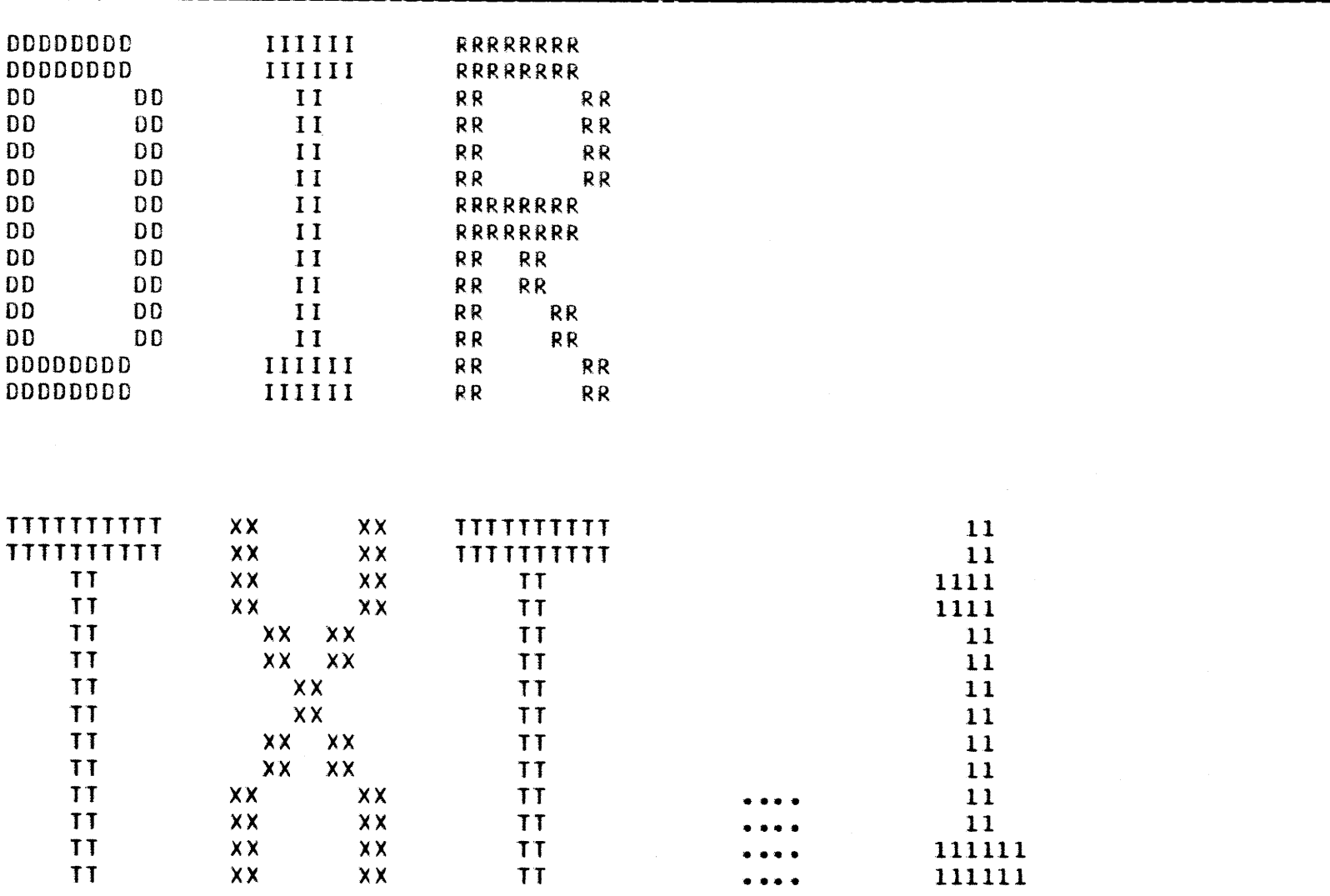


START Job DIR Req #332 for EGB)ate 15-Feb-84 15:43:15 Monitor: //, TOPS-20
File RM:<MTD.IMPL>DIR.TXT.1, created: 26-Sep-83 22:12:57
printed: 15-Feb-84 15:43:15
Job parameters: Request created:15-Feb-84 15:36:18 Page limit:225 Forms:NORMAL
File parameters: Copy: 1 of 1 Spacing:SINGLE File format:ASCII Print mode:ASCI

PACKAGE DIRECTORY SYSTEM

1. INTRODUCTION

PURPOSE

The purpose of this note is to describe the design and implementation of the R1000 package directory system. The package directory system is a central component of the programming environment both from the point of view of the user visible model of the environment and from the point of view of system design and implementation.

BACKGROUND

On the R1000, the universe (that is the set of types, objects and operations available to users (interactively or programmatically)) is represented as a hierarchy of nested Ada packages. Ada declarations in this "package directory" are used to represent all existing entities.

Declarations in this universe may be source, installed, or elaborated. Source means that the declaration exists in "text" form only and can not be referenced by other parts of the system. Installed declarations are an active part of the package environment, are semantically consistent, and may be referenced by other installed entities. An elaborated declaration is installed and elaborated (in the Ada sense) with a actual (runtime) value which may be accessed and manipulated.

The set of installed declarations forms a subtree rooted at the root of the universe and covering everything out to the source subtrees. Similarly the set of elaborated packages is a subtree of the installed universe rooted at the root of the universe.

There are a number of important invariants which apply to the installed universe, the most important being that the installed universe is semantically consistent (in accordance with Ada semantics). The system is designed to be understood and manipulated in terms of Ada semantics. Ada provides the semantic framework for understanding name resolution, objects and values, types, operations, and structuring (packages), as well as pragmatic and operational issues such as parameter passing, exception handling, and concurrency.

While Ada provides much of the semantic framework, it is essentaily a static, compilation-oriented language. In an interactive programming environment, many of the operations which manipulate the environment itself (adding new declarations, deleting declarations, executing a command in some context, etc.) must be viewed as meta-operations that occur outside of Ada semantics. These meta-operations transition the environment from one semantically consistent state to another; but, the resulting state may be one which could not have been reached through elaborating and executing a pure Ada program. For example, declarations in the environment are elaborated in a time-ordered, incremental fashion, rather than using strict linear elaboration order.

In addition to meta-operations which manipulate the environment itself, we must go beyond Ada semantics to address certain essential aspects of programming and software engineering. Ada semantics does not address issues such as project management (costing, scheduling, etc.), configuration management and version control. These aspects of software development must be integrated with Ada in a manner consistent with the programming methodologies which underly the effective use of Ada.

The package directory system is the central component in supporting the Ada-based semantic framework, in providing the set of meta-operations consistent with the needs of an interactive programming environment, and in addressing higher-level software engineering issues.

SCOPE

This document will provide some motivation and rationale for the design of the package directory system, but will primarily focus on the actual design structure and implementation strategy.

Section 2 describes the overall system structure and summarizes the major components. The description is given bottom-up so that higher levels can be introduced in terms of lower levels.

Section 3 describes the design and implementation of the central directory operations in terms of the facilities provided by lower levels. This section covers actions and queing, error conditions, intalling declarations, elaborating declarations, deleting declarations, withdrawing declarations, and the treatment of source declarations.

Section 4 describes the interactions between the editor system and the directory system.

Section 5 describes the file directory system and the implementation of file abstractions on the R1000.

Section 6 describes the design and implementation of the initial dependency data base.

Section 7 gives an overview of the design and implementation of incremental semantic analysis, including a description of the incremental symbol table facilities and a summary of the interface to the dependency data base.

Section 8 describes the design and implementation of change analysis, including interactions with the dependency data base and with the Ada manager.

Section 9 summarizes those aspects of Diana and the Ada manager relevant to understanding the package directory sytem.

Section 10 describes the incremental code generation facilities required by the package directory system.

Section 11 describes the runtime object manager facilities required by the package directory system.

The organization of this document is such that a single pass by the reader unfamiliar with the material will be somewhat frustrating. Little or no effort is made to define terms or explain concepts that are assumed to be generally known (at least within the software group). The primary purpose of this document is to capture information, not necessarily organize it in any particular way.

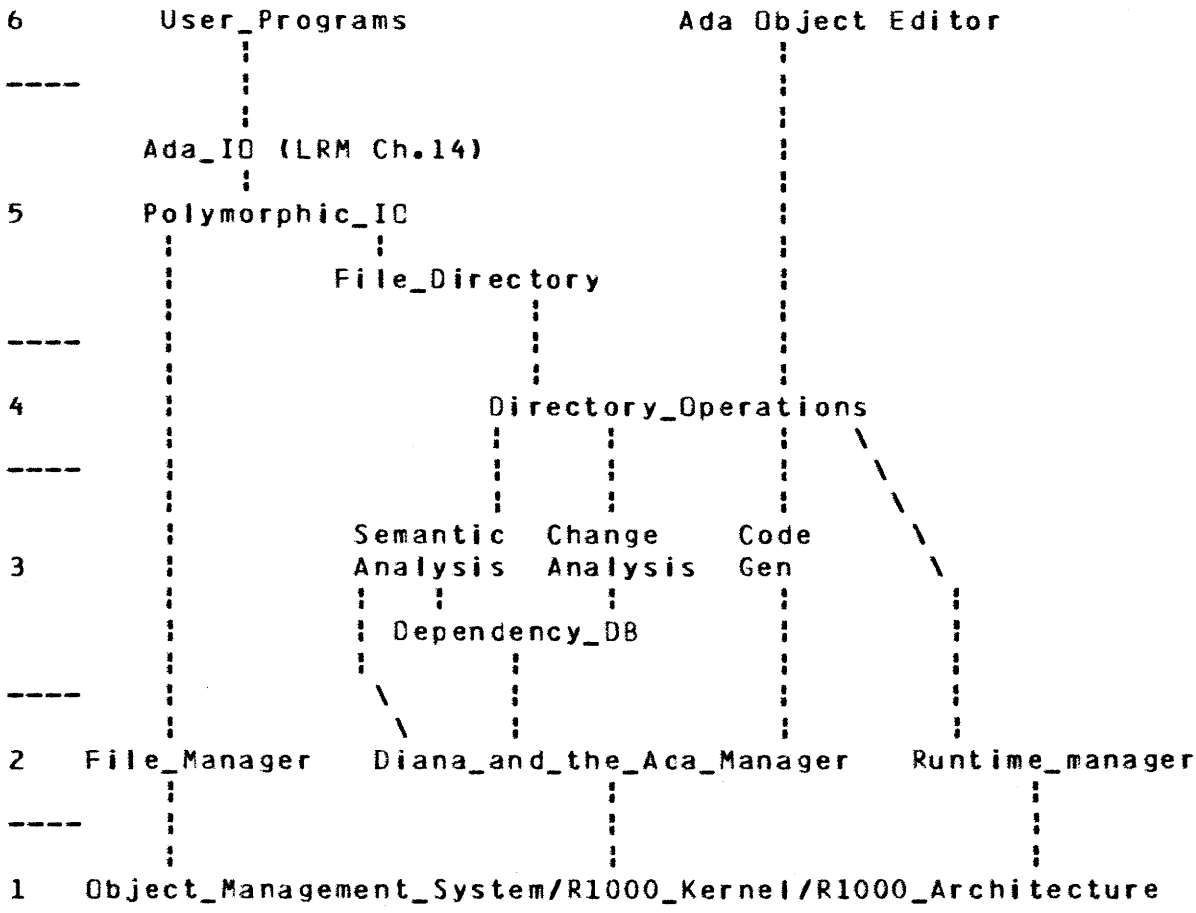
The chosen organization gives a bottom-up overview of the system in Section 2, followed by a inside-out description of the design and implementation. Section 3 describes the central directory operations and in some sense provides an overview of the implementation. Sections 4 and 5 focus on usage by higher levels of the system, in particular by the editor system and by programs written in terms of files. The remaining sections focus on lower-level components of the package directory system.

2. SYSTEM STRUCTURE

GENERAL

In this section we will describe the overall sysem structure and summarize each of the major component and its relation to the rest of the system. Later sections describe major components in more detail. The portions of the system dealing with code generation and the runtime environment are considerably less well-defined at this point.

Figure 1. Basic Directory System Structure.



The basic system structure can be seen in Figure 1. The system is basically structured into six layers -- the mechanism layer, the manager layer, the compilation layer, the directory layer, the file layer, and the user layer.

MECHANISM LAYER

The first layer includes the object management system, the R1000 kernel, and the R1000 architecture. This layer provides the fundamental low-level mechanisms and is not addressed in this document.

MANAGER LAYER

The second layer consists of the file object manager, Diana and the Ada manager, and the runtime manager. This layer builds on the previous to provide specific facilities for storing and managing Ada objects and file objects, and for easily manipulating the facilities of the architecture.

The file manager provides a low-level file abstraction integrated with all of the object management mechanisms. This file mechanism is built upon segmented heaps. The file type provided here is a polymorphic file type which allows elements to be of different types. The visible part of the file manager is available on the 2060 in <rr.managers>v_file.ada.

The package directory system is basically represented as a set of Diana trees managed by the Ada manager. Diana and the Ada manager are implemented in terms of segmented heaps. Semantic attributes representing semantic dependencies are implemented directly as cross-segment pointers. This constrains directory operations in that the validity of these cross-segment pointers must be maintained by all update operations. In addition to providing storage for the Diana tree, the Ada manager provides storage for auxilary information, including the state of the Diana unit (source, installed, elaborated). The operations provided by the Ada manager were selected specifically to support the kinds of "semantics preserving" operations that the directory system must perform. The Diana Reference Manual (2/28/83) describes Diana. The visible part of Diana and the Ada manager can be found on the 2060 in <diana.sim>v_diana.ada and <diana.sim>v_ada_manager.ada.

The runtime object manager is responsible for managing information about the runtime environment and performing the actual updates to elaborated packages. On the R1000 the directory system actually stores objects in elaborated packages. This allows objects to be addressed using the basic architectural addressing mechanisms and allows the use of Ada semantics for parameter passing, command and program execution, etc.

COMPILATION LAYER

The compilation layer consists of a set of large building blocks for constructing the directory layer. Semantic analysis and change analysis are primarily concerned with checking and maintaining consistency and supporting Ada semantics. The code generator and a variety of Diana utilities are used to actually affect changes after it has been determined that a change is legal.

Semantic analysis is responsible for determining when a declaration is semantically consistent, for attributing the Diana tree, and for reporting any errors. Semantic analysis also updates the dependency data base used by change analysis and other tools. A set of incremental interfaces have been added to semantic analysis, supplementing the traditional, batch-oriented interfaces. Semantic analysis is stuctured as a utility procedure. The client must interface properly with Diana and the Ada manager to set the proper defaults and get proper access to the tree being analyzed. The visible part of the semantic analysis package can be found in <semantics>v_semantics.ada.

Change analysis is concerned with determining the impact of incremental changes in the installed or elaborated universe. The system will not allow changes which would invalidate the invariants defined for the installed (and the elaborated) universe. Whenever a new declaration is installed, elaborated, or withdrawn, change analysis is invoked to determine whether such a change would obsolesce other entities. An entity is obsolete when it no longer satisfies the invariants which apply to that entity. Change analysis must retrieve information from the dependency data base to determine the set of units possibly affected by a change.

The code generator is invoked by the directory system in order to construct the code segments required to interface with the runtime manager in actually performing directory operations. Code generation provides interfaces for compilation of entire units, and a set of incremental interfaces which allow the creation of code fragments for individual declarations and statements (including code necessary to perform re-elaboration operations). In addition, code generation must cooperate with the runtime manager in terms of the assignment of offsets and other code generator attributes and in terms of managing runtime dependencies.

DIRECTORY LAYER

As Figure 1 indicates, the focal point of the directory system is a set of directory operations which provide the only means for modifying installed or elaborated declarations. There are operations to open insertion points (used by the editor), install individual declarations, install sets of declarations, install subunits, delete declarations, elaborate declarations, withdraw declarations, and get/set the value associated with a declaration. The visible part containing these operations can be found on the 2060 in <package_directory>v_dir_ops.ada.

FILE LAYER

The file layer provides file abstractions that can be used by system and user programs. The two primary components of this layer are the file directory and the actual file abstractions.

The file directory is an abstract directory built upon the primitive directory operations to provide basic directory functionality while hiding all knowledge of Diana and the Ada manager. This provides a fairly conventional view of directories and is used for implementing all file abstractions. The visible part of the file directory is in <package_directory>v_file_dir.ada.

The R1000 file abstractions are built on top of the file object manager and the file directory interface. The basic file I/O package on the R1000 is Polymorphic_IO which supports input and output of objects of any "safe" type to any given file. The visible part can be found in <package_directory>v_poly.ada. This package is the basis for the implementation of all of the Ada IO packages (Ada LRM, Ch. 14).

USER LAYER

The user layer is essentially everything above the directory and file layers. This includes the editor system and any user programs which perform IO.

The Ada object editor interfaces directly to the directory operations to allow interactive operations that modify the installed universe. The editor system must keep its data structures consistent with the directory data structures, must communicate problems to the user (semantic errors, obsolescence, etc.), and concern itself with properly synchronizing multiple users trying to interact with the package directory system. The editor follows a set of fairly rigid protocols which are designed to ensure reliable operation.

User programs will typically use the facilities described in Chapter 14 of the LRM, but Polymorphic_IO will also be available. Polymorphic_IO provides a more general file mechanism with very flexible operations. In some sense, Polymorphic_IO will be used on the R1000 like Common_IO is used on the MVs.

3. DIRECTORY OPERATIONS

GENERAL

The package Directory_Operations encapsulates all access to the underlying directory system and provides the primitives for implementing all of the meta-operations which change the state of the directory system. All modifications to installed and elaborated packages are performed by this package. The current set of operations is defined in <package_directory>v_dir_ops.ada. In this section we will review all of the major operations provided by Directory_Operations. The goal is both to describe the operations and to give some insight into the interactions between the various system components.

ACTIONS AND QUEUEING

All of the operations in Directory_Operations have an Action.Id parameter. If no Action.Id is provided by the caller, the operation will be performed as a single atomic action, and all locks will have been released when the operation completes. This does require that the caller have no locks on the unit being modified.

When the directory operation is part of a larger action (which may involve no more than acquiring a read lock on the unit sometime before invoking the directory operation) the caller provides the Action.Id. When the directory operation acquires an overwrite lock on the unit being modified, any locks held by the current action may be up-graded. Any errors (semantic errors, errors from change analysis, etc.) cause the unit being modified to be closed without saving it. This allows the caller to retain control over when the action is committed or abandoned, while guaranteeing that the actual overwrite only occurs if there are no errors.

All operations also have a Queue_Request parameter, which allows the caller to indicate whether the request should be queued if some other action has a lock on needed objects. If queuing is indicated (the default), then all attempts to open objects will be queued. If queueing is suppressed, then the operation will fail immediately if some object is locked by another action.

ERROR CONDITIONS

The primary types of errors which may prevent a directory operation from succeeding are object management errors, semantic errors, dependency errors, and operational errors.

Object management errors include all of the usual locking, queuing and access control errors. Typically the execution of a directory operation will involve acquiring overwrite access to one Ada unit, acquiring read access to several Ada units, and potentially acquiring elaborate access to an elaborated package. Any of these may fail for any number of reasons.

Semantic errors result from any attempt to install declarations which are not semantically consistent with the state of the universe. These errors are normal Ada compile time errors and are reported using the data structures described in <semantics>v_errors.ada. The directory operations are responsible for insuring that errors are reported in terms of the input trees, rather than in terms of any temporary address spaces.

Dependency errors are detected and reported by change analysis. Change analysis can provide a list of all dependencies that would be jeaopardized by a proposed modification to the universe (i.e., deleting a type declaration upon which several object declarations depend or some such).

Operational errors result from incorrect usage of the specific directory operations. Examples include trying to open an insertion point anywhere except within a DECL_S or ITEM_S (or STM_S), or trying to install three declarations into two insertion points.

INSTALLING DECLARATIONS

A declaration may be installed if the unit containing it is installed, and everything it references is also installed. The basic mechanism for installing declarations involves first opening a set of insertion points, and then installing unrooted declarations at those points.

The Open_Insertion_Point operation takes a Diana tree representing a declarative part (either a DN_DECL_S or a DN_ITEM_S) and a position within the declarative part (a position of zero indicates the beginning of the declarative part). The unit where the insertion is to be made is opened for overwrite, the validity of the insertion is checked, a DN_NONTERMINAL is inserted into the Diana tree in the appropriate place, and the unit is closed.

The Install operation takes a list of insertion_points and a list of trees to be inserted. The unit being modified is opened for overwrite, the unrooted insertions are copied into the unit, replacing the insertion points, semantic analysis is performed on each insertion in turn, and (if there were no semantic errors) change analysis is performed on each insertion. If there were any errors the unit will be closed without saving changes.

The Insert_and_Install operation combines the previous two operations for simple cases involving only a single declaration. Using this form of install in simple cases is more efficient, since the unit being modified is only overwritten once.

The Install_Subunit operation differs from the previous forms in that the subunit is not copied into the parent, but associated as a subunit. The id or designator of the stub declaration for the subunit must be provided. The subunit is attributed by semantic analysis, including attributing the AS_NAME of the DN_SUBUNIT and setting the SM_SPEC and SM_STUB on the subunit designator using the information in the STUB_ID.

ELABORATING DECLARATIONS

A declaration (and its subunits) must be installed before it can be elaborated, and everything it references during elaboration must already be elaborated (basically Ada eleboration rules).

Elaboration involves generating any necessary code (code for large program units will probably have already been generated), including the code segments necessary for performing elaboration operations. These code segments and necessary context information (including the Diana trees for the declaration being elaborated) are passed to the runtime manager which actual invokes the elaboration operations at the machine level.

WITHDRAWING ELABORATED DECLARATIONS

Withdrawing an elaborated declaration (leaving it as an installed declaration) involves providing the runtime manager with the information it needs to remove the declaration from the runtime environment, freeing resources as appropriate. A declaration may not be withdrawn in this fashion if any elaborated declarations depend upon it.

WITHDRAWING INSTALLED DECLARATIONS

Withdrawing an installed declaration first involves running change analysis to determine whether such a change would obsolesce any other declarations. If not, the declaration is removed from the Diana tree, and replaced by a DN_NONTERMINAL. The nonterminal has an associated_object attribute linking it to a new object where the declaration is then stored as a source object.

DELETING DECLARATIONS

Deleting a declaration basically involves withdrawing the declaration without making an insertion point and saving the source form of the declaration.

SOURCE OBJECTS

Source objects are represented by an insertion point with an associated_object attribute linking the insertion point to a separate object containing the source declaration. These source objects may in turn have nonterminals with links to other source objects. Source objects are less well-controlled than installed or elaborated declarations, and care must be taken not to lose soure subunits of source objects. When creating an insertion point it is possible to identify an associated source object, or the association may be established later using the Associated_Object operations.

OBJECT VALUES

The values contained by objects are stored in the runtime environment representing the package directory system. Directory_Operations provides a generic package for retrieving values from the runtime environment. Initially we will restrict usage to managed objects.

4. EDITOR INTERFACES

EXAMINING THE PACKAGE DIRECTORY SYSTEM

The user examines the universe using the editor system. The user will typically see a package in some context, with subunits elided to allow most of the package to fit conveniently on the screen. The user can examine elided subtrees by expanding the elision, getting a new window with the contents of the subunit.

Examining installed or elaborated packages raises certain synchronization issues. When the editor needs to first examine an Ada unit, the editor must open the unit with a read lock while it is building the object tree (the editor's internal data structure for representing pretty printing and elision information), so that there are no modifications to the unit while the object tree is being constructed. However, if the editor holds the read lock as long as that image exists, then no one may make any changes to that unit. The analogous situation on Tops-20 would be if doing VDirectory on <JIM> were to prevent JIM from adding or deleting anything in his directory. This is particularly nasty if the unit under consideration is relatively high in the hierarchy (:pdd on the MV's for example) where it is more likely that someone else would have a read lock when one is trying to make a change.

The solution to this problem is that the editor will try to minimize the holding of read locks. For the many windows in which the user is only reading or browsing, the editor can often get away with releasing the read lock as soon as the object tree has been constructed. Many of the editor operations used in browsing or simple reading only use the object tree.

There is the problem that the image may not correspond to the current value of the object. This can be addressed by periodically (at some fixed time interval or based on some notion of where the user's attention is currently focused) checking the time of modification of the underlying object. Only if the object has been modified (hopefully a rare occurrence if many people are looking at it) must the editor do anything. The obvious alternatives are to ask the user if he even cares, or to acquire another read lock, reconstruct the object tree, and the release the lock. Reconstructing the object tree need not imply rebuilding the entire object tree. The editor may determine which declarations have been removed and inserted and make only those changes to the tree. WP has hypothesized that running the pretty printer will have this effect.

There are variety of operations which do require a current Diana tree. Clearly any modifications to the contents of the window would require consistency between the object tree and the Diana tree, but in those cases the editor must have some kind of write lock and can update the object tree in a manner consistent with any changes to the Diana tree. Operations like "show defining occurrence" do not require an update lock, but do require that the object tree be consistent with the Diana tree. Actually, show defining occurrence and many similar operations probably will work correctly, since nodes are not reused.

In general, if the editor is doing something that requires a consistent Diana tree, then it must have a read lock. If the editor records the time when it creates the object tree, then it can use the time of modification of the unit to determine whether it is necessary to reconstruct the object tree after acquiring the read lock. In some cases it may be more efficient for the editor to hold the read lock while the user is performing several operations on the same unit; but, in general the editor should release read locks as soon as possible. Obviously, if the editor is intentionally providing exclusive access (at user request), then the editor must hold the read lock; however, the editor-writer should be aware of the consequences of holding read locks and should make the user aware if appropriate.

In some situations the editor may want to acquire a copy of the unit being edited (which is relatively inexpensive), but the standard mode should be to share permanent (read-only) trees. This maximizes sharing and minimizes the calls to the space manager to spawn phantoms (calling the space manager is not quite free).

INSTALLING AND ELABORATING DECLARATIONS

The interface for installing declarations involves two discrete steps. First the editor establishes insertion points in the installed package, using the Open_Insertion_Point procedure in Directory_Operations. Adding insertion points does require overwrite access, and will be serialized with respect to other readers and writers. A sequence containing the set of insertion points must be constructed for use by the install procedure. All of the insertion points must be in a single unit.

Next a sequence of declarations to be installed must be constructed. Declarations to be inserted must be source declarations. The Install procedure takes the list of insertion points and a list of declarations to be inserted at those points, returning a list of inserted declarations and status and error information. The installation of declarations changes the Diana tree, replacing insertion points by the inserted declarations. The editor must update the object tree correctly to reflect these changes.

This protocol for installation is designed to supported arbitrarily complicated multi-part declarations. There is a simpler interface for inserting a single declaration, but the editor does not currently use that interface.

Installing a subunit requires both the subunit and its corresponding stub_id. The editor must locate both of these Diana trees, get an Action.Id if appropriate, and then call Directory_Operations. The installed tree becomes a committed, permanent tree that is no longer consistent with the editor's object tree and reflects any transformations performed by semantic analysis. The editor must reconstruct the object tree if the user wishes to view the installed tree.

Elaborating declarations is straightforward. The editor must call Directory_Operations.Elaborate with an installed Def_Id. Status is returned to allow the editor to determine whether the elaboration was successful. The Diana tree is not modified in any way which would make it inconsistent with the Editor's object tree.

WITHDRAWING AND DELETING DECLARATIONS

Withdrawing an elaborated declaration involves passing the Def_Id to Withdraw_Elaborated_Declaration, along with an indication of whether the declaration is to become installed or source. Withdrawing installed (but not elaborated) declarations is similar, except the result is always a source declaration.

In the case where a delaration becomes source, the Diana tree for the containing unit will have been modified to remove that declaration. In this case the editor may have to reconstruct the object tree, or at least reflect that the withdrawn declaration is no longer in place.

Deleting a declaration first withdraws it and then actually deletes the declaration. The object tree must be updated accordingly.

SOURCE DECLARATIONS

Source declarations require somewhat special treatment. By definition, a source declaration will not be semantically consistent. However, the user would still like the directory system to keep track of source declarations, particularly source subunits. Since there is no installed declaration to represent these declartions, they will be represented as an insertion point in the parent with an associated but seperate Ada object for the source declaration. The editor displays this declaration in a separate window, with a prompt at the insertion point.

5. FILE DIRECTORY

For the moment, just look at the visible parts on <package_directory>. In particular, see V_FILE_DIR.ADA and V_POLY_IO.ADA.

6. DEPENDENCY DATA BASE

SYSTEM SYNCHRONIZATION REQUIREMENTS

It is possible to compute all dependency information directly from Diana trees; however, performance considerations, synchronization issues and implementation complexity all argue for keeping a central data base which augments the Diana tree. The dependency data base described here is designed to provide all the information required by change analysis to efficiently compute obsolescence information. This data base can be used by other tools for efficiently determining dependencies between units (for example, this data base allows a very inexpensive implementation of the queury "what are the installed using occurrences of FOO?").

The primary consideration here is the synchronization of updates to the package directory system. At any given time there are likely to be several invocations of semantic analysis. Semantic analysis potentially needs access to a large part of the universe, since programs, commands and declarations all view the package directory as the context for compilation and execution. In order for semantic analysis to operate correctly there can be no (apparent) changes to the environment. This places very severe serialization restrictions on all operations in the environment. Strategies based on having semantic analysis determine a minimal context and acquire read locks on that environment appear to introduce considerable complexity and not really solve the problem.

To minimize the serialization of operations, the system is structured so that semantic analysis need acquire no locks and may always proceed concurrently with other operations. The only serialization that impacts semantic analysis is that updates to the dependency data base are serialized. Semantic analysis is responsible for updating the data base every time a new dependency is introduced.

Thus, key points in semantic analysis are serialized with respect to changes in the universe, because updating and querying the dependency data base are serialized. Change analysis queries the dependency data base to determine which units and actions would be obsolesced by a given change. All the queries to analyze a given change are performed atomically. Change analysis guarantees that no change will be made which would obsolesce currently installed units or which would interfere with in-progress actions.

Since semantic analysis and change analysis are serialized by their competion for access to the central dependency data base, they need perform no other synchronization. Each can proceed without acquiring read locks on any part of the universe. This allows efficient traversal of the Diana trees, without the overhead of calling the object management system to open every unit visited. This is particularly nice in light of the fact that it turns out to be rather difficult (impossible?) to efficiently determine which units would need to be opened.

OBJECT VERSUS ACTION DEPENDENCIES

Dependencies may be object dependencies or action dependencies. Object dependencies are dependencies between a defining occurrence and an object which would be obsolesced if the defining occurrence is changed. Object dependencies are only in force as long as the object is installed or elaborated or is open for overwrite or update.

Action dependencies are dependencies between a defining occurrence and an Action.Id representing an action which would be invalidated if the defining occurrence changes. Action dependencies are no longer in force once the action has been committed or abandoned. The permanent effect of any action is reflected in object dependencies that were established during the course of the action.

It would be possible to keep dependencies of finer granularity than simply from defining occurrence to entire Ada objects. However the resulting data base would be unmanageable, both from the point of view of raw size and from the point of view of the amount of work involved in keeping the data base consistent with a changing universe. Change analysis can use this information along with the information in the Diana tree to compute finer grained dependencies. This design is based on the assumption that searching all the units in some subtree of the system is prohibitively expensive (poor locality, massive IO and CPU cost), but only traversing exactly the units involved is acceptable.

There are two object dependencies (Definition and Namesake) and one action dependency (Visibility) introduced by semantic analysis and maintained by the dependency data base. Initially, only semantic dependencies need be implemented.

DEFINITION DEPENDENCY

The Definition dependency is simply a dependency between a defining occurrence of an entity and any unit containing a using occurrence of that entity. Conceptually, this dependency is the inverse of the Diana SM_DEFN attribute. Since the dependency information only goes to the granularity of entire units, full implementation of REVERSE_SM_DEFN (or NFED_MS) would involve making a pass over all the units in the Definition dependency to find the actual using occurrences.

Semantic analysis records this information whenever it sets an SM_DEFN. Because of the cost of serializing updates to the data base, eventually some optimization will be required. It can be expected that any single object will quite likely have several references to the same defining occurrence (particularly defining occurrences for packages, types, and operators). Either the operation to update the data base should be able to (cheaply) determine that a dependency already exists and return immediately, or semantic analysis should encache the set of dependencies it has established, and only call the data base to reflect new dependencies. The former approach seems nicer, since it requires no additional data structures, and it is likely that (at least at some level) it is possible to allow such benign readers to proceed concurrently with "real" operations.

Change analysis uses this information to determine which units would be obsolesced by deleting or changing some defining occurrence. This is described in Section 8 below.

NAMESAKE DEPENDENCY

The Namesake dependency is used to analyze dependencies involving overloading, and is less straightforward than the Definition dependency. With every overloadable entity, O, we can identify its initial namesake, N. The initial namesake, N, is the first non-overloadable entity with the same identifier encountered when moving back (through all declarations in all enclosing scopes) from O to the root of the universe. If there is no such N between O and the root, we manufacture an a special Root_Namesake to serve as the (implicit) inital namesake for O and its namesakes. The Namesake dependency is a dependency from every N (explicit or implicit) to all units containing some O, such that N is the initial namesake for O.

Semantic analysis records this dependency everytime it encounters a defining occurrence for an overloadable entity. It would appear that determining the initial namesake and recording the dependency needs to be implemented as an atomic action; however, the atomic nature of the Get_All_Ids operation (discussed in Section 7) eliminates the need to make this atomic.

Some optimization may be in order, but it is unlikely to be as important as in the case of the Definition dependency, since there are many more repeated using occurrences than there are repeated definitions of overloadable entities.

Change analysis uses this information to analyze the interaction of changes with overloading. See Section 8.

VISIBILITY DEPENDENCY

The Visibility dependency is an action dependency rather than an object dependency, and is used to ensure that the context for compilation does not change (too much) during the course of a single action.

Complete understanding of this dependency requires an understanding of the incremental symbol table described in Section 7. Basically, the first time semantic analysis must identify some symbol, it must determine the set of all visible (defining) occurrences of this symbol. This operation must be atomic, and is viewed as an operation on the dependency data base to lock out all modifications to the universe while this set is being computed. Since the occurrences in this set determine the context for the rest of the analysis, the Action on whose behalf semantic analysis is being performed is now dependent on the continued existence of all of these (and no more) visible occurrences.

The Visibility dependency is represented as a dependency between a context (declarative region) and an <identifier, action> pair. Installing or withdrawing a declaration with the indicated identifier in the specified context will obsolesce the action if it is still in progress. Change analysis uses this dependency when changing visible occurrences or introducing new visible occurrences to know which (in progress) actions would be affectd.

DATA BASE IMPLEMENTATION

The visible part of Dependency_Data_Base can be found in [semantics]dependency_db.ada. The visible part has two major subpackages, Establish_Dependency and Query_Dependency, used by semantic analysis and change analysis, respectively. Initially, both of these subunits should be separate architectural packages visible only to their single special client. Eventually some of the queries should be packaged for use by higher level tools. The use of these two visible subpackages is described in detail in Sections 7 and 8. Here we will concern ourselves with the related issues of synchronization, permanence and storage managemenet.

...

7. INCREMENTAL SEMANTIC ANALYSIS

INTERFACE AND CONTEXT

The incremental interfaces to semantic analysis can be found in [semantics]v_semantics.ada and [semantics]v_errors.ada. The incremental semantics can be viewed as a set of utility procedurea which attribute a tree and return error messages. The client must acquire some form of write access to the unit containing the tree being attributed, and must set the Diana Current_Default unit to that unit. The client must also provide a heap (not currently implemented) for semantic analysis to use for storage. Semantic analysis generates a considerable amount of garbage of global types besides Diana.

A primary issue in interfacing to semantic analysis is that of providing enough information to allow straightforward computation of the compilation environment. Each visible interface addresses this problem in a slightly different way.

The first interface solves the context problem by taking a rooted tree. Semantic analysis can compute the context by following backpointers and traversing the structural tree back to a COMP_UNIT and then using semantic attributes on the designator to find parent units. This form is used for installing declarations in place.

The second interface analyzes an unrooted tree, but requires and insertion point representing the context for compilation. The unrooted tree will be processed as if it were in the position of the insertion point. This interface is used for checking semantic consistency without actually acquiring an overwrite lock and copying the tree into its context.

The third entry is for semanticizing subunits. The designator for the stub declaration corresponding to the subunit provides the necessary context. The entire subunit is analyzed, and appropriate attributes set to refer to the parent units.

The Resolve_Name entry analyzes an unrooted tree representing a name in the context of some declarative part. A DECL_S or ITEM_S provides the context, and the name is interpreted as if it appeared somewhere after the end of that declaritive part. This entry is typically used for name resolution, taking advantage of the semantic attributes to determine the object denoted by a name.

DEPENDENCY DATA

All entries to semantic analysis include an object id and an action id to be used in establishing dependencies. For casual or "approximate" semantics it is not necessary to maintain the dependencies, in which case a null object id and null action id can be used. Only when semantics is used in preparation for installing a declaration should dependencies be maintained.

The three dependencies discussed in Section 7 are fairly easy for semantic analysis to maintain. The interface between semantic analysis and the package Dependency_Data_Base is defined in the Establish_Dependency visible subpackage.

The Definition dependency is maintained simply by calling the Definition procedure with the Def_Id and the dependent object id every time an SM_DEFN is set.

The Namesake dependency is maintained when processing the defining occurrence for an overloadable entity. Semantic analysis first computes the initial namesake (which may be Diana.Empty if there is no initial namesake) and then calls the Namesake procedure with the object id (for the unit containing the overloadable entity) and the initial namesake.

The first time semantic analysis must identify some symbol, it must determine the set of all visible defining occurrences of that symbol. While it is doing this, there can be no changes in the environment. Once completed, the action is dependent on there being no changes in visibility which would invalidate the results of that determination. The interface for establishing the Visibility dependency is designed to address these synchronization needs, while allowing the knowledge of Ada visibility to remain in the semanticist, rather than in the dependency data base.

Essentially an open/close protocol is used in the Visibility dependency interface. Semantic analysis must first perform the Open_Visibility operation. This locks out other calls to the dependency data base, ensuring that the environment will not be modified. Then the set of visible occurrences can be computed. As semantic analysis is doing this computation it should call Establish_Dependency.Visibility (with the correct Action.Id) for each context (scope) that it visits, introducing a Visibility dependency. When done, semantic analysis must perform the Close_Visibility operation. An Open_Visibility which is never closed will time out, and the dependent action will be abandoned.

INCREMENTAL SYMBOL TABLE

INCREMENTAL ANALYSIS ALGORITHMS

8. CHANGE ANALYSIS

REQUIREMENTS

At all times the committed (permanent) form of the installed universe will be consistent according to Ada semantics. From the point of view of change analysis, there are two ways to change the universe -- a new declaration may be installed, or a declaration may be withdrawn. In the case of installation change analysis is invoked with the new (fully attributed) declaration in place in the installed parent tree. For withdrawal, change analysis is called with the victim before the declaration has been removed from the installed parent tree.

Clearly a simple and very conservative implementation of change analysis would be to apply the straightforward set of obsolescence rules defined by Chapter 10 of the Ada manual. Then every change to a compilation unit would obsolesce that entire unit, all subunits of that unit, and any unit which had visibility to the unit. This is suboptimal but acceptable in a conventional batch system. On the R1000 this would be disasterous, since the directory system on the R1000 is based on a set of installed and elaborated packages. The analogous situation on a conventional file system would be that adding a new object (file) would remove everything else in that directory and all subdirectories.

Both to reduce the amount of recompilation and to make the directory system work, we require that change analysis be much less conservative. Ideally, change analysis would only obsolesce exactly those declarations that are inconsistent with the proposed change. To simplify implementation we do not strive for the ideal, but for a compromise which allows most of the freedom of a conventional directory system and prevents program recompilation in many cases.

Initially change analysis will examine changes down to the granularity of inividual declarations, but will obsolesce entire units. The set of obsolescence rules given here is known to be conservative, and should be improved over time to allow more changes without obsolescing other units. In particular, it should be possible to reduce the granularity of obsolescence to single declarations.

VISIBLE INTERFACE TO CHANGE ANALYSIS

There are basically two operations provided by change analysis, checking an installation and checking a withdrawal or deletion. All operations require that an action id be provided.

There are two forms to the entries in the visible part of change analysis. One form simply returns status indicating that at least one unit would be obsoleted by the proposed change. In the case where there are dependency errors, this form is more efficient since it abandons processing as soon as the first dependency error occurs.

The second form actually returns the set of potentially obsolete units. In this form, change analysis must guarantee that the set it returns covers every unit that would actually be obsolete; however, it need not (and probably can not) guarantee that this set is minimal. Basically, change analysis must guarantee that withdrawing the units in the set of potentially obsolesced units and then performing the proposed change results in a universe which is semantically consistent (assuming that there are no other intervening changes).

DEPENDENCY DATA

The interface to the information maintained by the package Dependency_Data_Base is defined by the Query_Dependency visible subpackage. All of the processing associated with a single change must be atomic (at least for me to understand it), so an open/close protocol is used. The "actor" performing the change is identified by an action.id. Once an actor has called open to initiate a change, no updates may be made to the data base and only that actor may query the data base. The open should be done before calling change analysis, and the close after the change has been saved or it has been decided that the change is illegal.

There is a single entry to query the Definition dependency. The function Definition_Dependents takes a Def_id and returns the set of dependent objects.

There are two entries for querying Namesake dependency information. The first one takes an explicit initial namesake and returns the set of overloadable entities with that initial namesake. The second form is used when there is no explicit initial namesake. The symbolic form of the name (Diana.Symbol_Rep) is passed and used to find the Root_Namesake and return the correct Namesake_Set.

The entry Visibility_Dependents returns the set of actions that would be impacted by inserting or deleting a declaration in some context.

WITHDRAWING DECLARATIONS

Withdrawing a declaration should only obsolesce those units which depend on that declaration, not everything that depends on anything in the same declarative part. The impact of withdrawing a declaration may be defined in terms of withdrawing other declarations.

By passing the symbol_rep of the identifier or designator for the declaration and the enclosing context (declarative part) to Visibility_Dependents, the set of actions dependent on the entity being withdrawn can be determined.

For a non-overloadable entity all Definition dependent units must be withdrawn before that entity can be withdrawn. A non-overloadable entity may also serve as an initial namesake and if the set of namesakes is non-empty, all of the definition dependents of the namesakes must also be withdrawn.

For an overloadable entity, change analysis must find the initial namesake or determine that there is no initial namesake, and then call the appropriate Namesake function. The set of namesakes returned (which will include the original entity) must be withdrawn.

If the withdrawn declaration is a package, withdrawing the package has the effect of withdrawing all of the declarations of the package. Thus change analysis must be applied recursively. Withdrawing a program unit always requires withdrawing all subunits.

INSTALLING DECLARATIONS

When a declaration is installed it obsolesces any actions that are returned by calling Visibility dependents with the symbol for the declaration and the enclosing context.

When a declaration is installed it can obsolesce other declarations by either hiding declarations that were previously visible, or by changing the results of overload resolution in some way.

A non-overloadable entity may hide other declarations. Change analysis must first compute the set of hidden declarations. From the point of view of all units "below" the hiding declaration, inserting a hiding declaration is equivalent to withdrawing the hidden declarations. For each hidden declaration, change analysis must compute the intersection of the set of units "below" the declaration being installed and the set of units which would be obsolesced by withdrawing the hidden declaration.

Implementation note: The poor locality and computational expense of propogating obsolescence prohibits the straightforward approach of actually computing sets and intersecting them. The "below-ness" property should be used to prune the search early, rather than actually computing the full effect of withdrawing the hidden declarations.

There are several ways that an installation can impact overload resolution. The installed entity can itself be overloadable, in which case anything dependent on any of its namesakes is obsolete. As discussed above, change analysis must compute the namesake set, and then take the union of the definition dependents for each member of the namesake set. The resulting set is the set of obsolete units.

If the installed entity is not overloadable, then the only issue is whether the installed entity divides up a previously contiguous namesake region. This can be determined by finding the initial namesake (even though this is not an overloadable entity), getting the namesake set, and then determining which members of that set are "before" the new declaration, and which are "after". For all that are after, take the union of their definition dependents to get the obsolete units.