies Software environments include complex data types such as programs, requirement specifications, milestone reports, configurations, documentation, and many others These types are often

This work was supported in part by ONR under contract number N00014 86 K-0054 and in part by NSF under grant DMC 8505164

defined in terms of each other, and need to be broken down into categories For example, a configuration is made up of a number of instances of the type program, source and object modules might be viewed as subtypes of type program

2 The definition of derived data Cactis supports derived data in the form of functionally-defined attributes of objects It uses a mechanism based on attributed graphs to ensure that functionally-defined attributes, such as compilations, cost calculations, and milestone dependencies can be maintained efficiently It is particularly important in a software environment that this capability be efficient, as a significant amount of the information in a software system is derived

3 The ability to extend the type structure This is necessary to allow users of the software environment to dynamically add new tools such as debuggers and compilers

4 An efficient rollback and recovery mechanism, which provides the framework for the recall of versions It is of particular importance that versions not be represented as largely redundant objects, as objects in a software environment are likely to be quite large In Cactis, the rollback facility is both space and time efficient, and is supported with a general-purpose Undo facility

The capability number one, above, is provided by the subsystem of Cactis, called Sembase, a tool constructed at the University of Colorado (see [FKM85, Kin84]) The other three capabilities were developed recently, and a preliminary report on these efforts appears in [HuK86a] The system is now complete and consists of approximately 65,000 lines of C code, and uses a timestamping concurrency control technique

1 2 Related Work

Recently, significant interest has developed in semantic and object-oriented database models Cactis is both a semantic and an object-oriented model It is the encorporation of both philosophies in one model that makes Cactis uniquely suited to the support of software environments

A complete discussion of semantic models and their relationship to traditional models may be found in [HuK86d, KiM85] Briefly, traditional database models support record-like structures and/or inter-record links (e g , the relational, hierarchical, and network models) Semantic models support expressive data relationships, a typical semantic model allows a designer to specify complex objects, and also supports at least one form of derived relationship, generalization (sometimes called subtyping) With generalization, one sort of object can be defined as belonging to a subcategory or subtype of a larger category of objects [CDF82, SkZ86] and [FKM85] discuss data structures and access methods used to implement semantic databases

Object-oriented models emphasize the ability of a data object to encapsulate behavior, in this way an object may respond to changes elsewhere in the database For a discussion of a number of research efforts directed at implementing object-oriented database systems, see [DiD86] Such systems vary from extensions to the relational model to handle complex data [StR86] to database implementations based on the message passing paradigm of Smalltalk [MSO86, MaS86] An object-oriented system which uses persistent programming techniques is described in [AtK83] An object-oriented implementation designed to support extensible databases is described in [CDR86], this system provides specialized storage structures and access methods for accessing large objects Another extensible system, designed for such applications as engineering, is described in [MaD86] An object-oriented system designed to support multi-media databases is described in [WKL86] There have also been some work in the area of database support for software engineering, see [ZSG86] and [Nes86]

Cactis supports the data structuring facilities of a semantic model It also supports the encapsulation capabilities of an object-oriented model by allowing an object to contain the mechanisms which allow its attributes to be derived in terms of other objects in the database Other researchers have stressed the importance of derived data in knowledge based databases [LaS84, Mor84, ShK84] Much of the previous work in this area has come from AI research oriented toward constraint based programming systems [Bor81]

During the development of the Sembase subsystem of Cactis a couple lessons were learned While this project did produce a system capable of supporting a wide class of object-oriented systems, including some forms of derived information, it fell short in two ways First of all, only a subset of first order predicate calculus expressions may be used to manage derived data Secondly, the code, while very efficient, is tricky and inelegant Cactis supports a much wider class of derived information, and does so in a clean fashion, based on a simple algorithmic model

Cactis was influenced by and extends techniques from attribute grammers (see [Knu68, Knu71]) and incremental attribute evaluation (see [DRT81, RTD83]) to provide an effective model for performing efficient maintenance of functionally-defined data and for performing rollback The effectiveness of

this last capability is quite important, as databases with complex derived data have the potential of being very difficult to rollback

In the next section, the Cactis data model is briefly described, and the implementation of Cactis is discussed Section three discusses the database requirements of software environments, and section four describes specific examples of the application of Cactis to software environments Finally, section five discusses future directions

2 Cactis

In this section, we briefly describe the Cactis data model, and discuss the algorithmic techniques used to support functionally-defined information and Undo We also briefly describe the physical implementation of Cactis

2.1. The Data Model and Attributed Graphs

A Cactis database consists of a collection of abstract objects, atomic objects (such as strings, reals, integers, booleans, arrays, and records), types, relationships, constraints, subtypes, and predicates Abstract and atomic objects constitute the "data" of the database, and abstract objects are build recursively out of atomic objects via relationships, which define the logical connections which exist between the data objects The remaining constructs comprise the database schema Objects are broken into type/subtype hierarchies based on the values of relationships and attributes (defined below), via predicates Thus, the type Persons may have a relationship called Mother, which points back to Persons, and a relationship called Cars which points to the type Automobiles A Car Buff might be defined as the subtype defined by the predicate which calculates all Persons who own more than three cars A constraint might be that all Persons must own at least one car

Also, in a Cactis database, a data item may have attributes Attributes are atomic-valued relationships, such as Age of Persons, and may be of any C data type, except pointer This aspect of Cactis extends techniques derived from Knuth's attribute grammars [Knu68, Knu71] as well as from more recent work on incremental attribute evaluation [DRT81, RTD83] used in syntax directed editors These techniques have been used extensively in compiler construction to represent the semantics of programming language text Thus, in a Cactis database, the relationship structure between objects is viewed as an *attributed graph* Each node in the graph is an instance of a particular named type of data

The Cactis model allows *attribute evaluation rules* to be attached to certain attributes These rules allow attributes to be derived from other attributes within a given instance and from the values contained in related instances Thus, objects may be active in responding to changes in their environment rather than simply passively storing data Since attribute evaluation rules can be constructed from arbitrary functions of attributes, it is possible to model and manipulate the complicated semantics that real world entities often possess It should be noted that, in an attributed graph, the attributes of a given instance may be derived only in terms of attribute values passed to it from instances the given instance is directly related to via named relationships However, attribute values may be passed transitively from instance to instance Thus, if the data instance A is related to instance B and instance B is related to instance C, A's attributes may derived in terms of C's attribute values

There are two kinds of attributes in the attributed graph, derived and intrinsic Derived attributes have an attribution rule attached to them, while intrinsic attributes do not. This means that only intrinsic attributes may be given new values directly Derived attributes are only changed indirectly by computations resulting from changes to intrinsic attributes

An additional property of the Cactis data model is the ability to attach constraints to attributes In the data model, a constraint is implemented as a derived attribute value which computes a boolean value indicating whether the constraint has been violated The attribute evaluation rule in this case is simply the predicate defining the constraint Whenever an attribute which is designated as testing a constraint evaluates to false, rollback of the current transaction is performed Since constraint predicates are handled in the same manner as normal derived attribute values, the constraint predicate may be formed using any expression which returns a boolean value

2.2. Efficiency Considerations

A number of data models have made provisions for functionally derived data However, the actual implementations of most of these systems use techniques equivalent to triggers [BuC79] attached to data While this method is adequate for sparsely interconnected data, it can present problems for more highly interconnected data Since there is no restriction on the kinds of actions performed by triggers, the order of their firing can change their overall effect While this allows triggers to be extremely flexible, it can also become very difficult to keep track of the interrelationships between triggers Hence, it is easy for errors involving unforeseen interrelationships to occur, and much more difficult to predict the behavior of the system

under unexpected circumstances

By contrast, the effects of attribute evaluation computations used in the Cactis system are much easier to isolate and understand Each data type in the system can be understood in terms of the relationship and attribute values it stores, the values it transmits out across relationships, and the values it receives across relationships This allows the schema to be designed in a structured fashion and brings with it many of the advantages of modern structured programming techniques The ability to localize behavior in this fashion is particularly important in extensible systems, where end users are expanding the system during a session

Even if we can adequately deal with the unconstrained and unstructured nature of triggers, they can also be highly inefficient If we choose a naive ordering for recomputing data values after a change, we may waste a great deal of work by computing the same data values several times For example, a simple trigger mechanism might work recursively, invoking new triggers as soon as data changes Any trigger mechanism which uses a fixed ordering of some sort (e g depth first or breadth first) can needlessly recompute some values, in fact, in the worst case can recompute an exponential number of values On the other hand, the attribute evaluation technique used in the Cactis system will not evaluate any attribute that is not actually needed, and will not evaluate any given attribute more than once

The Cactis primitives include operations for creating and deleting object type instances, establishing and breaking relationships between instances, defining predicates and subtypes, and primitives for retrieving and replacing attribute values These primitive actions are augmented by the meta-action *Undo* Undo has the effect of forcing the rollback of one transaction This meta-action allows the user to freely explore the database, knowing that no actions need have permanent effect

Whenever changes are made to a database using one of the primitive data manipulation actions, Cactis must ensure that all observable attribute values in the database retain a value which is consistent with the attribute rules of the system This requires some sort of attribute evaluation strategy or algorithm One approach would be to recompute all attribute values every time a change is made to any part of the system This is clearly too expensive What is needed is an algorithm for incremental attribute evaluation, which computes only those attributes whose values change as a result of a given database modification This problem also arises in the area of syntax directed editing systems, so it is not surprising that algorithms exist to solve this problem for the attribute grammars used in that application The most successful of these algorithms is due to Reps [Rep82] Reps' algorithm is optimal in the sense that only attributes whose values actually change are recomputed

Unfortunately, Reps' algorithm, while optimal for attributed trees, does not extend directly to the arbitrary graphs used by Cactis Instead, a new incremental attribute evaluation algorithm has been designed for Cactis This new algorithm exhibits performance which is similar to Reps' algorithm, but does have a slightly inferior worst case upper bound on the amount of overhead incurred

The algorithm works by using a strategy which first determines what work has to be done, then performs the actual computations The algorithm uses the *dependencies* between attributes An attribute is *dependent* on another attribute if that attribute is mentioned in its attribute evaluation rule (i e is needed to compute the derived value of that attribute) When the value of an intrinsic attribute is changed, it may cause the attributes which depend on it to become out of date with respect to their defining attribute evaluation rules Instead of immediately recomputing these values, we simply mark them as *out of date* We then find all attributes which are dependent on these newly *out of date* attributes, and mark them *out of date* as well

This process continues until we have marked all affected attributes During this process of marking, we determine if each marked attribute is *important* Attributes are said to be *important* if they have a constraint predicate attached to them, or if the user has asked the database to retrieve their values When we have completed marking attributes during the first phase of the algorithm, we will have obtained a list of attributes which are both *out of date* and *important* We can then use a demand driven algorithm to evaluate these attributes in a simple recursive manner The calculation of attribute values which are not *important* may be deferred, as they have no immediate affect on the database If the user explicitly requests the value of attributes (i e makes a query) they become *important*, and new computations of *out of date* attributes may be invoked in order to obtain correct values An implementation which is similar, in that it uses lazy evaluation, is described in [BFN82]

In the worst case, the overhead of the algorithm (when amortized over the sequence of all possible transactions) is

$$O(|\textbf{Nodes(Could\_Change(A))}| + |\textbf{Edges(Could\_Change(A))}|),$$

where **Could_Change** describes a dependency graph It represents all attributes and objects reachable from the site of the original primitive change A, via some series of dependencies Edges model relationships and nodes model objects This behavior comes from the mark out of date phase of the algorithm

However, this is the worst case behavior In many real cases this traversal will be cut short by finding attributes which are already out of date For example if an attribute A were assigned 2 different values in a row before updating the system, the second assignment would only update A and not visit any other attributes and hence incur only O(1) overhead In general the actual performance of the Cactis attribute evaluation algorithm will depend on the particular attributes involved, particularly on whether some attributes may remain as out of date for long periods of time if they are not important and are not accessed Also, if a given attribute is changed as a result of two different primitive updates to intrinsic attributes, the given attribute will only be reevaluated once (unless of course, the given attribute has been accessed before the second primitive update is performed)

In order to support the primitives which break and establish relationships, a process similar to that used for intrinsic attribute changes is used When a relationship is broken, the system determines which derived attributes depend on values that are passed across the relationship These attributes are marked out of date just as if an intrinsic attribute had changed When a relationship is established, the second half of the attribute evaluation algorithm is invoked to evaluate attributes which are out of date and important In order to ensure that derived attributes can always be given a valid value, the database ensures that relationships are not left dangling across attribute evaluations This is either done explicitly by the transaction, or where necessary the system will provide special dummy instances to tie off any dangling relationships Also, the primitive to delete an instance can be treated the same as breaking all relationships to the instance, and the primitive to create a instance does not affect attribute evaluation until relationships are established

During the evaluation of attributes, certain attributes will have constraint predicates attached to them After an attribute is evaluated, this constraint predicate is tested If it evaluates false, a constraint violation exists By default, this causes the transaction invoking the evaluation to fail and be rolled back or undone Optionally, a special recovery action associated with the constraint can be invoked to attempt to recover from the violation In either case, the constraint must be satisfied or the transaction invoking the evaluation will fail and be undone

With respect to supporting undo, we note that all of the actions that take place as a consequence of changing an attribute value can be undone simply by restoring the old value of the attribute Updates resulting from structural changes can be undone by restoring the old structure Thus, Undo may be performed with the same algorithmic techniques used to support attribute evaluation

2 3. The Implementation of Cactis

We now examine the methods that Cactis uses to perform attribute reevaluation efficiently, given that the system is a mass storage database, not an in-memory system Specifically, the techniques we have outlined above are efficient in terms of the number of attributes that they recompute when changes are made However, they are not necessarily efficient in terms of the number of disk accesses needed Therefore, Cactis is very careful in selecting the order of computations when carrying out the actions of a database primitive

If we examine the routines which mark attributes as out of date and reevaluate attribute values, we see that they are each just a traversal of part of the attribute dependency graph We may in fact choose any traversal order which visits the same attributes In particular, we are free to choose an order which reduces the number of disk accesses required

In the implementation of the Cactis data model, we use an order of traversal which is chosen dynamically The way we choose this order is to use a concurrent system in which a number of sub-traversals are (conceptually) running at the same time Each time we reach a node which has two or more descendents to traverse, we fork a sub-traversal process to traverse the graph in each direction For example, when we mark an attribute out of date, we then schedule a traversal process for each of the attributes which depend on it When we evaluate an attribute, we request all the values needed to recompute its value in parallel We can think of this as a parallel traversal of the graph where each branch of the traversal runs independently To optimize disk access we use a greedy technique Of all the sub-traversal processes which are runnable at any given time we choose to execute the one which we expect to perform the least number of disk accesses

In practice we do not create actual separate processes to accomplish our parallel traversal but instead simulate multiple processes in a single process We break all computations into pieces or *chunks* to be scheduled independently For example, a normal attribute evaluation rule is implemented using two chunks The first schedules an evaluation for each of the attri-

bute values it depends on, then makes arrangements to schedule the second chunk when all the values are available The second chunk, which is scheduled only after all the values it needs have been computed, executes the attribute evaluation rule in order to compute the final value for the attribute It then stores the value and informs any process waiting for the value that it is now available

The scheme for implementing concurrency that we have described is very simple, easy to implement, and is quite efficient The technique we use is similar to that used in the OWL real-time concurrent programming language For additional information about implementation details, expected performance, translation of programs into chunks and experience with the OWL language see [Don83]

Once we have introduced concurrency to the system as outlined above, the process of choosing a good traversal order simplifies to a scheduling problem We choose a process to run which we expect to perform the least number of disk accesses The obvious choice for this process is one which can be processed using attributes currently in memory Note that each process is associated with one attribute, the one it is computing or marking out of date It may need other attribute values to compute its own value, but these are the responsibility of other processes Any needed values will have been collected in storage attached to the process before it is scheduled as runnable

We use a simple hashing scheme to index all pending processes by the instances that contain the attribute that they are associated with Whenever a disk block is read into memory, all processes which are associated with some instance stored on that block are promoted to a special very high priority queue When new processes are scheduled, we first check to see if the instance associated with the process is already in memory, if so we schedule the request on the high priority queue Since they can be executed without additional disk access, processes on the high priority queue always have priority over other processes

In order to improve the locality of data references, we cluster data in the Cactis model on the basis of usage patterns We keep a count of the total number of times each instance in the database is accessed, as well as the number of times we cross a relationship between instances in the process of attribute evaluation or marking out of date We will then periodically reorganize the database on the basis of this information In particular we will pack the database into blocks using the following greedy algorithm

Repeat
  Choose the most referenced instance in the database that has not yet been assigned a block
  Place this instance in a new block
  Repeat
    Choose the relationship belonging to some instance assigned to the block such that
      (1) The relationship is connected to an unassigned instance outside the block and,
      (2) The total usage count for the relationship is the highest
    Assign the instance attached to this relationship to the block
  Until the block is full
Until all instances are assigned blocks

This algorithm attempts to place instances which are frequently referenced together, in the same block This will tighten the locality of reference for the database

When scheduling processes, once all *in memory* processes have been executed we must choose the next one to execute We would like to choose the process which will cause the least number of disk accesses, however, we cannot know in advance which process this will be What we do instead, is use past behavior, or in the case of marking out of date, a worst case estimate, as a predictor of future behavior We keep information about past behavior in the form of a decaying average which changes over time This makes the database self-adaptive, allowing changes in the structure of the database to be reflected in changing averages and hence changing scheduling priorities

In the Cactis data model values flow across relationships in order to communicate information from one instance to another In order to provide statistics for self adaptive optimization of the attribute evaluation process, we tag each relationship with a decaying average of the number of instances visited (or alternately the actual amount of disk I/O incurred) when the value transmitted across the relationship was requested in the past We use these tags to assign a priority to pending processes in the scheduling queue The highest priority is given to the process with lowest expected disk I/O Processes which request values local to an instance rather than across a relationship are not of concern since they will be scheduled as high priority when the instance is brought into memory A special priority is given to processes which are the direct user requests that start a chain of computations

In the case of evaluating an attribute, we update statistics when we return to the attribute in order to store its new value However, in the case of marking out of date, we do not return and hence cannot store an updated statistic In this case we use

an alternate worst case statistic computed when clustering was last performed This statistic tells how many disk blocks will be visited in the worst case (i e assuming that no attributes to be visited are already marked out of date) A similar worst case statistic is used as an initial estimate for the dynamically changing decaying averages

To summarize our strategy for performing updates, we treat the traversals needed to implement attribute evaluation as a concurrent computation This allows us to dynamically choose a traversal order that reduces disk access In this framework, the choice of a traversal order simplifies to the choice of a scheduling order Sub-traversal processes which can be executed without disk access are given highest scheduling priority Once all computations that can be performed on in memory data have been completed we choose processes which have the smallest expected number of disk accesses to run first Expected disk accesses are measured by either using self adaptive past performance statistics in the form of a decaying average, or on the basis of worst case statistics gathered at cluster time

3 Issues Concerning the Application of Cactis to Software Environments

Software environments are an example of an application domain that is not well supported by traditional database systems A software environment serves as a means for managing the design, understanding, use, and reuse of software Cactis was constructed with such applications as CAD/CAM, PCB design, VLSI design, and in particular, software environment support in mind The idea was to provide a database tool that would serve as the central repository of an environment, and to allow the sorts of derived information needed in an environment to be specified with as little additional code as possible We are currently in process of implementing various parts of an environment with Cactis, in this section we describe these efforts

Below, we will first look at the specific requirements of software environments which do not seem to be met by traditional DBMS's, then see how each requirement is supported by Cactis Then in the next section we will consider several specific tasks in detail to give a broader view of the use of Cactis in a software environment

Software environments typically deal with highly structured and interrelated objects A primary example of this is of course computer programs, but software environments may also wish to deal with objects involving the management and control of an overall software development project The sorts of object generally included in descriptions of existing and proposed environments, such as [CKT86, Pen86], include software components and software dependencies, versions, documentations, requirements, milestone reports, test data, verification results, bug reports, etc Note that "software components" which are themselves highly structured and interrelated entities are only one element of this list Because of the complexity of the interrelationships defined in this model, it is essential that the consistency of the database is maintained automatically or semiautomatically Without some form of automatic support it is very likely that inconsistent data will be entered into the database

The kind of highly structured and interrelated data used by a software environment is precisely the kind of data that the Cactis data model is designed to handle In Cactis, relationships can be named and typed, and objects may be built using arbitrarily typed internal attribute values Furthermore, composite objects may be built using relationships representing containment This allows us to model objects such as computer programs which may use a complex recursively defined structure, in the same framework as simpler objects such as problem reports or bug fixes which refer to parts of programs

In addition to handling complex, highly interrelated data, another primary requirement of a database to support environments is extensibility [Cle84, WaP86] We would like to be able to extend the software environment with new tools to meet new or specialized needs This ability to create new tools which work in harmony with the existing system is, for example, one of the major strong points of the overall Unix programming environment To support extensibility, a database must be able to add new types of data and refine or modify existing types The object oriented subtyping structure of the Cactis data model is a good base for supporting these kinds of dynamic extensions

Finally, software environments, unlike most applications, deal with entities which change dramatically over time The ability to retrieve and manipulate multiple versions of programming entities can be crucial to the programming process In addition, we need the ability to manipulate versions and version streams as objects in themselves in order to support configuration management tools within the system All this must be done efficiently (see [ReG86]) We can again see that the formalism of functionally-defined data helps us accomplish this As we did in our discussion of Undo, we can note that all of the actions that take place as a consequence of changing an attribute value can be undone simply by restoring the old value

of the attribute Similarly, updates resulting from structural changes can be undone by restoring the old structure This means that although we may derive wide ranging effects from small changes to data, we need only remember the small changes made in order to restore the database to its old status

This gives us an efficient *delta* mechanism which allows us to recover old versions from the current one In particular, the information needed to remember a delta is proportional in size to the initial changes made to the database rather then the total change in the database which may result because of derived data Because we can support data of arbitrary types as objects in the Cactis model it is easy to create objects which represent the edit operations that make up a delta Since these deltas are normal objects they can be attached to other objects such as change descriptions, and in general can be integrated with the rest of the database

To reiterate the requirements we have stated, a database which supports a software environment should

- support highly structured and interrelated data
- provide automatic or semi-automatic mechanisms for maintaining consistency within interrelated data
- allow addition of new types of data within the framework of the exiting system and allow refinement or extension of the types of existing data to meet new needs
- support retention, recall, and management of multiple related versions of objects

While this list is not exhaustive, it is representative of some requirements found in the literature [Ost86] and is indicative of current research directions [BeE86, GMT86, WiA86] Cactis has been designed to meet these needs in an efficient fashion

```
Object Class milestone is

  Relationships
   depends_on        milestone_dep Multi Socket,
   consists_of       milestone_dep Multi Plug,


  Attributes
   sched_compl       time,      /* originally scheduled completion time     */
   local_work        timef,     /* time to complete milestone alone */
   exp_compl         time,      /* expected completion time         */
   late        boolean,         /* is this milestone expected late  */

  Rules
    exp_compl =
      Begin
        latest  time,

        /* sum local work and latest of things depended on */
        latest  = TIME0,
        For Each dep Related To depends_on Do
          latest  = later_of(latest,dep exp_time),
        End,
          return(latest + local_time),
      End,

    late  = later_than(exp_compl, sched_compl),

    consists_of exp_time  = exp_compl,
End,
```

Figure 1 Class Definition for Milestone Objects

4 Specific Applications of Cactis to Software Environments

As described above, software environments typically deal with data that is interrelated in such a way that changing one piece of data can have effects on many other data items It is important that the database be able to maintain consistency in this situation In this section, we describe concrete applications of Cactis to derived information in a software environment We describe the construction of a milestone manager, which is currently under construction, and a make facility, which has been completed

The data type "milestone" within an environment typically models the scheduled and expected completion times of a software component One milestone may depend on another, and changing the expected completion date for one milestone may have effects that ripple throughout the expected completion dates for other milestones in the system Changing a milestone is an instance of a simple modification which affects the consistency of the database If the expected completion date of a milestone is changed without also updating all the milestones that directly or indirectly depend on it, the database will be inconsistent and incorrect

In the Cactis data model we may include a rule which defines expected completion dates as a function of the milestones that they depend upon We may also compute an attribute that indicates whether the current expected completion date of a milestone is after its originally scheduled completion date, hence is expected to be late Figure 1 shows how an object class for milestones implementing these attributes has been constructed

Using its incremental attribute evaluation algorithm and appropriate attribute evaluation rules as outlined above, the system is able to efficiently ensure that all milestones always have consistent values Under the Cactis data model, ensuring the integrity of the database is no longer dependent on the entire collection that tools that operates on it, but can be handled in a centralized way by the database itself This increases confidence in the correctness of the database and simplifies the construction of individual tools that will use it or operate upon it In addition, new tests and constraints can be added to the database without modifying existing tools

Turning to the issue of extensibility, we have already indicated that automatic propagation of changes based on functionally-defined data allows new tests and constraints to be added to a database without disturbing the operation of existing tools As an example, we can add a "very_late" attribute to a milestone which indicates if the milestone's expected completion date exceeds its scheduled completion date by more than a fixed limit However, since the database itself is responsible for change propagation, existing tools which indirectly modify the expected completion date of milestones would not be affected at all by this new attribute Consequently we can add new functionality without having to modify existing tools

It is useful to note that, because of the subtyping mechanism of the Cactis model, it is possible to use values such as the very_late attribute above to the change subtype membership of an object dynamically Thus we can add new attributes and hence new functionality to particular objects dynamically based on their properties - again without disturbing existing tools Because of these properties the Cactis model is well suited to applications such as software environments where extensibility is a key issue

One unique feature of using the Cactis data model to support environments is its ability to represent the entire range of data within a system This can include data ranging from syntactic elements within programs, to module interconnections, to scheduling data, all the way up to facts about the personnel involved in a project All these forms of data can all be supported in a single unified framework In particular, since the Cactis model includes an attribute evaluation capability that was inspired by the work on syntax directed editors and incremental attribute evaluation [DRT81, Rep84, TeR81] we can support a whole range of capabilities for dealing with programs based on attribute grammars Examples of existing environments and environment generators based on attribute grammar formalisms include the Cornell Program Synthesizer Generator [ReT85], the Poe system [FJM84], and the SAGA system [CaK85] among others These systems provide a number of program development aids implemented as attribute computations Among the more powerful of the repertoire of techniques available is program flow analysis Program flow analysis can provide important information for testing, analysis and optimization of programs [FoO76, Ost81] While we will not discuss them specifically here, techniques for flow analysis based on attribute evaluation are described in detail in [BaJ78, Far77, Wil81] Briefly, since Cactis does not support data cycles, it can only handle flow analysis for simple languages such as a goto-less Pascal, however, the techniques described in [Far86] are being incorporated into Cactis so that it may support more general forms of flow analysis

In addition to tasks that operate at a level internal to programs, a software environment may need to work with programs as a unit A good example of such a capability is the

```
Object Class make_rule is

 Relationships
  output          make_result Multi Plug, /* to things that depend on this object */
  depends_on      make_result Multi Socket, /* to things this object depends on  */

 Attributes
  file_name       string, /* path name of file to create      */
  make_command    string, /* text of command to create the file */

 Rules

End Object,
```

Figure 2 Class Definition for Make_Rule Objects

```
output mod_time  = /* compute and return the youngest of things this object depends on */
      Begin
         youngest  time_val,

         youngest  = file_mod_time(file_name),
         For Each dep Related To depends_on Do
            youngest  = later_of(youngest,dep mod_time),
         End,

         return(youngest),
      End,
```

Figure 3 Evaluation Rule for Make_Rule Object

```
output up_to_date  =
      Begin
         need_recreate  boolean,
         this_time   time_val,

         need_recreate  = False,
         this_time  = file_mod_time(file_name),

         /* loop over all things this object depends on */
         For Each dep Related To depends_on Do
            /* make sure thing depended on is up to date */
            VOID(dep up_to_date),

            /* is this object out of date */
            If later_than(dep mod_time,this_time) Then need_recreate  = True,
         End For,

         /* recreate this object if necessary */
         If need_recreate Then system_command(make_command),

         return(1),
      End,
```

Figure 4 Evaluation Rule for Make_Rule Object

ability to control recompilation of programs source code based on last modification times and mutual dependencies The idea is to use dependencies and modification times to determine exactly those modules or files which could need recompilation and to automatically issue the commands necessary to do those recompilations This capability is provided by the Make program [Fel79] found in the Unix system as well as in other tools [Cle84] Because the Cactis data model can support arbitrary types, it is possible to supply this sort of capability within a Cactis database

While the Cactis model cannot directly handle the files that usually constitute source, object, and executable programs, it can deal with them indirectly In particular, it can represent a file stored in a normal file system simply by its name The file name can be used within Cactis objects like the one declared in Figures 2, 3, and 4 to implement a *make* capability Figure 2 gives the main body of an object class representing a dependency rule We have declared two relationships which relate this object to the things it depends on (via *depends_on*) and things that depend on it (via *output*) Because we need a many to many relationship here, there is an auxiliary object class not shown that is used to connect output relationships to depends_on relationships We have also declared two attributes, one of which represents the file name of the file we wish to create The other is the text of a command which, when executed by the operating system, will create the file Figures 3 and 4 give attribute evaluation rules for computing values transmitted across the output relationship of the object

Figure 3 shows an evaluation rule which computes the earliest modification time among the local object to be created itself and all the things it depends on We assume the routine *file_mod_time* returns the last modification time of the named file, or a time in the distant future if the file does not exist Figure 4 shows an evaluation rule which, when executed, will ensure that the object of interest as well as all the objects it depends on are up to date It begins by asking for and discarding the *up_to_date* value for each object it depends on This will ensure that all recursively dependent objects are recreated as needed and in an appropriate order Next, if the modification time of some object depended upon is later than the modification time of the object of interest it issues the command to recreate the object

What we have designed here is a very simple make capability Using the subtyping capabilities of the Cactis model, we have built on this simple capability For example, we can create a make_rule which insists that the object it maintained was always kept in an up to date state, or better, one which forces an object to be constantly up to date if a certain boolean attribute were true, or acted normally otherwise The ability to make such small changes and improvements can make creating new tools much easier, particularly when existing functionality can be left undisturbed as in the Cactis model

As a final example of how the Cactis model can support software environments we note that Cactis attributed graphs can be used to manage the user interface by making use of a new graphical presentation system created by the authors The basic idea behind this approach involves constructing and composing special program fragments that, when combined, are able to redraw a graphical display screen Attribute evaluation rules are used to create, combine and control these program fragments in order to manage a user interface This allows the user interface to automatically reflect the state of the underlying data regardless of how it is modified For a full explanation of how this system works see [HuK86b, HuK86c]

5. Directions

We are in the process of constructing a distributed version of Cactis, with the effort just getting under way As modern software environments will most likely be used in distributed workstation applications, this facility is viewed as crucial It will be necessary to allow different users at different machines to configure their own environments privately and share information Cactis is well-suited this task, as it allows the end user to conveniently tailor a local database Also, the concurrent implementation of Cactis is naturally suited to a parallel or distributed system In this way, various sub-traversals may actually be running at the same time

Acknowledgements

The authors would like to thank the team that constructed the software described in this paper Pam Drew, Shehab Gamalel-din, Janet Jacobs, Deacon Lancaster, Carla Mowers, Loraine Neuberger, Evan Patten, Don Ravenscroft, Tom Rebman, Kurt Rivard, Jerry Thomas, and Gary Vanderlinden In particular, Pam Drew and Tom Rebman deserve much credit for constructing Cactis, and Jerry Thomas constructed the data language processor

References

[AtK83] M P Atkinson and K G Kulkarni, "Experimenting with the Functional Data Model", *Technical Report on Persistent Programming, University of Edinburgh 5* (Semptember, 1983)

[BaJ78] W A Babich and M Jazayeri, "The Method of Attributes for Data FLow Analysis Part I Exhaustive Analysis Part II Demand Analysis", *Acta Informatica 10* (October 1978), 245-272

[BeE86] N Belkhatir and J Estublier, "Experience with a Database of Programs", *Proceedings of the Second Symposium on Practical Software Environments*,, December 1986

[Bor81] A Borning, "The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory", *ACM Transactions on Programming Languages and Systems 3* (October 1981), 353-387

[BuC79] O P Buneman and E K Clemons, "Efficiently Monitoring Relational Databases", *Trans Database Systems 4* (Sept 1979), 368-382

[BFN82] P Buneman, R E Frankel and R Nikhil, "An Implementation Technique for Database Query Languages", *ACM Transactions on Database Systems 7* (June 1982), 164-186

[CaK85] R H Campbell and P A Kirslis, "The SAGA Project A System for Software Development", *Proceedings of Symposium on Practical Software Development Environments*, Pittsburgh, 1985, 73-80

[CDR86] M J Carey, D J DeWitt, J E Richardson and E J Shekita, "Object and File Management in the EXODUS Extensible Database System", *Proceedings of the Twelfth International Conference on Very Large Databases*, August, 1986, 91-100

[CDF82] A Chan, S Danberg, S Fox, W Lin, A Nori and D Ries, "Storage and Acces Structures to Support a Semantic Data Model", *Proceedings of the Eight International Conference on Very Large Databases*, September 8-10, 1982

[Cle84] G M Clemm, "Odin An Extensible Software Environment, Report and Users Reference Manual", *University of Colorado at Boulder Technical Report 262-84*, March, 1984

[CKT86] K Cooper, K Kennedy and L Torczon, "The Impact of Interprocedure Analysis and Optimization in the Rn Programming Environment", *ACM Transactions on Programming Languages and Systems*, October 1986, 491-523

[DRT81] A Demers, T Reps and T Teitelbaum, "Incremental Evaluation for Attribute Grammars with Application to Syntax Directed Editors", *Conference Record of the 8th Annual ACM Symposium on Principles of Programming Languages*, Jan 1981, 105-116

[DiD86] K Dittrich and U Dayal, *International Workshop on Object-Oriented Databases*, Pacific Grove, California, September 23-26, 1986

[Don83] M D Donner, "The Design of OWL A Language for Walking", *SIGPLAN Notices 18* (June 1983), 158-165

[FKM85] D Farmer, R King and D Myers, "The Semantic Database Constructor", *IEEE Transactions on Software Engineering SE-11* (July 1985), 583-590

[Far77] R W Farrow, "Attributed Grammar Models for Data Flow Analysis", *PhD Dissertation Rice University*, May 1977

[Far86] R Farrow, "Automatic Generation of Fixed-Point-Finding Evaluators for Circular, But Well-Defined Attribute Grammars", *SIGPLAN Notices Notices 21* (July, 1986), 85-98

[Fel79] S I Feldman, "Make -- A Program for Maintaining Computer Programs", *Software - Practice and Experience 9* (April 1979), 255-265

[FJM84] C N Fischer, G F Johnson, J Mauney, A Pal and D L Stock, "The Poe Language-Based Editor Project", *Proceedings of Symposium on Practical Software Development Environments*, Pittsburgh, April 1984, 21-29

[FoO76] L D Fosdick and L J Osterweil, "Data Flow Analysis in Software Reliability", *ACM Computing Surveys 8* (September 1976), 305-330

[GMT86] F Gallo, R Minot and I Thomas, "The Object Management System of PCTE as a Software Engineering Database Management System", *Proceedings of the Second Symposium on Practical Software Environments*, December 1986

[HuK86a] S Hudson and R King, "CACTIS A Database System for Specifying Functionally-Defined Data", *Proceedings of the Workshop on Object-Oriented Databases*, Pacific Grove, California, September 23-26, 1986, 26-37

[HuK86b] S E Hudson and R King, "Implementing a User Interface as a System of Attributes", *Proceedings of the Second Symposium on Practical Software Environments*, December 1986

[HuK86c] S E Hudson and R King, "Semantic Feedback in the Higgens UIMS", *submitted to IEEE Transactions on Software Engineering*, 1986

[HuK86d] R Hull and R King, "Semantic Database Modeling Survey, Applications, and Research Issues", *USC Technical Report Tech Rep -86-201* (April 1986)

[Kin84] R King, "Sembase A Semantic DBMS", *Proceedings of 1st Int'l Workshop on Expert Database Systems*, Kiawah Island, South Carolina, Oct 1984, 151-171

[KiM85] R King and D McLeod, "Semantic Database Models", in *Database Design*, S B Yao (editor), Prentice Hall, 1985

[Knu68] D E Knuth, "Semantics of Context-Free Languages", *Math Systems Theory J 2* (June 1968), 127-145

[Knu71] D E Knuth, "Semantics of Context-Free Languages Correction", *Math Systems Theory J 5* (Mar 1971), 95-96

[LaS84] G M E Lafue and R G Smith, "Implementation Of A Semantic Integrity Manager With A Knowledge Representation System", *Proc First International Worksohop on Expert Database Systems*, Kiawah Island, South Carolina, Oct 24-27, 1984, 172-185

[MSO86] D Maier, J Stein, A Otis and A Purdy, "Development of an Object-Oriented DBMS", *Proceedings of the conference on Object-Oriented Programming Systems, Languages, and Applications*, September 29-October 2, 1986, 472-482

[MaS86] D Maier and J Stein,, "Indexing in an Object-Oriented DBMS", *Proceedings of the First International Workshop on Object-Oriented Database*, Pacific Grove, California, September 23-26, 1986, 171-182

[MaD86] F Manola and U Dayal, " "PDM An Object-Oriented Data Model"", *Proceedings of the Workshop on Object-Oriented Databases*, Pacific Grove, California, September 23-26, 1986, 18-25

[Mor84] M Morgenstern, "The Role of Constraints in Databases, Expert Systems, and Knowledge Representation", *Proc First International Worksohop on Expert Database Systems*, Kiawah Island, South Carolina, Oct 24-27, 1984, 207-223

[Nes86] J R Nestor, "Recreation and Evolution in a Programming Environment", *Proceedings of the Workshop on Object-Oriented Databases*, Pacific Grove, California, September 23-26, 1986, 230

[Ost81] L Osterweil, "Using Data Flow Tools in Software Engineering", in *Program Flow Analysis Theory and Applications*, S S Muchnick and N D Jones (editor), Prentice-Hall, Englewood, NJ, 1981, 237-263

[Ost86] L Osterweil, "A Process-Object Centered View of Software Environment Architecture", *University of Colorado at Boulder Technical Report 332-86*, May, 1986

[Pen86] M H Penedo, "Prototyping a Project Master Data Base for Software Engineering Environments", *Proceedings of the Second Symposium on Practical Software Environments*, December 1986

[ReG86] J J Reppy and E R Gansner, "A Foundation for Programming Environments", *Proceedings of the Second Symposium on Practical Software Environments*, December 1986

[Rep82] T Reps, "Optimal-time Incremental Semantic Analysis for Syntax-directed Editors", *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages*, Jan 1982, 169-176

[RTD83] T Reps, T Teitelbaum and A Demers, "Incremental Context-Dependent Analysis for Language-Based Editors", *Trans Prog Lang and Systems 5* (July 1983), 449-477

[Rep84] T W Reps, *Generating Language-Based Environments*, MIT Press, Cambridge, Mass , 1984

[ReT85] T Reps and T Teitelbaum, "The Synthesizer Generator", *Proceedings of Symposium on Practical Software Development Environments*, Pittsburgh, 1985, 42-48

[ShK84] A Shepherd and L Kerschberg, "Constraint Management in Expert Database Systems", *Proc First International Worksohop on Expert Database Systems*, Kiawah Island, South Carolina, Oct 24-27, 1984, 522-546

[SkZ86] A H Skarra and S B Zdonik, "The Management of Changing Types in an Object-Oriented Database", *Proceedings of the conference on Object-Oriented Programming Systems, Languages, and Applications*, September 29-October 2, 1986, 483-495

[StR86] M Stonebraker and L A Rowe, "The Design of Postgres", *Proceedings of International Conference on the Management of Data*, May, 1986, pages 340-355

[Tay86] R Taylor, "Arcadia A Software Development Environment Research Project", *University of California at Irvine, Dept of Information and Computer Scinece, Technical Report*, April 1986

[TeR81] T Teitelbaum and T Reps, "The Cornell Program Synthesizer A Syntax-directed Programming Environment", *Comm of the ACM 24* (Sept 81), 563-573

[WaP86] A I Wasserman and P A Pircher, "A Graphical, Extensible Integrated Environment for Software Development", *Proceedings of the Second Symposium on Practical Software Environments*, December 1986

[WiA86] D S Wile and D G Allard, "Worlds an Organizing Structure for Object-Bases", *Proceedings of the Second Symposium on Practical Software Environments*, December 1986

[Wil81] R Wilhelm, "Global Flow Analysis and Optimization in the MUG2 Compiler Generating System", in *Program Flow Analysis Theory and Applications*, S S Muchnick and N D Jones (editor), Prentice-Hall, Englewood, NJ, 1981 132-159

[WKL86] D Woelk, W Kim and W Luther, "An Object-Oriented Approach to Multimedia Databases", *proceedings of ACM SIGMOD conference*, May, 1986, 311-325

[ZSG86] C Zaroliagis, P Soupos, S Goutas and D Christodoulakis, "The GRASPIN DB - A Syntax Directed, Language Independent Software Engineering Database", *Proceedings of the Workshop on Object-Oriented Databases*, Pacific Grove, California, September 23-26, 1986, 235-236