



Adabase^{*}: A Data Base for Ada Programs

Walter F. Tichy

Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907

ABSTRACT

A central part of any programming support environment is the data base that stores all project information. This paper presents the design of Adabase, a data base that manages Ada program families. Program families consist of multiple versions and configurations, and family members share a significant number of common parts.

Besides Ada program modules and configurations, Adabase stores documentation and project control information. A set of high-level operations perform controlled update, propagation of interface changes, and automatic system generation. The logical data base structure is an attributed, directed graph. Since the standard, intermediate representation of Ada programs, Diana, is also an attributed, directed graph, we achieve a seamless integration of the two by formulating Adabase in IDL, the metalanguage in which Diana is described.

1. Introduction

A significant problem during software development and maintenance is that any nontrivial software system evolves into a family of related versions and configurations. The major causes for this phenomenon are imperfection and user demand[Bel79a]. The first cause, imperfection, results in a constant stream of repairs, with all the attendant problems of converting and maintaining old versions. The second cause, user demand, is a powerful force that causes new versions to arise almost spontaneously. Users always wish to apply a successful system in unexpected ways, in

* Adabase is not to be confused with the trademark ADABAS.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

unforeseen situations, or on different hardware. Invariably, this requires adaptations, improvements, bells and whistles to be added -- and additional errors creep in at the same time.

Multiple versions are a necessary evil. They are evil because they can complicate software development enormously. They are necessary, because we cannot eliminate any of their causes. Imperfection is inherent to any human endeavor, especially to one as error prone as writing programs. User demand will always be with us simply because it is faster to invent a new feature than to implement it. It is also quite impossible to predict, during the design stage, what demands a diverse user community will eventually place on a system. Moreover, even if we could write the all-encompassing software system that would work under all conceivable circumstances, no user would be happy with it because of severe overhead caused by excess generality.

There are two approaches for handling the multiple versions necessitated by user demand. One approach is to construct a new, unique system for every class of users. This approach has the advantage of unlimited flexibility, but is enormously costly. The second approach tries to economize by building system families whose members share common parts. This approach requires careful planning during the early design stages to make sure that family members can be configured from as many common parts as possible. Skillful management, design, and implementation must be combined to maintain order and to keep the family together.

This paper concentrates on the support tools for developing and maintaining system families. We present a data base design and a set of operations for managing multi-version, multi-configuration systems. The data base is the central data structure in a programming support environment, and almost all software tools access it. The structure of the data base is critical because it determines what kinds of families and what kinds of tools can and cannot be built effectively.

We propose using *attributed, directed graphs (ADGs)* for specifying data bases for program families. An ADG can represent both the contents and the structure of complex data objects. In our case, the nodes of the ADG store the data items like programs, documentation, and control information; the vertices linking the nodes represent the composition of the system family. The use of ADGs was suggested by a general model for system families, the AND/OR graph model. This model represents system families with a graph whose nodes are leaf nodes (for atomic units), AND nodes (for configurations), or OR nodes (for versions) [Tic82a]. ADGs have already been used successfully in designing and building a simpler data base for managing program revisions [Tic82b]. It also appears that graphical notation for representing the data base structure is more direct and convenient than any of the conventional data base specification tools. Finally, ADGs are used for intermediate representations of programs[Fei80a, Goo81a]. This can be exploited for a seamless integration of data base and programs.

The data base structure is strongly influenced by the programming language used, in particular the facilities for separate compilation. This is because the separately compilable program units are the primary entities in the data base. The interfaces between program units also affect the data base structure, because interfaces determine how the units can be composed into configurations, and how multiple versions are represented. The separate compilation and interfacing facilities of the programming language Ada[Ich80a] are sufficiently rich to permit flexible system families. This is why we chose to tailor the data base towards Ada. However, the discussion in this paper and the flexibility of ADGs should make it straightforward to adapt our design to other languages.

In Section 2, we discuss the interaction between Ada compilation units and the data base structure. Section 3 introduces the basic skeletons of Adabase, including multiple versions and configurations. Section 4 sketches a basic set of operations on the data base, including the facilities for automatic system generation and interface control.

2. The Structure of Ada Program Families

An Ada compilation unit is a subprogram declaration, a subprogram body, a package declaration, a package body, a generic unit, or a subunit. In this section, we discuss where we should introduce multiple versions for these compilation units, and how they are combined into configurations.

2.1. Compilation Units

The simplest program units are subprogram declarations and bodies. A *subprogram declaration* is a procedure

or function header, specifying the name, parameters, and return type. The *subprogram body* consists of the declarations and statements that implement the subprogram. Corresponding declarations and bodies may be compiled separately. We will exploit this separability by allowing multiple versions of subprogram bodies for every declaration.

A similar distinction holds between a package declaration and a package body. A *package declaration* contains declarations of logically related items, for example common data types, objects, and subprograms. The *package body* contains the implementation of the subprograms mentioned in the package declaration plus additional data structures. Other program units needing the facilities provided by a package refer only to the package declaration; the implementation of the package remains hidden. Again, we will exploit this separability by allowing multiple versions of a package body for every package declaration.

A package declaration may also have a *private part*. A private part contains data types whose structural details are irrelevant to their use outside the package. For instance, a package may provide a certain record type, but the actual layout of that record may be irrelevant to other units. The concrete layout of the record is then specified in the private part. Clearly, we should permit several versions of a private part for every package declaration, to give the implementor of the package adequate freedom for building needed versions.

A compilation unit may contain stubs instead of actual bodies of subprograms, tasks, and packages. A stub is only a placeholder; the actual body is in a separate compilation unit called a *subunit*. Adabase stores subunits in essentially the same way as ordinary units.

Generic units are templates of packages and subprograms. An ordinary package or subprogram becomes generic if it is preceded by a special clause. Adabase stores generic and ordinary units in the same manner by including the generic clause with the declaration.

2.2. Ada Program Configurations

An Ada program unit is said to depend on another unit if it uses that unit. For example, a package may use the data types and call the subprograms in other packages. This dependency is expressed with a *context specification* that lists all the units used. Clearly, context specifications establish configurations, i.e., combinations of units into systems and subsystems. The context specifications are interpreted by both the compiler and the binder. The compiler uses the list to look up the required package and subprogram declarations for performing type checking. The binder uses the list to retrieve the package and subprogram bodies that must be linked together.

Since our data base permits us to store multiple versions of bodies, the context specification is insufficient to select the right bodies in all cases. Special selection rules, defaults, and parameters for compiler and linker commands are needed. These will be discussed in Section 4.

3. The Logical Structure of Adabase

As mentioned in the introduction, we view the logical structure of Adabase as an attributed, directed graph. The vertices in the graph represent the organization of the program family. The attributes of the nodes store supporting information like access lists, locks, creation dates, derivation histories, etc. An implementation of Adabase is totally free to choose a suitable representation of the ADG. For instance, the ADG could be encoded in the file system of some operating system, or in relations of a relational data base system.

The data base may store source programs either in text form or in some intermediate form. The intermediate form is typically a tree generated by a syntax directed editor or a preprocessor. The standard intermediate form of Ada programs is Diana, which is also an attributed, directed graph. The Diana graph (actually a tree) is specified with IDL[Goo81a]. Using IDL for describing both the programs and the data base structure leads to a seamless integration of the two, because Adabase simply becomes an extension of Diana. Using IDL as a standard will also greatly simplify the portability of program data bases.

There is an important difference between Diana and Adabase. Diana is a standard that is unlikely to change. Portions of Adabase, however, will differ for different environments. This is necessary because different programming environments may have their own tools that require special supporting data structures. Examples are special test beds, various kinds of management data, access permissions, modification logs, etc. However, for the sake of portability, we will identify a kernel of Adabase that must be the same for all installations. This kernel stores the actual Ada programs and a standard documentation set. All additional data structures should be extensions of the kernel.

3.1. Basic Adabase Skeletons

In this section, we discuss the skeleton data structures needed for storing packages, subprograms, subunits, generics, and configurations. The skeletons are uniform for all compilation units. Section 3.2 fleshes out the skeletons with additional attributes.

3.1.1. Packages

The template for packages consists of two sets of versions groups: the package declaration group and the

package body group (see Fig. 1). The first group collects the versions of the package declaration. A single declaration is normally not sufficient, because it is likely to change over the lifetime of the package. Similarly, the second group combines all versions of the package bodies implementing the package declarations. Obviously, there should be significantly fewer members in the first group than in the second.

The package body group is organized as a forest of revision trees. A revision is created manually by revising an existing package body. A revision tree collects all revisions that were created from a single, initial package body. The tree structure reflects how the revisions were derived from each other. Since there may be several implementations of the package that evolve separately and may differ significantly (for instance, they may use different algorithms or data structures), several revision trees may be part of a single package body group.

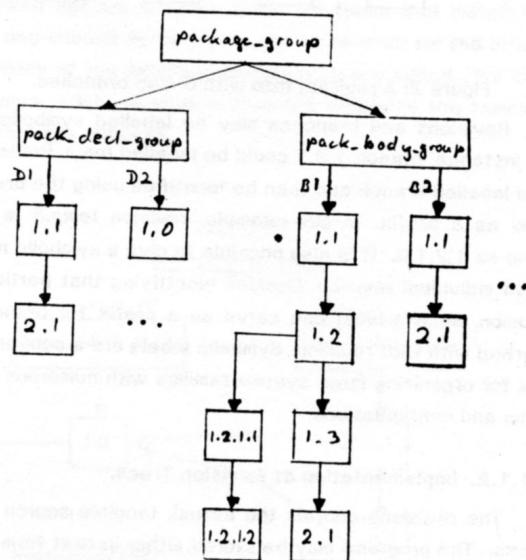


Figure 1: Package skeleton.

3.1.1.1. Revision Trees of Package Bodies

Figure 2 illustrates a single revision tree for a package body. The initial revision in that tree is numbered 1.1. Successive corrections, improvements, and expansions resulted in a sequence of revisions numbered 1.2, 1.3, and 2.1. This sequence is called the *trunk* and represents the mainline development of the tree. Sometimes it will be necessary to pursue separate lines of development, usually for exploratory or experimental purposes. To avoid disturbing the mainline development, experimental revisions should be placed on side branches. When it is time to combine several lines of development, a special operation for merg-

ing branches is available.

Branches are numbered *fork.1*, *fork.2*, ..., etc, where *fork* is the number of the fork revision. Revisions on a branch are again numbered sequentially, using the branch number as a prefix. For example, revision 1.2 of Figure 2 has two side branches, 1.2.1 and 1.2.2. Branch revisions may sprout additional branches.

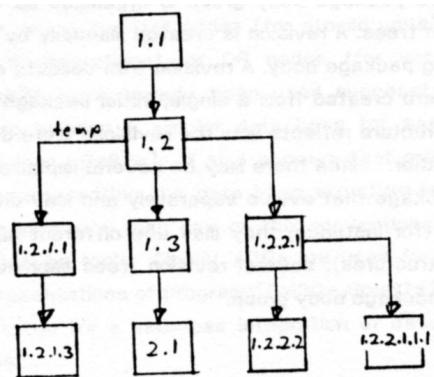


Figure 2: A revision tree with 3 side branches.

Revisions and branches may be labelled symbolically. For instance, branch 1.2.1 could be labelled *temp*. Revisions on a labelled branch can then be identified using the branch label as a prefix. In our example, revision *temp.3* is the same as 1.2.1.3. It is also possible to give a symbolic name to an individual revision. Besides identifying that particular revision, such a label can serve as a prefix for branches starting with that revision. Symbolic labels are a convenient tool for organizing large system families with numerous versions and configurations.

3.1.1.2. Implementation of Revision Trees.

The revisions contain the actual, tangible source programs. The programs may be stored either in text form or in some intermediate form, for example as Diana trees or as the output from a language oriented editor. To conserve space, one should store only *deltas*, i.e., successive differences. Differences between text files are easy to compute on a line-by-line basis [Hec78a]. Finding differences between program trees is more expensive, but a language oriented editor could provide the differences by recording changes during editing sessions.

The revision tree should be organized in such a way that the latest revision can be regenerated quickly, since it is the one accessed most often. A technique with this property stores the latest revision intact and uses reverse deltas to regenerate older revisions. A reverse delta is simply an edit script that transforms a revision into its predecessor. For example, to regenerate revision 1.1 in Figure 2, the retrieval operation would extract revision 2.1 and then

apply the edit scripts for revisions 1.3, 1.2, and 1.1 ... sequence.

Experience shows that the use of deltas leads to immense space savings, because normally successive revisions differ only slightly. In fact, differences make the "luxury" of saving multiple revisions economically feasible. However, branches need special treatment to preserve the space savings. The naive solution would be to keep complete copies for the ends of all branches, including the trunk. Clearly, this is unacceptable because it requires too many complete copies. Instead, one should use the following arrangement. The latest revision on the trunk is stored intact, the deltas on the trunk are reverse deltas as before, but deltas on side branches are *forward* deltas. Regenerating a revision on a side branch proceeds as follows. First, retrieve the latest revision on the trunk; second, apply reverse deltas until the fork revision for the branch is obtained; third, apply forward deltas until the desired revision is reached. Figure 3 shows the tree of Figure 2, with each node a triangle whose tip points in the direction of the delta. For an analysis of the various techniques see [Tic82b].

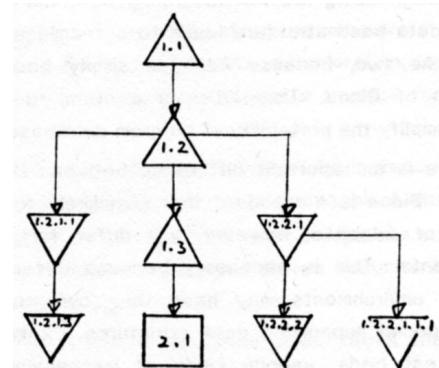


Figure 3: Use of forward and reverse deltas.

3.1.1.3. Revision Trees for Package Declarations

For uniformity, we chose to organize package declarations as forests of revision trees as well. Although we expect that branches in these trees will be rather short, this approach has the advantage that the operations for adding and retrieving revisions are the same for bodies and declarations.

Revisions of package declarations require an additional data structure, namely the private group. This group consists of one or more private parts that specify structural details of private types in the package declaration. Thus, each package may have a set of different concrete representations for its externally visible data types, giving the implementor considerable freedom in implementing a

package body. A single private part may be shared among several private groups.

A particular revision of a package body can implement only a single revision of a package declaration. This fact is recorded with vertices linking each body to its declaration. On the other hand, a revision of a package body may work with several of the private parts belonging to its declaration. An example is if the private parts differ only in some constants. Thus, each revision of a package body must also list the private parts (if any) with which it is compatible. Figure 4 presents a complete package group with several of the links between bodies and declarations shown.

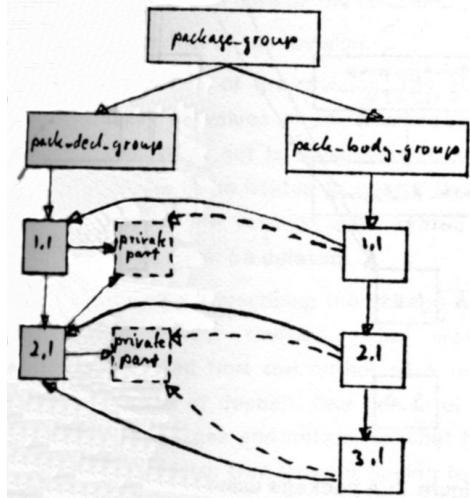


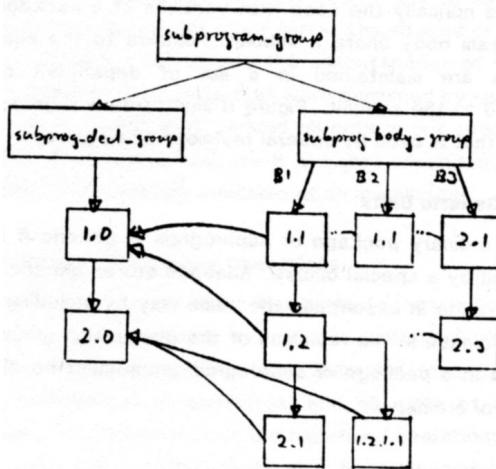
Figure 4: A package group with links between declarations and bodies.

3.1.2. Subprograms

The skeleton for subprogram groups is isomorphic with the package group (see Figure 5). The only exception is that there are no private parts. An important consideration is that subprogram declarations need not be entered into the data base explicitly, because they can be derived from the body. In this case, storing the first body of a subprogram has the side effect of deriving and saving the corresponding declaration. When a later revision of the body is entered, it is checked against the previous declaration, and a new declaration is added if necessary.

3.1.3. Subunits

Subunits are bodies of subprograms, packages, and tasks whose declarations appear in other program units. When a unit containing subunit declarations is deposited into the data base for the first time, entries for the subunits are also created. The subunit declarations are stored into the corresponding declaration groups, and the body groups are left empty. The bodies are normally added later. When a body is deposited, it is checked against the exist-



It is normally the case that versions of a package or subprogram body share a subunit. Pointers to the sharing versions are maintained in a list of dependent units attached to the subunit. Figure 6 illustrates a subprogram subunit that is used by several revisions of a package.

3.1.4. Generic Units

An ordinary package or subprogram is generic if it is preceded by a special clause. Adabase stores generic and ordinary units in essentially the same way by including the generic clause in the revisions of the declaration group. All versions in a package or subprogram group must be either generic or ordinary.

3.1.5. Configurations

As mentioned in Section 2.2, context specifications determine configurations. In Ada, context specifications are lists of names of packages and subprograms. Since Adabase may store multiple versions of these units, such a simple list is no longer sufficient. At the minimum, a context specification must indicate which revision of each declaration is selected. The selection of the corresponding body can be delayed until linking time, and is usually resolved by defaults, although that too must be specifiable explicitly. The actual programming language syntax for selecting versions is of no concern here. In the ADG, this information is represented by attaching a context node to all declarations and bodies in our skeletons. Figure 7 illustrates a package depending on several other units.

In analogy to the subunit groups, each subprogram and package declaration also maintains a list of dependent units. This list is actually generalized in that it also records the body revisions implementing the declaration (not shown in the figures). This information is important for propagating interface changes (see Section 4). Note also that a single context specification may spawn a large number of versions, because each of the units usually has a number of bodies that implement the selected declaration.

3.2. Detailed Structure of Adabase

This section adds additional attributes to the skeletons introduced above. We will only discuss packages; subprograms and subunits have an identical set of attributes. The IDL specification of package groups can be found in the Appendix.

Each package group has a name, a creation date, an author (the person who initialized the group), and a description. The description should give an overview of the package group and is requested at the time of initialization. The description may have several, standardized subsections, filled in with a special editor. The kernel of Adabase leaves the fine structure up to the particular implementation and

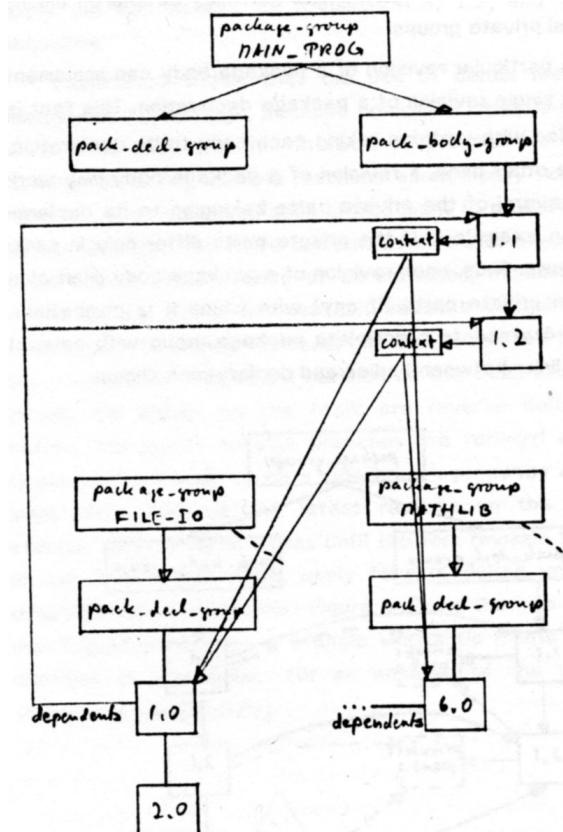


Figure 7: A package using 2 other packages.

treats the description simply as a character string.

The nodes for the package declaration group and the package body group need no additional attributes. A revision tree (for both declarations and bodies) has the following fields:

- | | |
|---------------------|---|
| Name: | The name of the revision tree. |
| Description: | A description of the revision tree, stating, for instance, the implementation strategy. Again, there may be standardized subsections, but the kernel Adabase treats descriptions as uninterpreted character strings. |
| Accesslist: | A list of programmer names who may deposit or remove revisions and change tree attributes. The accesslist only restricts write operations; read operations are possible for everybody with general read permission for the data base. |
| Locklist: | A list of pairs (programmer name, revision number). A pair indicates that the programmer with the given name is currently modifying the revision with the given number, and will deposit a new revision in the near future. Locks prevent two or more programmers from making competing changes to the same revi- |

sion. Locks do not affect read permission.

NameList: A list of pairs (name, number). A pair specifies how a symbolic name is mapped to the number of a revision or a branch in the tree. Symbolic names relieve the user from remembering revision numbers and can be used to give revisions in different trees the same name, although their numbers may be different.

An individual revision has the following attributes:

Number: The number of the revision.
Date: The creation date of the revision.
Author: The author of the revision.
Status: The status of the revision. The status field takes on values of an enumerated type. A convenient set is *experimental*, *stable*, and *released*. The status is useful for selection purposes. A revision with status *released* should never be deleted.
Log message: A message describing the change that led to the current revision. This message is requested from the author of a revision at the time of deposit. One can later read the log messages and determine what happened to a revision tree without having to compare source programs.
Program: This attribute contains the actual program text, either a declaration or body for a subprogram, package, or subunit. Except for the latest revision on the trunk, the program text is merely a delta. The delta is an edit script for text representations, and a tree transformation script for intermediate representations.
Derivations: A set of objects derived from the *Program* attribute by means of processors like compilers, linkers, etc. For instance, there might be several object modules in this set, each compiled for a different machine. If the *Program* attribute is represented in some intermediate form, one of the derived objects might be its pretty-printed text version. Derived objects can always be regenerated, so they can be deleted whenever free space in the data base is getting low.

Adabase manages derived objects fully automatically. Each derived object has a history attribute that records which versions of which other objects (usually declarations and private parts) were used during the object's generation. In this manner, data base operations can determine

automatically whether a derived object has to be regenerated after changes of other parts. Linked sets of several object modules must include the concatenation of the history attributes of all units that were combined by the linker. The exact structure of derived objects is not specified in the kernel Adabase, because it is highly dependent on the compilation technology available at an installation.

4. Summary of Adabase Operations

This section discusses the basic Adabase operations. For brevity, many useful, but obvious operations are omitted, for example operations for initialization, renaming, deletion, modification of attributes, and all sorts of queries. Instead, we concentrate on the high-level operations, those that automate numerous tasks that normally must be done manually.

Navigation through the data base is interactive. There are two possible styles of navigation, and Adabase can support either one. The first is the style offered by many data base query languages, in which the user can move a cursor through the data structure using a variety of commands. At each step, he issues additional commands to display information addressed by the cursor.

The second style of navigation uses the technology of language oriented editors. Such editors normally offer commands that create and manipulate programming language constructs instead of character strings. They also provide facilities to navigate through program syntax trees. These facilities can be carried over to data base navigation by extending the programming language to also describe the data base. The editor can then be used to manipulate programs as well as the data base structure. Wherever the user positions his cursor, the editor automatically displays the structure of the surrounding nodes. This approach simplifies navigation and provides a uniform user interface for editing both program and data base. This idea was pioneered by A.N. Habermann[Hab82a].

The normal procedure for the day-to-day editing of programs is a 3-step process. First, the programmer selects an existing revision and checks it out for modification. The checkout operation locks the selected revision to prevent competing modifications and copies the revision into the programmer's workspace. Second, the programmer edits and tests the revision in his private workspace until he is satisfied. Finally, he checks the modified revision back in. The checkin operation tests whether the programmer holds a lock for the previous revision, deposits the new one, and releases the lock. It also prompts for a log message describing the change.

A question at this point is when to check whether a revised body is consistent with its declaration. Ideally, a language oriented editor should perform the analysis during

the editing session. The editor simply retrieves the corresponding package declaration for that purpose. If no such editor is available, another approach is to perform a semantic analysis during checkin and to abort the operation if the analysis fails. A third alternative would be to wait with the analysis until the revision is compiled. This approach has the disadvantage that possibly important error messages are delayed. In any case, our representation is general enough to work with all three techniques.

Revising a declaration may have more serious consequences than revising a body. For example, changing a parameter list of a subprogram may require that all units using that subprogram be updated. For this purpose, Adabase offers a special operation for propagating declaration changes. The operation scans the list of dependent units inherited from the parent revision. This list records all bodies and declarations that use the declaration (including the bodies implementing it). For every unit on that list, the propagate operation asks the user whether he would like to update the unit. If he answers yes, the unit is checked out and the user can change it. Otherwise, the element in the dependents list is marked as out of date.

Package declarations usually contain several items. A change of a single item may not affect all units appearing on the dependents list of that declaration, because not all units may use that particular item. To avoid redundant checkouts and recompilations, a dependents list should actually be attached to each individual item in a package declaration rather than the package as a whole. The lists are updated during checkin operations or during compilations.

Adabase offers a general-purpose selection operation that is used to retrieve a particular revision. Normally, the retrieved revision is passed on to other tools, for example the compiler, linker, debugger, pretty-printer, cross-referencer, etc. The selection operation takes the name of a version group, an indication of whether to chose a declaration or a body, and optionally a tree name, a revision number (symbolic or numeric), an author, a date and a status. If none of the optional parameters is given, the selection operation checks whether the user is currently editing a revision of the given version group. If so, that one is selected, otherwise the latest revision on the trunk of the default revision tree. If optional parameters are given, the latest revision that satisfies all parameters is selected.

The selection operation is used heavily by the construction command. This command compiles and links programs. Given a program unit, it compiles it and the units appearing in its context list. This is a recursive process. Of course, the operation avoids recompilation if checking history attributes reveals that the compiled version already exists. The build operation takes the same parameters as

the selection operation and applies them uniformly to retrieve all needed revisions. Symbolic revision names assigned judiciously throughout the data base make it possible to select individual configurations with a small number of command parameters. There is also a command option that prompts for the proper version whenever there is a choice.

5. Conclusions

We have presented a data base design that stores families of Ada programs. We emphasized the family aspect by permitting versions wherever desirable. In practical applications, not all Ada program units will require the full range of versions. However, the data base must be prepared to handle them. If only a few kinds of versions are used, the user interface should hide the unneeded complexity.

Major building blocks of Adabase are the revision trees. We have implemented a simpler system, RCS (short for Revision Control System)[Tic82b], that handles individual revision trees of text objects. It conserves space by storing differences as discussed in Section 3.1.1.2. RCS is used to manage revisions of programs, documentation, papers, and all kinds of other long-lived text. Currently, we are investigating how the method of differences can be extended to graph structured objects, as generated by language oriented editors. A long-term project is to implement a prototype of Adabase and to investigate distributed program development data bases.

Appendix: Partial IDL Specification of Adabase

```

Structure ADABASE Root unit_group is

unit_group ::= package_group | subprogram_group | subunit;
sub_unit   ::= sub_package_group | sub_subprogram_group
              | sub_task_group;

-- only the package group is defined here.

package_group => name          : String,
                  date         : String,
                  author       : String,
                  description  : String,
                  generic      : Boolean,
                  pack_decl_group : Seq Of revision_tree,
                  pack_body_group : Seq Of revision_tree;

revision_tree => name          : String,
                  description  : String,
                  accesslist   : Seq Of name,
                  locklist    : Seq Of lock,
                  namelist    : Seq Of name_binding,
                  root_revision : revision;

revision     => rev_number   : Seq Of Integer,
                  date         : String,
                  author       : String,
                  status        : States,
                  log_message  : String,
                  program      : COMP_UNIT,
                  --COMP_UNIT is defined in Diana.
                  derivations   : Set Of String,
                  next_revision : revision,
                  -- link to next revision on branch.
                  branches     : Seq Of revision,
                  -- branches forked by a revisions.
                  context       : Seq Of revision,
                  -- pointers to used revisions.
                  dependents   : Seq Of revision,
                  -- pointers to using revisions.
                  declaration   : revision,
                  -- pointer to declaration implemented by a body.
                  private_parts : Seq Of DECL REP_S;
                  -- for declarations, private_parts lists the
                  -- available parts; for bodies, private_parts
                  -- lists the compatible private parts.
                  -- DECL REP_S is defined in Diana.

lock         => name          : String,
                  rev_number   : Seq Of Integer;

name_binding => name          : String,
                  rev_number   : Seq Of Integer;

-- enumerated type for Status attribute.
States       ::= Experimental | Stable | Release;
Experimental ::= ;
Stable       ::= ;
Released     ::= ;

```

End

References

- Bel79a. Belady, L.A. and Lehman, M.M., "The Characteristics of Large Systems," pp. 106-138, in *Research Directions in Software Technology*, ed. Peter Wegner, M.I.T. Press (1979).
- Fei80a. Feiler, Peter H. and Medina-Mora, Raul, *An Incremental Programming Environment*, Technical Report, Carnegie-Mellon University, Department of Computer Science (April 1980). Also presented at the Workshop on Programming Environments, Ridgeville, CT in June 1980.
- Goo81a. Goos, Gerhard and Wulf, William A., *Diana Reference Manual*, Technical Report, Carnegie-Mellon University, Computer Science Department (March 1981).
- Hab82a. Habermann, A. Nico, *The Second Compendium of Gandalf Documentation*, Technical Report, Carnegie-Mellon University, Department of Computer Science (May 1982).
- Hec78a. Heckel, Paul, "A Technique for Isolating Differences Between Files," *Communications of the ACM* 21(4) pp. 264-268 (April 1978).
- Ich80a. Ichbiah, Jean D., *Reference Manual for the Ada Programming Language*, United States Department of Defense (July 1980).
- Tich82a. Tichy, Walter F., "A Data Model for Programming Support Environments and its Application," in *Automated Tools for Information System Design and Development*, ed. Hans-Jochen Schneider and Anthony I. Wasserman, North-Holland Publishing Company, Amsterdam (1982).
- Tich82b. Tichy, Walter F., "Design, Implementation, and Evaluation of a Revision Control System," in *Proceedings of the 6th International Conference on Software Engineering*, IPS, ACM, IEEE, NBS (September 1982).