The Delta Delta

Revision 1.01
April 21, 1986

This document discusses the new features of the Delta release of the Rational Environment.

Table of Contents

List of Figures

List of Tables

1. Delta Requirements and Goals

The Delta release is a disk-incompatible revision of the Rational Environment whose user interface is consistent with the existing Gamma release. Its intent is to provide significant enhancements in system performance, particularly in the editor and in the subsystem tools, and to provide access control, configuration management, and version control facilities for the first time in a Rational product. The release must be ready for production in late 1986 and must not divert development resources from the design and implementation of the Epsilon Release.

1.1. Editor

The design goal for the Delta Ada object editor is, as much as possible, to keep the functionality of the Gamma editor, but with increased performance. In particular the time needed to open an Ada unit should be bounded by the time it takes to create a new window and fill it with text. We believe we can meet this performance goal with only minor reductions in functionality; the only major feature that will be dropped is elision.

In the Delta system, the Rational Editor should perform as efficiently as a traditional text editor. The goal for Delta is to improve editor performance by at least a factor of two.

1.2. Access Control

The Delta system will provide rudimentary access control for all directory objects, based on user names and access lists. An object editor will be provided for access lists.

1.3. Code Generation

In the Delta system, the R1000 Code Generator will support incremental insertions into coded specs. This capability will prevent clients from being obsolesced when certain kinds of upward compatible changes are made.

The code generator will support compatibility between different views with respect to incremental changes.

The code generator will also have significantly increased loader speed.

1.4. Code Archive

The goal of `Code_Archive` is to support the following operations, without the performance cost of recompilation from source.

- Copy a subsystem (Spec and Load Views) to another machine. Once the

subsystem is copied, it is possible to compile and run other programs that depend on it. The copied subsystem cannot be debugged.

- Copy a Load View of a subsystem to another machine. The copied Load View can be used with compatible Spec Views on that machine.

- Copy a main program (whether it's in a subsystem or not), its load information (elaboration code segment), and the closure of units it depends on, to another machine. The copied program can be executed but not debugged.

1.5. Subsystems Tools

In the Delta system, Rational Subsystems(tm) should be comparable in features and performance to the Vax(tm) Ada development capabilities. Specifically, the following operations should be efficient enough for common use in developing software on an R1000 by teams of developers:

CHANGING BODIES
: Spawn a new Load View and make changes that do not affect the corresponding Spec View. Code the changes and test the full system with the changes. Release the changes for other developers to use.

CHANGING SPECS
: In a Load View, make changes to a spec that has coded dependents. Test and release the changes to other developers.

EXPORTING COMPATIBLE CHANGES
: Propagate compatible spec changes from a Load View to a Spec View that has coded dependent Views. Build a version of the system that uses the changed specs. Test and release it.

EXPORTING INCOMPATIBLE CHANGES
: Propagate incompatible spec changes from a Load View to a Spec View, creating a new Spec View if needed. Recompile clients that are affected by the spec changes, creating new Views as needed. Build a version of the system with the new modules. Test and release it.

DISTRIBUTING RELEASES
: Move a View to another machine with enough information to allow compilation and debugging of programs that import the View. Debugging of the moved View is not necessary.

These scenarios reflect those that our existing and potential customers are believed to be most sensitive to. To quote from **[BLB.PDT.ENV]Vax_Comparison.txt**:

"... Historically we have done very well in the single developer areas. Thus the focus is in the large project area. ... Currently there are some situations where customers may not be able to use Rational subsystems to decompose their systems. [For example, mutual dependencies across subsystem interfaces.]

"The team development scenarios assume that there is an existing system (a frozen baseline) that must be modified. Thus, the first activity in all cases is to spawn a "workspace" where the changes can be made, without impacting the ability to execute the baseline. The single developer scenarios assume that the developer is working alone and does not have to worry about impacting the work of others.

"The scenarios are based on the assumption that the systems that customers build consist of multiple program libraries. These program libraries are not necessarily hierarchically decomposed. These libraries do, however, export a set of abstract interfaces represented as package or subprogram specs. These abstract interfaces will be referred to as layer interfaces [analogs of our Spec Views]. The implementations of these libraries will be referred to as layer implementations [analogs of our Load Views]. Note that most of our early customers will not be using abstraction or private types to any great extent."

On the Vax, each "layer" would be compiled into a *program library*. Within the program library, the layer interface units are indistinguishable from the layer implementation units. Imports from layer to layer are accomplished by *entering* in a program library links from the importing library to each of the imported units. The links are similar to a Rational Link Pack, except that all units of the *execution* closure of the imported unit must have a link. When the target of a link is recompiled, the referencing library becomes obsolete; the link must be *reentered* by the importer and then the program library must be recompiled.

A "workspace" is called a *sublibrary* and is the analog of a View of a Rational Subsystem. A significant difference is that sublibraries are differentials from the main program library.

On the Vax, the scenarios are performed as follows:

CHANGING BODIES

Create a (differential) sublibrary; check out the source from CMS (the Vax configuration management system); edit the source file for the body; compile into the sublibrary; link the entire system. Test by recompiling the main program into the sublibrary, or check the body back into CMS, merge the body changes back into the main library and then test. Merging changes back to the program library can be done automatically even in the face of several developers all making concurrent changes in different sublibraries. Only in the worst cases is any recompilation required.

CHANGING SPECS

Basically the same as CHANGING BODIES. In this case, the changed spec is compiled into the sublibrary and the transitive compilation closure of units that are obsolesced are also compiled into the sublibrary.

EXPORTING COMPATIBLE CHANGES and EXPORTING INCOMPATIBLE CHANGES

Implementation of the changes proceeds basically as in CHANGING SPECS. When the changes are reintegrated, all links from the client are obsolesced. They have to be re-established in each client manually (one reenter operation per client).

For review, in the Gamma System these operations are performed as follows:

CHANGING BODIES

Spawn a new Load View for the subsystem; edit the body; compile all units in the new Load View; link the entire system. Reintegration is not required if the new Load View is on a primary development path (unlikely in most customer situations). Tedious manual reintegration steps are necessary if several developers are making changes in the subsystem simultaneously.

CHANGING SPECS

Spawn a new Load View and make the spec change; recompile everything in the view.

Unlike the Vax, you have to copy all the units in the view and recompile them, not just those obsolesced. Because the spec is in the implementation of the subsystem, as in the Vax case with the same design, the spec change will not obsolesce any unit outside of the library containing the changed spec. Reintegration problems exist as in CHANGING BODIES.

EXPORTING COMPATIBLE CHANGES and EXPORTING INCOMPATIBLE CHANGES

First a new Load View needs to be spawned as in CHANGING BODIES. The spec is then changed and then a new Spec View is spawned. Finally, all of the views that import the Spec View that was changed need to be spawned now importing the new Spec View with the change in it. All of this stuff must be recompiled and relinked. The net effect is that a copy and recompile are performed for all of the units in the subsystem, the Spec View for the subsystem, and the transitive closure of the views (spec and load) that import the subsystem changed.

Recoding cannot be done automatically over the "activity"; the developer must supply the correct wildcards.

In summary, the following problems with the Gamma subsystem tools must be overcome in the Delta release of the Rational Environment:

- *Excessive space and time are required to copy all units to a new view and then recompile them.*

- *Tedious manual reintegration steps are necessary if several developers are making changes in the subsystem simultaneously.*

- *Another tedious manual process is necessary to spawn and import the transitive closure of views that import the changed Spec View.*

1.5.1. Configuration Management and Version Control

Configuration Management is the process used to construct, release, and maintain multiple consistent views of a subsystem. Delta provides a mechanism to control the various transformations that occur with different projects at different stages in the lifecycle.

Version control is the process of controlling and tracking changes that occur within a single unit throughout its life. This includes control over which versions can be changed, who can change them, and the recording of what and why the versions were changed.

2. Access Control

2.1. Access Control

Access control restricts the operations that can be performed on objects. The entities that are to be restricted are jobs and users directly executing operations.

Delta access control is intended to be a subset of Epsilon access control.

The specification of access control consists of definitions of

1. The subjects users that are to be restricted, and their executing agents (jobs).

2. The objects to which access is to be controlled.

3. The types of access that are allowed and their semantics.

4. The mechanics of access checks, and the implementation details of access control.

5. The exceptions to the access rules taken advantage of by Rational tools.

2.1.1. Resolution of Recent Issues

1. Make consistent with Epsilon revisions. Added Create access and changed definitions of Write, Owner, and Delete access.

2. Eliminate possibility of creation of dependents locking up objects because the user has access to the original object but can't demote the dependent. Changed implementation strategy to remove access controls to trees.

3. Operator capability for operations in packages Job, Queue, Scheduler. Added operations to list of controlled operations.

4. Restrictions on Ada units independent of compilation closure of unit (and execution closure of unit). Change for locking solves this.

5. Restrictions on link pack changes, switch association changes, and freeze/Unfreeze changes. Added explicit tests in these cases to implement restrictions.

6. Access control on pipes, terminals, etc. Consider implementing if time permits. No major issues.

2.1.2. Subjects To Be Restricted

The entity in Delta that executes operations that are subject to control is called a job. In some cases, users at terminals execute operations that are not thought of as being in a specific "job"; for purposes of access control, these are considered to be executed by an implicit job started by the user.

An executing job has associated with it the identity of the user that started it. This user identity is used in determining the allowed access.

Each user has a user name and is a member of one or more groups (each user is at least in a singleton group that has the same name as the user). An access list associated with each object grants specific kinds of access to specific groups. For a job to be granted access to an object, its user identity must be a member of a group that is listed on the object's access list.

There are several distinguished groups that are predefined:

1. **Public**. All users on a machine are members of the group Public.

2. **Network Public**. All users on the Rational Internet are members of this group. This is not really used in Delta.

3. **Privileged**. Members of this group are not restricted by any access controls. All operations and objects can be accessed. The users Operator and Rational are, by default, members of this group.

2.1.3. Access Control Lists

An *Access Control List* (or *ACL* for short) is associated with each object. There is an access control list for each version of each object.

When new versions are created, they inherit the ACL of the previous version. When new objects are created, their ACL is set to a default ACL associated with the containing world. The ACL can also be explicitly set.

The ACL lists group names and the classes of access that each listed group is allowed. If not explicitly listed, a group is granted no access.

A job (based on its user identity) is granted access to an object if there is an entry on the ACL for the object that lists a group that the user is a member of and access of the type required by the job.

2.1.4. Objects

Each directory, file, and Ada unit has associated with it an *access control list*, which lists group names and the type of access they are allowed. Only these directory objects can have access controlled. There are a few exceptions to this: certain environment operations are also controlled via access control lists. These operations are controlled by the access control list on some defined set of objects stored in the universe. Thus, all controls are based on access controls to directory objects.

Link packs have their access controlled based on special rules noted below.

Access to the code database object of a world is similarly controlled.[1] Devices, users, sessions, groups, pipes, and code segments do not have any access control. Devices and pipes will be considered as time permits. Read and Write access would control the read and write operations for pipes and for terminals. Proabaly, a job must have read or write access to a terminal in order to open it.

An access control list entry with the special group name **Public** describes the access allowed to all users (all users are members of this group).

Each world has a special kind of access called "Owner" access. A user who has Owner access to a world is allowed to change the ACLs of objects within the world. When a world is created, the creating user is given Owner access to the world.

When an Ada or file object is created, it is assigned an access list based on a default ACL associated with the containing world. Note that this applies to new objects that are created; new versions of objects inherit the ACL from the previous version.

When a world is created, the creating user is given Owner access and the access control list is set to be the same as the access list of the containing world (not the default ACL for new objects used in the non-world cases).

2.1.5. Access Types

There are 5 different types of access in Delta:

2.1.5.1. Read Access

Intuitively, Read access is required when a user (or program) is to inspect the current state of an object. This includes things like executing Definition, opening for mode In, and executing Display or Definition in the debugger.

---

[1] The exact details here are pending.

2.1.5.2. Write Access

Write access covers operations that change the value of an object. This includes things like Edit and Promote/Demote.[2]

Write access to an object also controls the ability to delete it. This is done in operations such as Object-D and Compilation.Destroy.

2.1.5.3. Create Access

For worlds, Create access controls the ability to create new objects in the world. A job must have Create access to a world to create objects anywhere in it.

Create access applies only to worlds and is represented by the Write access type for the world.

2.1.5.4. Delete Access

Delete access is required to delete a world. It applies only to worlds.

2.1.5.5. Owner Access

Owner access is used only for Worlds and does not apply to other objects.

Owner access to a world is required to change the access list of an object in that world. Owner access also controls the ability to change the links in a world, the compiler switch file associations in a world, and the ability to freeze and unfreeze objects in the world.

Informally, we say a user has Owner access to an object if the user is an owner of the object's containing world.

Owner access is represented by the Read access type for a world.

2.1.5.6. Cross Reference of Access Types

Non-library objects (other than devices, pipes, and code segments) have access types Read and Write.

Worlds have access types Create (represented by Write), Delete, and Owner (represented by Read).

Directories have no access restrictions of their own. All of their restrictions are based on restrictions indicated by the containing world.

---

[2] It is unclear whether Write implies Read.

2.1.5.7. Access That Isn't Controlled

"Execute" access is not controlled in Delta. Any user that can access a coded Ada unit for Read can execute the unit.

In Delta, the diana tree of an object is not access protected. This means that a user with knowledge of and access to Diana tools would be able to read almost any Ada unit in the systems. This is a characteristic of Delta only and does not carryover into Epsilon.

Part of the reason for the non-protection of trees is to allow Environment tools the ability to access and modify trees in the course of operations that are otherwise valid for a user from an access control point of view. Such operations include demoting dependents of a unit that the job executing the demote may not have access to, and executing a program whose closure is not accessible to the job that initiates the execution.

The compiler and other parts of the environment must be modified to explicitly check access in order to prevent promotion or demotion of units that a user could not otherwise access. This approach will tend to reduce the security and reliability of the access control system.[3]

The effect of this is that jobs cannot wontonly demote or promote units to which they do not have Write access, however, a side effect of promoting or demoting a unit to which a job does have access is the change in the state of a unit to which it does not have access.

2.1.6. Operations and the Access They Require

When an operation is attempted by a job that does not have sufficient access to the object, the operation fails. The failure may be indicated by a status parameter or by the raising of the exception **System.Access_Error** is raised if there is no status parameter. **System.Access_Error** is a flavor of **IO_Exceptions.Use_Error**. This means that a handler clause listing System.Access_Error or IO_Exceptions.Use_Error will be executed for this exception. The image value for the exception is "IO_Exceptions.Use_Error (Access error)".

Generally speaking, Read access is required for any operation that examines or displays a object; Write access is required for an operation the modifies a object; Delete access is required for any operation that will destroy an object; Owner access (to the containing world) is required for operations that change access lists associated with an object. For access control purposes, a world is said to contain itself, so owner access to the world is sufficient to change its ACL and default ACL for containing objects.

---

[3]This is done to solve problems such as an inaccessible dependent making it impossible to demote or delete a given unit.

Write (create) access to the containing world is required to create a new object or a new version of an existing object.[4]

2.1.6.1. I/O Operations

I/O Operations make access control tests as follows. **In_File** opens require only Read access. **Inout_File** and **Out_File** opens require Write access.

IO_Exceptions.Use_Error is raised for access failures from Text_Io and related I/O packages.

2.1.6.2. Special Access Controls

There is a special file in **!machine** whose ACLs control availability of certain operations. The file is:

!Machine.Operator_Capability

Successful execution of some of the operator and system maintenance commands requires Write access to **Operator_Capability**. The following tables summarize these requirements.

!Commands.Daemon.
- **Schedule**
- **Quiesce**
- **Cancel**
- **Run**
- **Set_Log_Threshold**
- **Set_Disk_Threshold**
- **Collect**
- **Set_Priority**
- **Snapshot_Warning_Message**
- **Snapshot_Start_Message**
- **Snapshot_Finish_Message**

!Commands.Operator
- **Create_User**
- **Delete_User**
- **Enable_Terminal**
- **Disable_Terminal**
- **Force_Logoff**
- **Shutdown**
- **Set_System_Time**

---

[4]This means that you can't expect to use environment editing features with Write access to the object but not to the containing world.

```
    - Internal_System_Diagnosis
    - Shutdown_Warning
    - Archive_on_Shutdown
    - Cancel_Shutdown
    - Limit_Login
    - Limit_Background

!Commands.System_Backup
    - all operations need Write access

!Commands.Terminal
    - all operations need Write access

!Commands.Group
    - Creating and changing groups

!Commands.Job
    - Kill (on session for a different user)
    - Disable (on session for a different user)
    - Enable (on session for a different user)

!Commands.Queue
    - Create
    - Destroy
    - Default
    - Enable
    - Disable
    - Register
    - Unregister
    - Remove
    - Kill_Print_Spooler
    - Restart_Print_Spooler

!Commands.Scheduler
    - Disable
    - Enable
    - Set_Cpu_Priority
    - Enable_Scheduling
    - Disable_Scheduling
    - Set
    - Set_Parameter
    - Set_Job_Class
```

2.1.6.3. Subsystem Tools

Subsystem tools are governed by the same access control rules applying to other jobs. This means that if an object is accessed by a job running the subsystem tools, that access is governed by the normal access control rules for that job.

Setting access lists on objects in a subsystem and then executing subsystem tool operations may cause those operations to fail in strange ways. This statement is true of any operation, of course, but customers may think of the subsystem tools as part of the system rather than as some random program.

Subsystem tools will probably have to modified to recover from access control errors gracefully.

2.1.6.4. Other Special Case Access Checks

Adding or deleting links in a world requires Owner access to the world. This is checked by an explicit check in the Links commands.

Switch file associations also require Owner access to the world in which the associations are being made. This is implemented by an explicitly coded check in the switch commands.

The Freeze and Unfreeze operations are also specially controlled. Owner access to the world containing the objects to be frozen or unfrozen is required to successfully execute the operation. The check is explicitly implemented in the freeze and unfreeze commands.

2.1.7. Editing Access Lists

The Library Object Editor provides interactive display and editing of access lists.[5]

When a particular explanation level is specified, access lists are displayed. To edit an access list, the display of the access list is selected and the edit key is pressed. This creates a command window of the form:

```
Set_Acl (ACL => "<current access list>", Object => "<selection>");
```

The user can then edit the string parameter and commit the command.

Alternatively, an in-place editing operation could be provided. Other issues include the ability to display the access list of a single object versus all in the directory.

---

[5]This feature is expendable.

```
package Access_List_Tools is
    -- This package provides program level access control operations.

    subtype Access_Class is string;  -- of only the following characters:
    Read   : constant character := 'R';  -- objects only
    Write  : constant character := 'W';  -- objects only
    Delete : constant character := 'D';  -- worlds only
    Create : constant character := 'C';  -- worlds only
    Owner  : constant character := 'O';  -- worlds only

    subtype User_Name is string;
    subtype Group_Name is string;
    -- A user names is the simple string name of the user.

    subtype Object_Directory_Name is string;
    -- An object string name is as defined by the directory
    -- package

    subtype Access_List_Rep is string;
    -- A string representation an access list has the following syntax:
    --
    --  Acl       ::= Acl_Entry [',' Acl_Entry]*
    --  Acl_Entry ::= Group '=>' Access
    --  Group     ::= Identifier
    --  Access    ::= Acc_Type+
    --  Acc_Type  ::= 'R' | 'W' | 'D' | 'C' | 'O' |
    --                'r' | 'w' | 'd' | 'c' | 'o'
    --
    --  Examples:  "Phil => R , TRW:RW",  "Public=>COD"

    Access_Tools_Error : exception;  -- Raised by functions
```

Figure 2-1: Access List Tools Package - Types

2.1.8. Archive and Reloading of Access Control Information

The string form or each object's ACL is saved with the object. The Restore operation has an option of trying to restore the object's original ACL or substituting a new ACL specified as a parameter to the restore operation.

If the original ACL is to be restored, any entries that reference nonexistent groups are removed. The restoring user is always given owner access to a restored world (the right-most ACL entry is deleted if the ACL would be too long after adding Owner access for the restoring user).

If the restore tape is from gamma and contains no ACLs, then the ACL

```
   function  Get (For_Object  : Object_Directory_Name)
                   return Access_List_Rep;
   procedure Get (For_Object : Object_Directory_Name;
                  List       : out Bounded.Variable_String;
                  Status     : in out Simple_Status.Condition);
   procedure Set (For_Object : Object_Directory_Name;
                  To_List    : Access_List_Rep;
                  Status     : in out Simple_Status.Condition);
   procedure Set (For_Object : Object_Id;
                  Act        : Action.Id;
                  To_List    : Access_List_Rep;
                  Status     : in out Simple_Status.Condition);

   -- Get or set the access list for the specified object.
   -- Setting the access list requires "Owner" access.
   -- function Get raises Access_Tools_Error if an error occurs.
   -- The procedure version should be called in that case to get the
   -- actual error information.
   -- ACL for world must be contain only C, O, or D access.  Others
   -- must be only R or W access.

   function Check (Ident   : User_Name := utilities.Current_User;
                   Object  : Object_Id;
                   Desired : Access_Class) return boolean;
   function Check (Ident   : User_Name := utilities.Current_User;
                   Object  : Object_Id;
                   Act     : Action.Id;
                   Desired : Access_Class) return boolean;
   function Check (Ident   : User_Name := utilities.Current_User;
                   Object  : Object_Directory_Name;
                   Desired : Access_Class) return boolean;
   -- Check of the specified user has the indicated access to the
   -- specified object.  Only meaningful for Ada objects, files,
   -- and Directories.

   function Get_Default_ACL
         (For_World : Object_Directory_Name) return Access_List_Rep;
   procedure Get_Default_ACL
         (For_World : Object_Directory_Name;
          List      : out Bounded.Variable_String;
          Status    : in out Simple_Status.Condition);
   procedure Set_Default_ACL
                  (For_World : Object_Directory_Name;
                   To_List   : Access_List_Rep;
                   Status    : in out Simple_Status.Condition);
   -- Get or set the default ACL for new objects created in the specified
   -- world.  Function raises exception if error occurs.
end Access_List_Tools;
```

Figure 2-2: Access List Tools Package - Subprograms

"Public=>RW" is used as the value to be restored with a non-world object, and "Public => COD" is used for restored worlds.

```
package Access_List is
    -- Commands package
    subtype Access_Class is string;
    Read   : constant character := 'R';  -- objects only
    Write  : constant character := 'W';  -- objects only
    Delete : constant character := 'D';  -- worlds only
    Create : constant character := 'C';  -- worlds only
    Owner  : constant character := 'O';  -- worlds only

    subtype User_Name is string;
    subtype Group_Name is string;
    subtype Object_Directory_Name is string;
    subtype Access_List_Rep is string;
```

Figure 2-3: Access List Package - Types

2.1.9. Network Remote Accessors

FTP requires the user to provide a remote login and password. This user identity is used for access control purposes on remote machines.

If an RPC server is established on a machine, its identity is used for access control purposes.[6] RPC servers are started by users and run with the user's identity for access control purposes.

Applications that require general access to objects on a machine must run as Privileged (unless there is a system-wide convention to set ACLs appropriately). This means that using systems the way they are typically used at Rational will require all users to be members of Privileged, or all ACLs to allow all access to all users, as we do now on the 2060 and MVs.

2.1.10. Machine.Initialize

Machine.Initialize runs as user Operator, which should be a member of the Privileged group. Jobs started form it also run with **Operator** identity.

Probably should be a **Program.Run_Job** (prog, user) to solve this problem. This operation would only be executable from a job with identity of a user that is a member of the Privileged group.[7]

[6] This raises interesting questions about Source_Archive.Transfer.

[7] This feature is probably expendable.

```
    procedure Display (For_Object : Object_Directory_Name
                                 := "<cursor>");
    -- Display the access list of the specified object on
    -- Current_Output.  Error message to Current_Error if the
    -- operation fails.

    procedure Set (For_Object : Object_Directory_Name
                          := "<selection>";
                   To_List    : Access_List_Rep);

    -- Get or set the access list for the specified object.
    -- Setting the access list requires "Owner" access.
    -- Any error message is sent to Current_Error.

    procedure Display_Default_ACL
                (For_World : Object_Directory_Name := "<cursor>");
    -- Display the list of owners of the specified world in an output
    -- window.  Error message to Current_Error if the operation fails.

    procedure Set_Default_ACL
                (To_List   : Access_List_Rep;
                 For_World : Object_Directory_Name := "<selection>");

    -- Set the default ACL for the specified world.  Owner access
    -- to the world is required.  The default ACL is given to newly
    -- created objects.
    -- A message is displayed to Current_Error indicating any errors.

end Access_List;
```

Figure 2-4: Access List Package - Subprograms

2.1.11. Implementation

2.1.11.1. Representation and Storage of Access Lists

*** This section not updated consistent with the last pass.

The access list will be stored in the object header.[8]

The segmented heap object manager generic probably needs to include space for the access list. The structure of the list is shown in Figure 2-8. Note that these representations limit an R1000 Delta system to 1024 groups and 7 entries per access list.

[8]This section to be expanded.

```
with Access_List_Tools;
package ACL_Utilities is

    subtype Access_List_Rep is Access_List_Tools.Access_List_Rep;
    subtype Group_Name is string;

    function Remove_Access (From_Acl : Access_List_Rep;
                            By_Group : Group_Name)
                        return Access_List_Rep;
    function Add_Access (To_Acl      : Access_List_Rep;
                         For_Group   : Group_Name;
                         Access_Type : Access_List.Access_Class)
                        return Access_List_Rep;
    -- These operations manipulate the string representation of
    -- access lists.  Remove_Access modifies the ACL to eliminate
    -- any access the By_Group may have.  Add_Access adds Access_Type
    -- for For_Group (unless it's already there).

    -- Examples:  Remove_Access ("Phil=>RW, Jim=>RW", "Phil")
    --              returns "Jim=>RW"
    --            Add_Access ("Public=>R, TRW=>D", "phil", "RW")
    --              returns "Public=>R, TRW=>D, Phil=>RW"

    -- Other such ACL utilities may be added.  These are just
    -- string manipulation routines and could easily be shipped native
    -- (and source).

end ACL_Utilities;
```

Figure 2-5: Access List Utilities Package

2.1.11.2. Checking of Access Restrictions

The access restrictions are tested when the object to be accessed is opened (or equivalently resolved to a Diana pointer at the directory level). This is done at the object management level in the system.

In addition, a number of additional restrictions must be tested in various places in the system. The tests are explicit for the specific operations and not part of an overall mechanism.

The specific places include:

- Link pac operations
- Switch association operations

```
package Group is
    subtype Name is string;
    subtype User_Name is string;

    procedure Add_To_Group (User  : User_Name;
                            Group : Name);
    -- Add the specified user to the specified group.  Operator
    -- privilege is required to execute this operation.
    -- Errors are written to Current_Error.

    procedure Remove_From_Group (User  : User_Name;
                                 Group : Name);
    -- Remove the specified user to the specified group.  Operator
    -- privilege is required to execute this operation.
    -- Errors are written to Current_Error.

    procedure Display (Group : Name);
    -- Display the names of users in the specified group on
    -- Current_Output.

end Group;
```

Figure 2-6: Group Package

- Freeze/Unfreeze operations

- Compilation promote and demote operations (check for explicit access to units that are explicitly referenced; no check for implicitly referenced objects)

2.1.12. Restrictions, Limitations, and Risks

There are a number of restrictions and limitations with the Delta implementation of access control.

1. If editing operations create new versions, Write(Owner) access to the containing world may be required to successfully edit something, not just write access to the object. A related implementation problem may require Write access to the containing library.

2. Since following diana pointers involves opening no objects, once read access is granted to a unit, it is possible to read all units in it closure, independent of their ACLs, using Diana operations.

3. Body to Visible part transitions may also skip ACLs, but visible part to body transitions probably won't.

```
package Group_Tools is
    -- Tool interface to group operations.

    function Is_Member (User  : User_Name;
                        Group : Name) return boolean;
    function Is_Member (User  : Object.Id;
                        Group : Short_Id) return boolean;

    procedure Add_To_Group (User  : User_Name;
                            Group : Name;
                            Status : out Simple_Status.Condition);
    -- Add the specified user to the specified group.  Operator
    -- privilege is required to execute this operation.

    procedure Remove_From_Group (User  : User_Name;
                                 Group : Name;
                                 Status : out Simple_Status.Condition);
    -- Remove the specified user to the specified group.  Operator
    -- privilege is required to execute this operation.

    function Contents (Group : Name) return string;
    -- return the names of users in the specified group.
    -- The list consists of names separated by a single blank.
    -- The null string is returned if the group is empty or
    -- nonexistent or its name is malformed.

end Group_Tools;
```

Figure 2-7: Group Tools Package

4. Because of the low level on the access check, the user level behavior is not very predictable. This can be fixed given enough resources but may require pervasive changes in the system. For example, Write access to directories may be necessary to create and delete objects.

5. Do directories remain Ada units in Delta? If not, ACL for directory needs to be stored somewhere. Answer: yes. so what?

6. Not very secure against penetration efforts.

7. There are several features whose implementation depends on assumptions about how things in the current system work. If these assumptions prove false, then implementation of these features would likely have to be abandoned.

```
subtype Access_Classes is Access_Class range 0..15
-- 3 bits.  A bit vector for RWD access.

Read   : constant := 1;
Write  : constant := 2;
Delete : constant := 4;

subtype Short_Group_Id is long_integer range 0..1023;
-- 10 bits.  Limits machine to 1024 controllable groups.
-- These short ids must be managed and collected.

type ACL_Entry is record        -- 13 bits total
    Group   : Short_Group_Id;
    Allowed : Access_Classes;
end record;

type ACL is array (0..6) of ACL_Entry;  -- 91 bits.
-- Total of 7 ACL entries allows.
```

Figure 2-8: Access Lists

3. Front End Changes

3.1. Editing

The reasons it takes so long for the Gamma editor to open a unit is that it must

1. pretty-print the Diana tree to get the image the user sees, and

2. build a data structure (the notorious "object tree") that lets it quickly map points on the image to corresponding points on the Diana tree.

We do not see any way to make pretty-printing and tree-building fast enough to be acceptable. Our approach, then, is to fix the editor so that the computation is unnecessary. In Delta, what the user thinks of as an "Ada object" will, in fact, be maintained by the Ada object manager as a pair of objects: a Diana tree and a pretty-printed image. The Diana tree will be augmented with enough information so that the editor can quickly find the point in the Diana tree that corresponds to any given point in the image. The principal challenge in getting all of this to work is that the Gamma image and object tree data structures are too large to make permanent with a clear conscience. Most of the work involves changing the editor to do without most of the information in the object tree.

3.1.1. Line Count Attribute

The Ada grammar is factored into two parts that we call the "outer" grammar and the "inner" grammar. The terminals of the outer grammar are start symbols of the inner grammar. Every construct generated by the inner grammar is pretty-printed using an integral number of lines. We call the constructs of the inner grammar "statements" even though they don't correspond exactly to the Ada LRM's notion of statements. By this definition, in the following example:

```
if A > 0 then
    B := 1;
    C := 2;
else
    D := 5;
end if;
```

every line is a statement, but the conditional construct taken as a whole is not a statement.

Delta Diana has a new attribute, **Diana.Lx_Line_Count**. Every node in the tree that corresponds to a construct of the outer grammar has an **Diana.Lx_Line_Count** attribute. The value of the attribute is the number of lines in the pretty-printed image of the construct. Based on HJL's statistics on the distribution of Diana Node_Kinds, roughly 10% of the nodes in a Diana tree will have an **Diana.Lx_Line_Count** attribute. If we assume that the **Diana.Lx_Line_Count** attribute increases the size of a node by about 10%, the total increase in the size of the tree is about 1%.

Suppose the cursor is sitting on an id in a program. Let's see how the editor will locate the corresponding Diana node from the line and column of the cursor. We initialize a variable N to the line number. We then start at the root and work our way down into the tree. As we encounter each subtree, if its **Diana.Lx_Line_Count** is smaller than N, we subtract the attribute from N and skip over the subtree. Otherwise we enter the subtree. We continue this process until we reach a terminal of the outer grammar. At this point, N and the column number will tell us the coordinates of the cursor relative the pretty-printed image of the statement it is in.

3.1.2. Pseudo-Pretty-Printing

Once it has located the statement that contains the cursor, the editor will locate the proper node within the statement using a process we call *pseudo-pretty-printing*. Most of the complexity of the pretty-printer involves deciding where to put the line breaks. The pseudo-pretty-printer is a much simpler program, which, given a Diana tree, figures out the series of tokens in the corresponding Ada text. For each token in the Ada program, the pseudo-pretty-printer outputs the number of characters in the token.

To find the Diana tree corresponding to a cursor position within a statement, the editor starts a pointer at the beginning of the image of the statement and pseudo-pretty-prints the Diana tree for the statement. Each time the pseudo-pretty-printer outputs a token, the editor moves its image pointer ahead by the width of the token, and then it moves it ahead past any white space in the image. When the pointer reaches the cursor, the pseudo-pretty-printer will be at the proper Diana node.

Pseudo-pretty-printing also works in the other direction. Given a Diana node within a statement, we can find the corresponding image on the screen by pseudo-pretty-printing the statement as above, and remembering where the image pointer was just before and after the Diana node was pseudo-pretty-printed. The "grow selection" operation will be implemented with this technique.

3.1.3. Underlining

There are tools (such as **Semantics** and **Show_Usage**) that hand the object editor a node in a Diana tree, and expect the object editor to underline the corresponding place in the image of the tree. This is easy to do with the pseudo-pretty-printer as long as we know what statement the node is in and on what line of the image that statement begins. We can find both of these as long as we know the path from the root of the tree to the node we are supposed to be underlining.

There are two ways of obtaining this path. The first is to start at the node and do **Diana.As_Parent** until we reach the root. We know this can take time proportional to the size of the entire tree. In the case of **Semantics** and **Show_Usage**, the editor is given a list of nodes to underline. Because the list is sorted in traversal order, we can make a

single pass through the tree and find all the nodes not be underlined, and know the path to the root as we encounter each one. This entire list can be processed in a time proportional to the size of the tree. Since the program that produced the list also grinds through the entire tree, maybe nobody will notice how slow the editor is.

3.1.4. Image Objects

The pretty-printed image will be stored in an *image object*, separate from the Ada object that it renders. The data structure is similar to the one used by the Gamma editor to represent editor buffers; a binary tree data structure that implements a "string" abstraction in which "find the N'th element", "insert N items in the middle", and "delete N items from the middle" can all be done in logarithmic time.

The image is stored as a string of lines. Each line has two parts: the text information, and the font information. The text information is stored as count of leading blanks and a bounded string. The bounded strings are obtained from a storage allocator. When a character insertion would exceed the bounds of the string, the old one is given back and a new one with a larger bound is obtained from the manager. The font information is stored as a linked list of places on the line where the font changes, sorted by column number. Since the font implicitly begins in the plain font, the vast majority of the font lists will be empty.

The **Image** subsystem will be modified so that it uses this data structure directly. Thus, the only work required to open an Ada unit for editing, is to open the **Diana** tree, open the image object, and pass a pointer to the image object to the core editor.

3.1.5. Issues

There are some issues associated with keeping the image as a part of the Ada object. The notion of "your view of an Ada object is a function of what your pretty-printer options are" goes out the window. All users see the same image of the Ada object.[9] That image cannot be changed unless the object is opened for writing. This also means that our current implementation of elision will not work. The Delta editor will not support elision.

There is one form of "different users have different images" that cannot be casually brushed aside: the **Show_Usage** command. We cannot require that a user get write access in order to invoke **Show_Usage** on a unit. Moreover, the underlining done by **Show_Usage** should not be left around as part of the permanent state. The solution to this problem is that the core editor will have two places where it keeps font information: temporary and permanent, and will display the union of these fonts. The permanent

[9] Where the pretty-printer gets its options and what it means to change the options for a given Ada unit are still unresolved.

font information is part of the permanent data structure, and can only be changed if the object is open for update. However, when an editor opens an object (for read or update) it also allocates an (initially empty) object in which temporary font information is stored. The **Show_Usage** command will update this temporary structure.

3.2. Diana Changes

A number of changes to the Diana implementation are planned for Delta. The goal of these changes is to improve performance of the system by reducing the size of Diana trees and to correct the minor errors in the Diana implementation before making the massive revisions planned for Epsilon. The changes will also allow a more complete implementation of change analysis. The limited scope of these changes should not destabilize the Diana implementation.

3.2.1. Predefined Operators

In Delta, skeletal operator definitions will not be built for the predefined operators of a type, except for the predefined types in Standard. The Gamma implementation, which builds these skeletal declarations, is not compatible with the formal definition of Diana. Removing these skeletal declarations will make Diana trees smaller and will also reduce the size of the dependency database because no entries will be made for them anymore. To eliminate these skeletal subprogram declarations, the semantic attribute **Diana.Sm_Operator** must be defined for each target, and the semanticist must set it on each reference to a predefined operator.

The proposed set of enumeration identifiers for the values of the **Diana.Sm_Operator** attribute is defined in Appendix I. When a target must use an **Other_...** value, its code generator will have to look at the operands of the operator to determine the correct operator to use.

Each operator in the Standard tree for a target will have an attribute that provides, for that specific target, the **Diana.Sm_Operator** value that the semanticist is to use when setting the **Diana.Sm_Operator** attribute for references to that predefined operator.

3.2.2. Derived Subprograms

The skeletal declarations generated by Gamma for derived subprograms will not be generated in Delta, in accordance with the Diana specification. All references to derived subprograms will designate the subprogram from which the referenced subprogram was ultimately derived. Tools can detect that a derived subprogram is involved by comparing parameter and/or result types of the declaration and reference; they will be different only when an derived subprogram is involved.

3.2.3. Line Count

Table 3-1 lists the kinds of Diana nodes that need a new integer attribute called **Diana.Lx_Line_Count**, which is how many lines are taken up by the pretty-printed image of the subtree rooted at the node. For efficiency, this attribute will be part of the variant record that defines **Diana.Node**. See also Section 3.1.1.

DN_ABORT	DN_EXIT	DN_RETURN
DN_ACCEPT	DN_GENERIC	DN_SELECT
DN_ADDRESS	DN_GOTO	DN_SELECT_CLAUSE
DN_ALTERNATIVE	DN_IF	DN_SELECT_CLAUSE_S
DN_ALTERNATIVE_S	DN_ITEM_S	DN_SIMPLE_REP
DN_ASSIGN	DN_LABELED	DN_STM_S
DN_BLOCK	DN_LOOP	DN_SUBPROGRAM_BODY
DN_CASE	DN_NAMED_STM	DN_SUBPROGRAM_DECL
DN_CODE	DN_NONTERMINAL	DN_SUBTYPE
DN_COMPILATION	DN_NULL_STM	DN_SUBUNIT
DN_COMP_UNIT	DN_NUMBER	DN_TASK_BODY
DN_COND_ENTRY	DN_PACKAGE_BODY	DN_TASK_DECL
DN_CONSTANT	DN_PACKAGE_DECL	DN_TASK_SPEC
DN_CONTEXT	DN_PACKAGE_SPEC	DN_TERMINATE
DN_DECL_S	DN_PRAGMA	DN_TIMED_ENTRY
DN_DEFERRED_CONSTANT	DN_PRAGMA_S	DN_TYPE
DN_DELAY	DN_PROCEDURE_CALL	DN_USE
DN_ENTRY_CALL	DN_RAISE	DN_VAR
DN_EXCEPTION	DN_RECORD_REP	DN_WITH

Table 3-1: Nodes Having **Diana.Lx_Line_Count** Attribute

3.2.4. Image Object

To save pretty-printing time, the pretty-printed image of an Ada object will be kept as a permanent object in the system. The image object contains no Diana pointers and is an attribute space of the associated Ada object. It will be destroyed by the Ada manager when the Ada unit is destroyed. The image object must be opened and closed separately from the associated Ada object. Neither Diana nor the Ada Manager will do this automatically. See also Section 3.1.4.

3.2.5. Quick List Membership Test

Diana.Is_In_List is a new predicate that given a **Diana.Tree** value quickly determines whether or not the designated node is in a structural list.

3.2.6. Attribute Spaces

Ada attribute spaces will be changed to make them smaller and more efficiently relocated by using only the offset portion of the **Diana.Tree** values for the domain of the maps. Iterators over the attribute maps will also be added.

3.2.7. Etceteras

A number of errors and omissions in the Rational Diana implementation will be corrected for Delta:

- **Diana.Sm_Value** shall return **Diana.No_Value** when applied to **Dn_Allocator** and **Dn_Null_Access** nodes.

- **Dn_Used_Name_Id** nodes shall be used to represent *all* references to record component names and to parameter names appearing to the left in a named association.

- References to the functional attributes **'Succ**, **'Pred**, **'Val**, **'Value**, **'Pos**, and **'Image** shall be represented by a **Dn_Function_Call** node whose **Diana.As_Name** is a **Dn_Attribute** node. **Dn_Attribute_Call** nodes shall be used exclusively for **'First(*n*)**, **'Last(*n*)**, **'Length(*n*)**, and **'Range(*n*)**.

- The compilation unit id node for the body of a generic package or subprogram shall be a **Dn_Generic_Id** as required by the *DIANA Reference Manual*.

- A **Dn_Used_Bltn_Op** node shall be used for the reference to a predefined operator in a subprogram rename and in the declaration of a generic formal subprogram.

- Within the body of a task or task type, a reference to the task shall be denoted by a **Dn_Used_Object_Id** since such references are references to the task object.

- The **Diana.Sm_Base_Type** for a range in a choice of an array aggregate shall denote the base type of the index type of the array when such a base type node normally exists; otherwise the base type will be represented by the subtype indication in the declaration of the applicable index type.

- The **Diana.Sm_Constraint** on a slice shall denote either the range of the slice itself or, if only a type mark is given, it shall denote the range of the corresponding subtype if it exists.

A number of other attribute values were questioned by Systeam, but will not be changed

because they all require construction of unrooted Diana trees to represent various base types and constraints. The Rational Diana will continue to designate nodes in the structural tree that will "do" for capturing the correct semantic interpretation.

Time permitting, a number of space and time optimizations will also be implemented:

- The size of structural pointers will be reduced to a size more appropriate for the average-sized Diana tree. An escape mechanism will be provided for extraordinarily long references.

- The **Dn_Used_Name_Id** and **Dn_Used_Object_Id** nodes will be made the same size so that the transformation between them can be done in place. **Diana.Sm_Original_Node** will be eliminated as well and will be replaced by a Boolean that will indicate if the node has been transformed or not.

- To improve the performance of **Diana.As_Parent**, the structural link of every node will point to it's parent. In the case of a node on a list, the parent is the list cell. Each list cell, in turn, will have a structural link to the list header node. To reduce the space impact of this change, the list cells for longer lists (**Dn_Decl_S**, **Dn_Item_S** and **Dn_Stm_S**) will be allocated in short blocks of 4-8 cells per block. Each block will have a parent pointer, not each cell.

3.3. Distributed Dependency Database

To reduce the working set for most compilations, the Delta dependency database will be distributed throughout the worlds of the universe. A central dependency database is retained, but it records only unit-to-unit dependencies. The detailed, declaration-level dependencies are stored in the referencing units themselves. In addition, a *top declaration database* is created for each library unit specification. The top declaration database is an attribute space associated with the library unit. It records certain dependencies that relate declarations within the library unit and its secondary units.

The central dependency database has the same structure and synchronization characteristics as the Gamma dependency database. It contains entries for unit ids only.

Within each unit, a map is maintained from the **Diana.Tree** value for a referenced symbol to a short Boolean array. Each array index corresponds to one of the dependency database relations. The value at a given index is true if the corresponding relation holds between the referenced id and the referencing unit.

The top declaration database is a map from the **Diana.Tree** value for a top declaration (possibly a placeholder) to a pair of sets of object ids. One set contains the units that have the **Subordinate_To** relation to the top declaration, and the other set contains the units that have **Sees_Used_Namesake_Via_Use_Clause** relation to the top declaration.

The top declaration database will be open for update while any secondary unit of the library unit is being compiled. Thus semanticization and coding will be serialized a bit more in Delta than in Gamma because of contention for this database. There should be no user-noticeable impact, unless hoades of programmers regularly work in one unit with many subprograms.

The top declaration database also becomes the site for defining placeholders. With placeholders in the top declaration database, we will be able to recover space when they are no longer needed. Placeholders are not garbage-collected in the Gamma system.

4. Code Generation and Archive

4.1. Code Generation

4.1.1. Incremental Operations on Coded Units

A goal of Delta is to allow incremental insertions and deletions in coded units without causing obsolesence of dependent units. This means that, unlike Gamma, offsets for declarations are allocated independent of the textual position of the declaration.

Every program unit is composed of a set of declarative regions. A package may have three such regions, namely, the visible part, the private part, and the body. A subprogram has one region.

Offsets for declarations are assigned relative to some runtime frame. A runtime frame is either a module (package or task) or a subprogram frame. Each runtime frame contains declarations from some set of declarative regions. Normally, a frame contains the declarations in the declarative regions of a single program unit. Optimizations, such as package integration, may cause additional declarative regions to be mapped into the same runtime frame.

In order to perform an incremental insertion into a declarative region it is necessary to know how offsets are allocated in the runtime frame that contains that declarative region. This can be done by associating with every declarative region an *offset usage vector* that records which offsets of the enclosing runtime frame are used by that particular declarative region. Then the overall runtime frame can be characterized by the union of the usage vectors for all the declarative regions in the frame. When a new offset needs to be allocated it can be an offset that is not currently used in the frame, and the usage vector for the containing declarative region is updated.

Incremental assignment of offsets has the following obsolesence characteristics:

- The code segment associated with the declarative region is obsolesced.
- The permanent cg attributes of the declarative region are NOT obsolete.

For example, if an incremental insertion is made into a package visible part, the code segment associated with package is made obsolete but clients of the visible part are not affected. Main programs, however, are affected because they have a direct dependency on the code segments.

4.1.2. Maintaining Compatibility Among Views

In order to maintain compatibility between different views of the same subsystem, consistency must be maintained in the assignment of offsets to equivalent declarations in different views. This requires a mechanism to identify equivalent declarations, assign them offsets, and to maintain these offsets independent of any of the individual units in the subsystem.

4.1.2.1. Compatibility Database

In order to identify declarations as equivalent a *Compatibility Database* is created for each library unit visible part in a subsystem. The data base contains a declaration map, a child map, and a set of target maps. The declaration map is a map from a declaration signature to a unique declaration number. The declaration signature is composed of the fully qualified image of the declaration[10] and the declaration number of the parent package. This signature uniquely identifies the declaration across all views containing the library unit.

The declaration number for each declaration is stored on the **Diana.tree** of the declaration as a permanent attribute. Declaration numbers are computed immediately after the declaration has been installed.

The data base also contains a child map that maps each declaration to declarations nested within it. For example, the child map is used to get from each nested package to declarations that are within that package in any view of the subsystem.

Target dependent information in the data base is represented by a set of target maps. Each target map is a map from declaration number to target specific information about the declaration.

Because the compatibility database is necessary to maintain runtime compatibility with archived code, the database must be archivable by source archive.

The compatibility database is also required in order to support other targets. For other targets the database will typically maintain the compatibility of assembler labels between spec views and load views.

4.1.2.2. Offset Allocation

When a new declaration is inserted into a visible part the compilation coupler is called to assign the target dependent information that goes into the target map. For the R1000 this target information is the runtime offset of declaration. The coupler call invokes the

[10]The declaration image is fully qualified in the sense that all used names in the declaration are fully qualifed Ada names.

R1000 Code Generator. The code generator is passed the **Diana.tree** for the declaration that was inserted. The code generator is able to determine the declarative region that this declaration is part of and which runtime frame that region will be in. The target map is then queried to build up a *composite usage vector* that is essentially the union of the usage vectors for all the units in different views of the subsystem. A free offset can then be assigned to the new declaration.

For some views the offset that was assigned may conflict with offsets used by the private part or the body. This conflict would be determined by looking at the usage vectors of those regions. If an offset conflict occurs then the declarations for the private part and the body must be allocated new offsets. This new offset allocation causes the demotion of the body to the installed state. In order to make offset conflicts infrequent a buffer zone can be created between the offsets in the visible part and those in the private part or body.

4.1.3. Checking Compatibility of Spec and Load Views

Compatibility between spec and load views can be determined through a comparison of the offset usage vectors for the visible parts of the different views. The compatibility data base and the offset allocation scheme guarantee that offsets can be used to uniquely identify declarations for the R1000 target. Thus, the usage vectors can simply be compared to determine compatibility.

4.1.4. Code Database

The *Code Database* is used to accelerate loading. The code database exists in every world and captures all the information needed to load a unit that resides in the world. In addition the code database contains all of the information needed to archive the code in the world.

For a normal world the code database will be distributed among the attribute spaces of the library visible parts. Each such attribute space will contain all the coding information for that unit and all of its children. For a code archived world the database will exist as a single object that contains all of the coding information every unit in the world.

The code database is updated after every promote or demote operation that involves the coded state. This produces a small amount of serialization at the end of every such promote or demote. Because the code database caches the information needed for loading, there is no need to open or traverse Diana trees as in the Gamma system.

The actual contents of the code database will be the following for each Ada unit in the world:

- The simple name of the unit.

- The current state of the unit.
- The code segment name (if the unit is coded).
- The context dependencies that are used to determine elaboration order.
- The imports that are required (if the unit is coded).
- The subunits of the unit. This is needed in order to determine completeness.[11]

The code database contains all of the information needed to archive a world, and is in fact the object that is archived when code archive is run.

4.1.5. Relocation of Attribute Spaces

In order to support the relocation of Ada units, operations must be provided to relocate the attribute spaces produced by the code generator. Iterators will be provided over each attribute map to allow all Diana pointers, which require relocation, to be fixed.

4.2. Code Archive

This section was previously published as part of **RM**: **[SYSTEM.SPECCODEARCHIVE.LPT]**.

4.2.1. Features

Here are the code archive features available in Delta. They are chosen to be simple to implement, while still providing useful functions.

Only subsystem Load Views can be copied using **Code_Archive**. Other kinds of objects, including subsystem Spec Views, can be copied using **Source_Archive**, but not **Code_Archive** (for convenience, **Code_Archive** may provide a way to put **Source_Archive**'d objects onto the same tape with **Code_Archive**'d objects). Any Ada library unit in a copied view can be executed in the same way as the original. Main programs in a copied view are still main programs, although they may be bound to versions of units imported from outside the subsystem.

When saving a Load View, the user can specify which library units in that view will be 'exported', that is, will be visible once the view has been restored. If a unit is exported,

---

[11] This information is not available directly in the directory system. The directory system contains information on subunits when the subunit object has been created, but not when only the stub exists.

then its **Code_Archive** copy can be compatibility-checked[12] and compiled against in the same way as the original unit. If a unit is not exported, then its **Code_Archive** copy cannot be compatibility-checked or compiled against (it is still possible to compile against a compatible Spec View, but not against the Load View). By default, no units are exported. The user can also choose to export the units named in the view's **EXPORT** file, or to export all library units in the view.

Copied objects are different from the original objects in the following ways:

- Copied objects cannot be modified (they can be deleted).
- Copied Ada units have no bodies or separately compiled subunits. Although the code generated from these compilation units is copied and can be executed, the source is not copied. There are no objects which contain the source, and it cannot be examined.
- Copied Ada units cannot be debugged. Definitions in the visible part of an exported spec are visible and can be read by the debugger, but definitions in bodies or non-exported units are not.

Units which were coded with a pre-**Code_Archive** code generator cannot be copied. Such units must be recoded before they can be copied. This recoding is upward compatible, that is, the recoded units can coexist and run with old units.

4.2.2. Implementation Approach

For each Load View, all objects except for Ada units are copied using **Source_Archive**. The visible part of each exported Ada unit is copied as source text. Loader information is copied using algorithms to be provided by SWB. Code segments are copied verbatim (except for their debug tables, which are removed).

The implementation depends on the following changes in the native code format and the loader:

- Code segment addresses and exception names will be passed at runtime via module import spaces, rather than being wired into the code (as is now done).
- Loader information for a subsystem will be kept in a single loader data base. This object will be a child of the subsystem world. For each Ada library unit in the subsystem, the loader database will contain the following information:
    - The code segment, and the code segments for its subunits. Code segment ids will be recorded in the database, and code segment objects will be children of the database object.

[12] Using the batch compatibility checker or the Delta CG compatibility mechanism

- All dependencies on other units. This includes **with** dependencies, dependencies introduced by subunits, and elaboration order dependencies.

- If the unit is a main program, the id of its elaboration code segment will be recorded in the database, and the elaboration code segment will be a child of the database.

This information is sufficient to load or execute any unit in the view, without touching the Ada object for the unit. This will have performance advantages in the non-**Code_Archive** case, and it makes it possible for **Code_Archive** to prune Ada objects out of the view without affecting execution semantics.

4.2.3. Interchange Form

Here is a breakdown of the components of an interchange set, that is, the items of information that are produced by **Code_Archive.Save** and consumed by **Code_Archive.Restore**.

The interchange form is designed to be generated (by **Save**) and consumed (by **Restore**) sequentially, in the order given below. The medium used to transfer an interchange set must provide sequential, transparent byte transfer, and a way to mark the end of the interchange set (*e.g.*, end-of-file). Two interchange sets may be catenated, to form a larger but still correct interchange set.

Object ids may appear anywhere in the interchange set, for example in the loader information. The interchange form of an object id includes the object class, an object instance number, and whether the object is part of the same interchange set. Object instance numbers are arbitrarily assigned by **Code_Archive.Save**, and are unique only within an interchange set. The first occurrence in the interchange set of each object id includes the pathname of the object.

4.2.3.1. Libraries and Files

All libraries and files (excluding loader information) are interchanged in **Source_Archive** form.

4.2.3.2. Ada Units

The interchange form of an Ada unit is its object id and, if the object is exported, the source for its visible part. The source for each unit is interchanged as ASCII text (possibly in **Source_Archive** form). Since each unit is appearing for the first time in the interchange set, its object id includes its path name.

4.2.3.3. Loader Information

Loader information is interchanged in a canonical form to be defined by SWB. Within the loader information, object ids (of Ada units and code segments) are represented in the interchange form described above.

4.2.3.4. Code Segments

The interchange form is an object id, followed by a code segment, consisting mainly of R1000 macro-instructions. The interchange form of the code segment has a truncated debug table.

4.2.4. Conversion Algorithms

The **Save** algorithm converts from the native representation of Ada units to their interchange form, and the **Restore** algorithm does the reverse.

4.2.4.1. Save

Save requires the following steps:

1. Libraries and files are **Source_Archive.Save**'d.
2. The object ids of all library units are saved.
3. The source of exported library units is saved.
4. Loader information is saved.
5. All code segments which were referenced by the loader information (that is, whose object ids appeared in its interchange form) are saved. The debug tables are removed before saving them.

4.2.4.2. Restore

Each subsystem view is **Restore**'d either entirely or not at all. If one tries to restore a view, and a view of that name already exists on the target machine, then nothing in the view will be restored. **Restore** will not merge selected units from the restored view into the target view.

5. Subsystems, Configurations, and Version Control

CMVC-WART is in two pieces. The first is the low level reservation and history database. The second is the commands that use this database. It is anticipated that the marketing organization will take a leading role in developing the command interface, both in specification and implementation. This interface will be tested using the Gamma system, and then refined (or reimplemented) for Delta. The database will be built by Development.

This chapter is in two parts. The first section is the Ada spec for the CMVC-WART database manager. Sections 5.2 through 5.3 discuss some issues concerning the command interface and presents, through example, a possible interface.

5.1. The CMVC-WART Spec

```
with Action;
with Calendar;
with Directory_Implementation;

package Cmvc_Implementation is
    package Directory renames Directory_Implementation;

    -- CMVC is based on the notion of elements. An element is a logical
    -- entity that encapsulates changes within objects over time. Any
    -- object class that has a text representation and conversion functions
    -- can be managed by CMVC. Elements have an internal name, supplied
    -- when the element is created. This is the name used for all CMVC
    -- operations. There is an external name associated with each version
    -- set, which is used for all copy in and out operations.

    -- Each version of an element is a snapshot of that element. It
    -- represents a physical realization of that element at some point in
    -- its history. A Gamma version has this property. Each successive
    -- version is a new generation of the element.

    -- A view is defined to mean a snapshot of a collection of elements; it
    -- calls out specific versions of specific elements. This isn't
    -- necessarily a physical representation. In other words, this isn't a
    -- world but a more abstract concept.

    -- A version set is a time ordered set of versions for one element.
    -- There is a straight line of descent; each version was created from
    -- its parent by changing the parent. There is no skipping; every
    -- generation of the element is represented. Each version set for an
    -- element represents a different time line, and each can be reserved
    -- independently. Only one version in a version set can be checked
    -- out, and this is the newest version. The version set is optionally
    -- named, which is useful for iterating over the sets. The version set
```

```
-- also maintains the external name for this set of versions of the
-- element.

-- A configuration is a realization of a view. A configuration can have
-- a directory name, which means there is some representation on the
-- disk for it. There are two types of configurations, release
-- configurations and working configurations.  A working configuration
-- is one in which versions can be changed.  A release configuration
-- specifies a set of versions which are frozen; neither the release
-- configuration nor the versions specified can be changed.
-- Configurations are named by the user.

-- Many operations want a library to work in.  To simplify the
-- specification of this library, CMVC maintains a map of user/session
-- pairs to a default library. Commands would use this map to simplify
-- the user interface. Operations exist to get and set the map.

-- The database maintains a map between user/session pairs and a
-- configuration. This configuration can be used by commands as a
-- default configuration. There are operations to get and set this
-- configuration.

-- In many cases software must run on various targets.  It is often the
-- case that most of the elements are the same, but a few may be
-- different to account for the differences in the targets.  It is
-- desirable to capture this information in order to allow simultaneous
-- independent changes to the elements that are different, while
-- controlling access to the elements that are the same.  Another
-- common scenario is the need to have two people work on one element
-- in parallel for some time, then merge the changes together. The
-- process of splitting elements and operating on them is called
-- maintaining varying lines of descent. These variants branch out from
-- some line, and may rejoin later. Branching out is done using create;
-- bringing two alternate lines together is done using merge. In CMVC,
-- alternate lines of descent is accomplished by using version sets and
-- configurations.  Each version set is a line of descent for some
-- element.  Each configuration selects at most one version set for an
-- element.  Configurations that refer to the same version set are
-- linked; the element has one reservation across all such
-- configurations, and can only be changed serially.  If the
-- configurations refer to different version sets for an element, the
-- element can be reserved independently.

-- There is a database for each set of related configurations.  The
-- database must be provided to each operation discussed below.  There
-- is a small machine wide map that maps user/session pairs to a
-- default database.  The commands can use this to make talking to the
-- database easier.

No_Such_Version     : exception;
No_Such_Version_Set : exception;
No_Such_Element     : exception;
Bogus_Parameters    : exception;
```

```
Unknown_Error       : exception;

-- The above are the only exceptions propagated out of this package.
-- Unknown_error is raised for internal errors and unexpected errors
-- propagated out of other packages.  Bogus_Parameters is raised when
-- the parameters make no sense, such as when configurations from one
-- database are mixed with elements from another.

subtype Library_Name is String;
subtype History_File is String;

type Error_Status is private;
subtype Error_Msg is String;
function Nil return Error_Status;
function Is_Nil (Status : Error_Status) return Boolean;
function Is_Bad (Status : Error_Status) return Boolean;

function Get_Error_Msg (Status : Error_Status) return Error_Msg;

                ---------------------------------

-- There is a database for each set of related configurations.

type Database is private;
function Nil return Database;
function Is_Nil (Db : Database) return Boolean;
function Name_Of (Db : Database) return String;

procedure Open_Database (Db_Name : String;
                         Db : out Database;
                         Status : out Error_Status);

procedure Create_Database (Db_Name : String;
                           Db : out Database;
                           Status : out Error_Status;
                           Dont_Keep_Source : Boolean := False);

-- If dont_keep_source is true, no source differentials are kept in the
-- database.  This means versions cannot be retrieved from the
-- database, and that the merge command depends on external objects
-- versus internal information.  It is set to true to save disk space
-- and to speed up check in.

procedure Expunge_Database (Db : Database; Status : Error_Status);

-- Removes all elements and version sets not referenced by a
-- configuration.

procedure Set_Default_Database (Db : Database;
                                User : String := "";
                                Session : String := "");

function Default_Database (User : String := "";
```

```
                        Session : String := "") return Database;

                ---------------------------------

type Configuration is private;
function Nil return Configuration;
function Is_Nil (Config : Configuration) return Boolean;
function Name_Of (Config : Configuration) return String;
function Database_Of (Config : Configuration) return Database;

subtype Configuration_Object is Directory.Object;
function Is_Configuration_Object (Obj : Configuration_Object)
             return Boolean;


procedure Open_Configuration (Config_Name : String;
                              Config : out Configuration;
                              Error : out Error_Status;
                              Db : Database := Default_Database);

procedure Open_Configuration (Obj : Configuration_Object;
                              Config : out Configuration;
                              Error : out Error_Status);

-- Return a handle to a configuration.  The configuration must exist.
-- This handle is used for most interesting element operations.

procedure Create_Configuration (Config_Name : String;
                                Status : out Error_Status;
                                Golden_Config : Configuration := Nil;
                                Default_Library : String := "";
                                Version_Set_Name : String := "";
                                Initial : Configuration := Nil;
                                Make_Copies : Boolean := False;
                                Is_Release : Boolean := False;
                                Db : Database := Default_Database);

-- Create a new configuration.  It can optionally be initialized.  If
-- make_copies is true, new version_sets are made in each of the
-- elements selected by the initial configuration, which are then
-- initialized with the version selected by the initial configuration.
-- These version sets are given the name provided.  If no name is
-- provided, the version sets are given the configuration name.  If the
-- name to be given clashes with an existing name, "_N" for some value
-- of N is appended to make it a unique name. If make_copies is false,
-- the new configuration references all the same version sets and
-- versions as the initial configuration, and the version_set_name
-- parameter is ignored. In other words, it is linked. If is_release is
-- true, the new configuration is frozen (is a release).

-- Golden_Config is the name of a configuration that is to be copied
-- into automatically whenever an element is checked in.  The intent is
```

```
-- to allow the user to keep a current copy of everything.  This
-- package doesn't actually do the copying, but makes available the
-- information to the command packages.

-- Default_Library is the name of the library to be used by default
-- for this configuration.  It is used in all commands that require a
-- library, but are given the null string.

function Golden_Config (Config : Configuration) return Configuration;

function Default_Library (Config : Configuration) return String;

procedure Create_Config_Object (Name : String;
                                Config : Configuration;
                                Status : out Error_Status);

-- Create a configuration object with name 'name'.  This object can be
-- used to get a configuration handle.  If a configuration object
-- editor is ever provided, it would accept one of these.

procedure Delete_Configuration (Config : Configuration;
                                Status : out Error_Status;
                                Delete_Release : Boolean := False);

-- Delete a configuration.  The elements and version sets are not
-- deleted.  Delete_release must be true to delete a release
-- configuration.

function Is_Release (Config : Configuration) return Boolean;

-- Returns true if the argument is a release configuration.

procedure Set_Default_Configuration (Config : Configuration;
                                     User : String := "";
                                     Session : String := "");

function Default_Configuration (User : String := "";
                                Session : String := "")
             return Configuration;

               ---------------------------------

procedure Set_Default_Library (Library : Library_Name;
                               User : String := "";
                               Session : String := "");
function Default_Library (User : String := "";
                          Session : String := "") return Library_Name;

               ---------------------------------

type Element is private;
function Nil return Element;
function Is_Nil (Elem : Element) return Boolean;
```

```
function Name_Of (Elem : Element) return String;

type Element_Class is private;
function Nil return Element_Class;
function Is_Nil (Class : Element_Class) return Boolean;
function Image (Class : Element_Class) return String;
function Value (Class_Name : String) return Element_Class;
function Class_Of (Elem : Element) return Element_Class;

Ada_Class_Name  : constant String := "Ada_Class";
Text_Class_Name : constant String := "Text_Class";

procedure Define_Class (User_Class_Name : String;
                        Class : out Element_Class;
                        Already_Exists : out Boolean;
                        Status : out Error_Status;
                        Db : Database := Default_Database);

-- Define a user class.  The class name is passed in. An element_class
-- is returned.  If an identical user class has already been defined,
-- already_exists is set to true.  These classes are only valid within
-- the database given.  Classes defined in more than one database
-- cannot be compared by comparing the element_class, but must be
-- compared by getting and then comparing the string_name.

type Version_Set is private;
function Nil return Version_Set;
function Is_Nil (Set : Version_Set) return Boolean;
function Name_Of (Set : Version_Set) return String;

subtype Generation is Natural;
Last_Version : constant Generation := Natural'Last;
Nil_Version  : constant Generation := 0;

type Version is private;
function Nil return Version;
function Is_Nil (Vers : Version) return Boolean;

procedure Open_Element (Element_Name : String;
                        Elem : out Element;
                        Status : out Error_Status;
                        Db : Database := Database_Of (Default_Configuration));

-- Gets a handle for an element.  This handle is used to look at the
-- various version sets for the element.

procedure Create_Element (Element_Name : String;
                          Config : Configuration;
                          Class : Element_Class;
                          External_Name : String;
                          Elem : out Element;
                          Status : out Error_Status;
                          Initial_Value : Version := Nil;
```

```
                        Version_Set_Name : String := "";
                        Dont_Keep_Source : Boolean := False);

procedure Create_Element (Element_Name : String;
                        Db : Database;
                        Class : Element_Class;
                        External_Name : String;
                        Version_Set_Name : String;
                        Elem : out Element;
                        Status : out Error_Status;
                        Initial_Value : Version := Nil;
                        Dont_Keep_Source : Boolean := False);

procedure Create_Element (Element_Name : String;
                        Config : Configuration;
                        Class : Element_Class;
                        External_Name : String;
                        Elem : out Element;
                        Status : out Error_Status;
                        Initial_Value_Object : String := "";
                        Version_Set_Name : String := "";
                        Dont_Keep_Source : Boolean := False);

procedure Create_Element (Element_Name : String;
                        Db : Database;
                        Class : Element_Class;
                        External_Name : String;
                        Version_Set_Name : String;
                        Elem : out Element;
                        Status : out Error_Status;
                        Initial_Value_Object : String := "";
                        Dont_Keep_Source : Boolean := False);

-- Create a new element.  The new element can be inserted into a
-- configuration (using the first procedure).  An empty version set is
-- created for the element.  In the case that a configuration is
-- supplied, the version set name defaults to the configuration name.
-- In the database case, the version set name must be supplied.  The
-- version set is optionally initialized by copying the contents of a
-- version into it (as generation 1). The configuration must be a
-- working configuration. Path_from_library specifies a string that is
-- to be prepended to the element name and appended to the library name
-- when the element is copied out of the database. Initial_value_object
-- is the name of some directory object that is to be used as an
-- initial value.

procedure Delete_Element (Elem : Element;
                        Status : out Error_Status;
                        Config : Configuration := Default_Configuration);

-- Delete the element from the configuration.  The element is not
-- deleted from the database.  The configuration must be a working
-- configuration.
```

```
procedure Delete_Element (Elem : Element;
                          Db : Database;
                          Status : out Error_Status);

-- Delete the element from all working configurations in the database.
-- This operation ignores release configurations.

procedure Add_Element (Elem : Element;
                       Status : out Error_Status;
                       Config : Configuration := Default_Configuration);

-- This operation adds an element (and a version set) to a working
-- configuration.  There must be only one version set associated with
-- the element to use this operation.

function Is_In_Configuration (Elem : Element;
                              Config : Configuration := Default_Configuration)
             return Boolean;


               --------------------------------

procedure Open_Version_Set (Set_Name : String;
                            Elem : Element;
                            Set : out Element;
                            Status : out Error_Status);

-- Get a handle on a version set.  This handle is used to traverse
-- across the versions contained in the set.

procedure Open_Version_Set (Elem : Element;
                            Set : out Version_Set;
                            Status : out Error_Status;
                            Config : Configuration := Default_Configuration);

-- Get a handle on a version set determined by an element/configuration
-- pair.

procedure Create_Version_Set (Set_Name : String;
                              Elem : Element;
                              External_Name : String;
                              Set : out Version_Set;
                              Status : out Error_Status;
                              Initial_Value : Version := Nil);

procedure Create_Version_Set (Set_Name : String;
                              Elem : Element;
                              External_Name : String;
                              Set : out Version_Set;
                              Status : out Error_Status;
                              Initial_Value_Object : String := "");

-- Create a new version set in some element.  The new version set can
```

```
-- be initialized.

procedure Add_Version_Set (Set : Version_Set;
                           Config : Configuration;
                           Status : out Error_Status;
                           Replace_Ok : Boolean := True);

-- Add (or replace) a version set to a configuration.  This operation
-- implies the adding of an element as well, as a version set is
-- contained within an element.  The configuration must be a working
-- one.  The configuration is set to refer to the newest version in the
-- set.

procedure Prune_Version_Set (Set : Version_Set;
                             Up_To_Generation : Generation;
                             Status : out Error_Status);

-- Throw away the first up to the up_to_generation versions from the
-- version set.  This operation fails if any configuration references a
-- version to be discarded.  An iterator exists to help find these
-- blocking configuration(s).

procedure Change_External_Name (Set : Version_Set;
                                External_Name : String;
                                Status : out Error_Status);

-- Modify the name used for the version set.  This name is appended to the
-- library name to build a complete external name.
-- Note that this affects all configurations using the version set.

procedure Change_Name (Set : Version_Set;
                       New_Name : String;
                       Status : out Error_Status);

-- Change the name of the version set.

function External_Name (Set : Version_Set) return String;

-- Return the path specified when the version set was made.

function Element_Of (Set : Version_Set) return Element;

                ---------------------------------

function Get_Version (Set : Version_Set;
                      Gen : Generation := Last_Version) return Version;

function Get_Version (Elem : Element;
                      Config : Configuration := Default_Configuration)
              return Version;

-- Get a handle on a particular version.  The second form returns the
-- version selected by a configuration.  This handle is used to get
```

```
-- the version's history and contents.

function First_Generation (Set : Version_Set) return Generation;

-- Return the generation of the first version for the set.  This is
-- something other than one after a prune_version_set

function Last_Generation (Set : Version_Set) return Generation;

function Generation_Of (Vers : Version) return Generation;

function Version_Set_Of (Vers : Version) return Version_Set;

function Element_Of (Vers : Version) return Element;

                --------------------------------

procedure Create_From_Db (Vers : Version;
                          Where : String;
                          Status : out Error_Status);

procedure Check_Out (Set : Version_Set;
                     Vers : out Version;
                     Status : out Error_Status;
                     Config : Configuration := Default_Configuration;
                     Action_Id : Action.Id := Action.Null_Id;
                     User : String := "");

procedure Check_In (Set : Version_Set;
                    Current_Source : Directory.Object;
                    Vers : out Version;
                    Status : out Error_Status;
                    Config : Configuration := Default_Configuration;
                    Action_Id : Action.Id := Action.Null_Id;
                    User : String := "");

-- Check out (or in) some element.  Since the element is specified
-- by the version set, the element need not be provided.  Check out
-- creates a new version.  The configuration is changed to reflect
-- the use of the new version.  Check in verifies the same
-- configuration is being used.  The command package can check to see
-- if the user doing the check in is the same one that did the
-- check out.

-- The action parameter is provided so the command package can do
-- preprocessing, and back out if the check in or out fails.  The action
-- does not allow backing out of the database operation itself.  There
-- is no mechanism provided to back out of a database operation.

-- Check out returns the version for the new copy.  The version can be
-- used to locate a copy of the element on the disk somewhere by using
-- the last_known_object history item. The application must copy this
-- object to the destination.  If the last_known_object is Nil, the
```

```
-- application should request a copy to built out of the database,
-- supplying the destination location.

-- Check in wants the directory.object of the item being checked in. It
-- uses this to compute the differentials, and also saves it in the
-- database (for passing to the next check out). It returns the
-- version for any later processing that might be needed (like
-- compiling).

-- The command package check in might also want to check for the
-- existence of a golden configuration, and copy the object there. The
-- resulting object would be given to check in. If a golden
-- configuration is desired, the returned version should be accepted
-- into that golden configuration to bring that configuration up to
-- date. Remember to do all of the work required before check in under
-- one action, so the operations can be backed out if the check in
-- fails. The most common failure is 'wrong configuration', so the
-- command package might want to check that itself first.

-- The user string is used to mark who did the operation. If "" is
-- supplied, the login name is used.

procedure Accept_Changes (Set : Version_Set;
                          Vers : out Version;
                          Status : out Error_Status;
                          Gen : Generation := Last_Version;
                          Config : Configuration := Default_Configuration);

-- The requested version is located and its version is returned. The
-- configuration is changed to refer to the version. This operation
-- can be reversed by accepting the previous version.

procedure Merge_Changes (Elem : Element;
                         From_Config : Configuration;
                         Conflicts_Detected : out Boolean;
                         Vers : out Version;
                         Status : out Error_Status;
                         To_Config : Configuration := Default_Configuration;
                         List_File : String := "";
                         Make_Parsable : Boolean := False;
                         Effort_Only : Boolean := False;
                         Join_Configs : Boolean := True;
                         User : String := "");

-- The versions in the two version sets selected by elem are merged
-- together, with the result being left in the to_config. This
-- operation always creates a new version. The to_config is marked
-- to refer to the new version. An error occurs if the two
-- configurations don't refer to the last versions in the version
-- sets.

-- This command requires that the two version sets be related,
-- which means that one of the sets must have been created from the
```

```
-- other. If the split point cannot be located, or never existed,
-- the merge fails.

-- If conflicting changes are found, the out parameter
-- conflicts_detected is set to true.  This by itself is not an error

-- List_file specifies a text file where the merged result can be
-- placed.  The merge points in the file are marked in the same fashion
-- as file_utilities.merge, unless make_parsable is true, in which case
-- no marks are put in at all.   If conflicts are detected, these are
-- marked regardless of the setting of make_parsable.

-- Effort_only will do the merge without actually updating the database.

-- Join_configs, if true, will change from_config to refer to the
-- same version set as to_config. In other words, the two
-- configurations are relinked.

                ---------------------------------

type Basic_History is
    record
        Ever_Checked_Out      : Boolean;
        When_Checked_Out      : Calendar.Time;
        Checked_Out_To_Config : Configuration;
        Ever_Checked_In       : Boolean;
        When_Checked_In       : Calendar.Time;
        Edit_Time_Stamp       : Calendar.Time;
        Last_Known_Object     : Directory.Object;
        Split_From_Version    : Version;
        Merged_From_Version   : Version;
    end record;

-- Split refers to the source version when the set was created.  merged
-- refers to a version that was merged into this one.
-- Checked_out_to_config is nil if the config has been deleted.

procedure Get_Basic_History (Set : Version_Set;
                             History : out Basic_History;
                             Status : out Error_Status;
                             Gen : Generation := Last_Version);

-- Return the history for some generation in the set.

function Who_Checked_Out (Set : Version_Set;
                          Gen : Generation := Last_Version) return String;
function Who_Checked_In (Set : Version_Set;
                         Gen : Generation := Last_Version) return String;

-- Return the string history items

function Is_Checked_Out (Set : Version_Set) return Boolean;
```

```
-- simple way to see if the version is currently checked out

procedure Set_History (From_File : History_File;
                       Set : Version_Set);

-- Copy the text file named 'from_file' to the history database, and
-- associate it with the last version in the version set.  The version
-- must be checked out and not checked in.

procedure Append_History (From_File : History_File;
                          Set : Version_Set);

-- Same as above, only the file is appended instead of replacing.

procedure Get_History (To_File : History_File;
                       Set : Version_Set);

procedure Get_History (To_File : History_File;
                       Vers : Version);

-- Copy the history file from the database into a text file named
-- 'to_file'.  If the 'set' form is used, the file of the last version
-- in the set is copied.  Otherwise the file for the selected version
-- is copied.

                ---------------------------------

type Configuration_Iterator is private;
procedure Initialize (Db : Database; Iter : out Configuration_Iterator);
procedure Initialize (Elem : Element; Iter : out Configuration_Iterator);
procedure Initialize (Set : Version_Set; Iter : out Configuration_Iterator);
procedure Initialize (Set : Version_Set; Up_To_Version : Generation;
                      Iter : out Configuration_Iterator);
procedure Next (Iter : in out Configuration_Iterator);
function Done (Iter : Configuration_Iterator) return Boolean;
function Value (Iter : Configuration_Iterator) return Configuration;

-- Iterate over configurations.  The iterator can be built to iterate
-- over all configurations, all configurations that reference some
-- element, all configurations that reference some version set, or
-- all configurations that reference the first up to n'th version of
-- a version set.

type Element_Iterator is private;
procedure Initialize (Config : Configuration; Iter : out Element_Iterator);
procedure Initialize (Db : Database; Iter : out Element_Iterator);
procedure Next (Iter : in out Element_Iterator);
function Done (Iter : Element_Iterator) return Boolean;
function Value (Iter : Element_Iterator) return Element;

-- Iterate over elements.  The options are to iterate over all elements
-- in the database, or to iterate over all elements in a configuration.
```

```
    type Version_Set_Iterator is private;
    procedure Initialize (Db : Database; Iter : out Version_Set_Iterator);
    procedure Initialize (Config : Configuration;
                          Iter : out Version_Set_Iterator);
    procedure Initialize (Elem : Element;
                          Iter : out Version_Set_Iterator);
    procedure Next (Iter : in out Version_Set_Iterator);
    function Done (Iter : Version_Set_Iterator) return Boolean;
    function Value (Iter : Version_Set_Iterator) return Version_Set;

    -- Iterate over the version sets specified by a configuration, over the
    -- version sets associated with an element, or over all version sets in
    -- the database.  Iterating over version sets is useful for finding an
    -- external name and matching it against the name of some object, in
    -- order to find an element name.  This would be done by stripping off
    -- the library and then comparing what is left to the external names
    -- for the version sets.

    type Version_Iterator is private;
    procedure Initialize (Set : Version_Set; Iter : out Version_Iterator);
    procedure Initialize (Config : Configuration; Iter : out Version_Iterator);
    procedure Next (Iter : in out Version_Iterator);
    function Done (Iter : Version_Iterator) return Boolean;
    function Value (Iter : Version_Iterator) return Version;

    -- Iterate over the versions in a version set or the versions selected
    -- by a configuration.

end Cmvc_Implementation;
```

5.2. Check Out And In

A major question facing the design of a command package is "Are elements checked out to a person or a place?" This section discusses this issue.

5.2.1. Checking Out To A Place

In this scenario, elements are checked out to a library or world. Anybody can check out the element, work with it in the library, and check it back in. The element can be copied to some other library and changed, but cannot be checked in from that other library.[13]

The principle advantage of this method is that elements can be easily found. They are changeable only in one location, which means it is easy to find the latest version. A second advantage is that the process is independent of a user's session attributes. Since rights come from location, moving to a new library automatically tells the system everything it needs to know to check in and out elements.

[13] Under some enforcement schemes, the copy becomes unchangeable.

The principle disadvantage is the lack of personal control over where the element is placed, manipulated, and tested. Since copies are discouraged (or even prevented), the user cannot make a sublibrary to play in. The element must be manipulated in it's final resting spot.

5.2.2. Checking Out To A Person

In this scenario, elements are checked out to a person. The user decides where the element is put. The user is free to copy it where ever she wants to. If enforcement is used, only that person would be allowed to edit the element, no matter where it is located.

The principle advantage of this scheme is control and accountability. Only one person can make changes, so that person can know everything that happened to the element. That person can be help accountable for testing and documentation.

A disadvantage is the necessity for the user to provide information about what the user is doing. For example, the user must tell the system what database and configuration is being used. Moving to a new directory doesn't automatically cause this information to change.

5.2.3. Comments

It isn't clear which of the two schemes is better. Each has its good and bad points. Enforcement is equally difficult for each scheme. It is possible that we will have to provide both.

Enforcement for the place method would be done by marking the object in the directory corresponding to the element as checked out or not. If the object is checked out, the object can be modified. If it isn't checked out, it can't. Copy of the object always clears the checked out indication. The object cannot be moved.

Enforcement of the person method is similar. The object would be marked as checked out by a person when it is created (or checked out into). When an attempt is made to modify the object, a check is made that the user who checked it out is the one making the attempt. If not, the operation fails. Copying the object clears the user information; moving the object does not.

The next section assumes a check out to user paradigm.

5.3. Commands

This section proposes a command level interface. It is recognized that there are many possibilities and variants of this one. This section should be viewed as an example of how to use the database.

The examples below make use of a hypothetical software organization. The project makes use of two separate worlds (or subsystems), **Command** and **Low_Level**; **Command** makes use of **Low_Level**. The group in question is working on **Low_Level**. The group manager is Shirley. Two programmers, Fred and Gail, work for Shirley.

Shirley wants to set up an environment where Fred and Gail can work independently, without stepping on each others toes, but ensuring serialization of changes (reservations). She also wants a world that collects all of the changed elements for release purposes.

Low_Level contains four packages **A, B, C, D**.

5.3.1. Starting Up

Shirley first creates a database for **Low_Level**. This is done using the command

```
Cmvc_Cmd.Create_Database(Name => "Low_Level_Db");
```

Since she wants to use this database for the rest of the session, she executes

```
Cmvc_Cmd.Set_Default_Database("Low_Level_Db");
```

She then creates a configuration for the world for collecting the releases using

```
Cmvc_Cmd.Create_configuration("Golden");
```

Also, she creates a world to hold the elements, named **"!Low_Level.Golden"**. She sets her default library to that world. The elements for the four packages are then created using

```
Cmvc_Cmd.Create_Element("A");
```

and so on. Lastly, she creates the worlds for Fred and Gail, named **Fred** and **Gail**, in **"!Low_Level"**, and makes private copies of the Golden configuration for them using the command

```
Cmvc_Cmd.Copy_Configuration("Golden", "Fred", Linked=>True);
```

and so on.

Shirley can, as a convenience to Gail and Fred, change their login procedures to select the appropriate database, configuration, and library. For the remainder of this discussion, assume she has done this.

5.3.2. Continuing Development

Whenever Fred and Gail log in, their login automatically sets up an appropriate default configuration and database. Thus, when Gail wants to change (or create) package A, she issues the command

```
Cmvc_Cmd.Check_Out(Element => "A", To_Library => Default));
```

The package is copied to the specified library. Gail works on the package for a while.

While Gail is working on A, Fred fixes a major bug in B. He does so by checking the package using the **Cmvc_Cmd.Check_Out** command, modifying it, then checking it back in using

```
Cmvc_Cmd.Check_In(Element => "B");
```

Gail wants this change, as the bug is causing problems for her. She gets the change without checking out B by using the command

```
Cmvc_Cmd.Accept_Changes(Element => "B");
```

Package B is copied to her default library, since she didn't specify one to the command.

Every night, Shirley rebuilds the save directory (Golden) in order to gather together all the changed elements. She does this by executing the commands

```
Cmvc_Cmd.Accept_Changes(Element => "0", Deferred => True);
Cmvc_Cmd.Build;
```

The first command brings her configuration up to date. The second causes the changed elements to be copied to her default library, then compiled.

5.4. Issues

5.5. Improved View Mechanisms

5.5.1. Relocating Ada Units

The process of copying an installed or coded Ada unit to a new location without changing its compilation state is called *relocation*. Minor changes to the Gamma Diana implementation will make relocation more efficient, but these changes are not required to make relocation possible. Relocating an Ada unit should be 3 to 5 times faster than copying the unit as source and recompiling it. The goal is to relocate Ada objects without creating any garbage spaces in the process.

Relocation involves two major operations. In the first operation, a set of units, ***A***, is copied from one location to another, forming the set ***A'***, and the dependency database is changed to reflect the fact that the units of ***A'*** are new referencers of the same objects that were referenced by the units in ***A***. In the second operation, a set of units, ***B***, is modified to look as if they were compiled against the units of ***A'***, even though they were actually compiled against the units of ***A***, and the dependency database is modified to reflect these changes in referencing patterns.

In detail, relocation consists of the following operations:

COPY — Copy a set of coded units, ***A***, to a new site and update the dependency database.

C1. Each Ada unit a in A, its associated image object, Cg_Attribute space[14], code segment, and top declaration database are block-copied to the new site, forming a corresponding unit a' in the set of relocated units A'. Units a and a' have the same edit time stamps.

C2. When a library unit, u in A, is relocated, its associated top declaration database is also relocated. As the top declaration database is relocated, a Diana pointer to a unit a in A is modified to reference the corresponding unit a' in A'. Relations involving references to units not in the same world as u are not copied.

C3 If a range set in a map of the central dependency database contains a reference to unit a in A, it is augmented by adding a reference to the corresponding unit a' in A'.

MODIFY Simulate recompiling a set of units, B, once compiled against unit set A, against COPY'ed units A'.

M1. Each object in the universe of interest, B, that might reference a unit in A is traversed to find embedded Diana pointers.[15] Each pointer that points to an Ada unit in A is changed to point to the corresponding unit in A'. Pointers to units outside of the set A are left unchanged.

M2 When a pointer in unit b is changed to point to unit a' rather than unit a, b is deleted from each dependency database range set whose domain is a and is added to each corresponding dependency database range set whose domain is a'.

The set of modified units B must include A' if the system is to be left in a consistent state. Thus, the COPYing of A is always followed by the MODIFYing of A'. At a later time, the set B can be expanded to include other units as long as the relation between A and A' can be re-established. As long as a in A and a' in A' have the same edit time stamp, they are interchangeable with respect to the MODIFY operation.

Unless otherwise noted, the phrase "relocate A" means "COPY A and then MODIFY A'". A preliminary spec for the **Relocation** package is shown in Figure 5-1.

The Ada units of a Load View have no referencers outside the Load View (as far as Diana is concerned). Thus, a new Load View can be spawned from an existing Load View quite efficiently by relocating the Ada units within it and copying or rebuilding the

---

[14] Reorganized for Delta for more efficient access.

[15] Iterators must be added for all attribute maps that are used to implement code generator attributes. Other iterators already exist.

```
package Relocation is
    -- The relocation package changes the compilation context of Ada units
    -- trying to minimize the recompilation needed to change that context.
    -- The new context can be established by copying the units into the new
    -- context or by changing the imports for the existing context.

    -- As far as this package is concerned, a <<subsystem>> is a world that
    -- contains worlds that have identical structure. The nested worlds are
    -- called <<views>>.  Units can be relocated only between views of the
    -- same subsystem.

    -- Two units are <<equivalent>> if they each belong to a view of the
    -- same subsystem, have the same name with respect to the view, and have
    -- the same modification time stamps.  Immediately after a unit has been
    -- relocated, it is equivalent to the original unit.  It will remain
    -- equivalent until it or the original is demoted to source.

    -- A unit foo is a <<client>> of another unit bar if the unit foo withs
    -- bar or is a secondary unit of bar. A unit foo is a <<supplier>> of
    -- another unit bar if bar is a client of foo.

    -- A new context is <<consistent>> with the present context of a
    -- (compiled) unit if for every supplier of the unit in the present
    -- context there exists an equivalent unit in the new context.

    type Successfulness is
        (Ubiquitous_Other_Error, -- don't know what happened,
                                 -- but it ain't good
         Successless,            -- some units could not be copied,
                                 -- but for good reason.
         Left_Some_As_Source,    -- but all were copied
         Recompiled_Some,        -- but none failed to recompile
         Relocated_Some,         -- but none were demoted to source
         No_Action_Needed);      -- all units at destination are equivalent

    procedure Copy (Units : String;
                    Destination : String;
                    Relocatable_Clients : String;
                    Fixed_Clients : String;
                    New_Suppliers : String;
                    Allow_Demotion : Boolean := True;
                    Recompile : Boolean := True;
                    Action_Id : Action.Id := Action.Null_Id;
                    Status : out Successfulness);

    -- Intelligently copies the designated units and the specified clients
    -- to the indicated destination, optionally substituting new suppliers.
```

Figure 5-1: Preliminary Relocation Package - Part 1

other data structures of the view, including the link pack, activities, and object sets (as in the current implementation of **View.Spawn**).

```
    -- (description of Copy continued)

    -- Destination specifies a set of views (worlds) using wildcards or an
    -- activity.  Each unit that is copied is copied to the destination view
    -- that belongs to the same subsystem as the unit to be copied.

    -- The relocatable clients are the units that should be co-relocated if
    -- they reference the primary set of units.  The destination parameter
    -- must specify containers for these relocated clients. The fixed
    -- clients are any additional units that are to be modified to
    -- reference the relocated units (if necessary).  These clients are not
    -- copied but are modified in place.

    -- The new suppliers are units that are to replace the current
    -- suppliers of the units to be relocated.

    -- If an object specified by a client or supplier parameter is a world
    -- or directory it implies that all nested units are to be considered
    -- for the category.  An activity may be used to specify the worlds. The
    -- specified client and supplier sets may be larger than needed. Only
    -- the actual clients and suppliers will participate in the relocation.

    -- If the source unit is equivalent to the corresponding unit in the
    -- destination world, no copy takes place.  If the destination context
    -- is consistent with the present context of a unit, it is copied and
    -- modified preserving its present compilation state.  If the
    -- destination context is not consistent with present state of a unit,
    -- it is copied as source (if allowed) and recompiled (if requested)

    procedure Modify (Units : String;
                      New_Suppliers : String;
                      Allow_Demotion : Boolean := True;
                      Recompile : Boolean := True;
                      Action_Id : Action.Id := Action.Null_Id;
                      Status : out Successfulness);

    -- Modifies the specified units to look as if they were compiled against
    -- the units in the designated set of new suppliers.  If the suppliers
    -- are not equivalent to the present suppliers of a unit, the unit is
    -- demoted (if allowed) to source and recompiled (if requested).

    -- If a non-null action id is passed, the copy/modify is performed as
    -- an atomic operation, which fails unless all units can be copied/modified
    -- (and recompiled if requested).
end Relocation;
```

Figure 5-2: Preliminary Relocation Package - Part 2

The MODIFY portion of relocation can be used when updating the imports of a view. One

can assume that a good number of the units in the new Spec View are relocated copies of the units in the old Spec View. These units are easily identified because they have the same edit time stamp. The units in the new Spec View that have been edited since they were relocated are called ***differentiated units***. The Spec View was probably spawned to build these differentiated units.

When `View.Import` imports a new Spec View into a view, any unit that references a differentiated unit (and its clients) must be demoted to source and recompiled. Other units need only be MODIFY'ed to reference the newly imported Spec View.

5.5.2. Compatible Spec View Changes

A Spec View change is *consistent* if it can be blessed by Change Analysis. A change is *compatible* if each client that was compiled against the Spec View before the change will, without being recompiled, still execute correctly when loaded with an associated Load View. The Subsystem Tools allow only compatible changes to a Spec View. When an incompatible change must be made, a new Spec View must be spawned.[16] In Delta, more kinds of changes are compatible compared to Gamma, so a Spec View it can be used longer without the need for a time and space consuming spawn.

Because of the hard pointers in Diana and the coding strategy of the Delta system, a change to a declaration in a Spec View is a compatible change if

- The change is the addition or deletion of a library unit or it is an incremental change to the specification of a library package, *and*
- The change is consistent with all *unfrozen* clients of the library unit, *and*
- The declaration has no installed direct referencers, frozen or not.

A change to a source unit or to a unit that has no clients is always a compatible change.

5.5.3. Incompatible Spec View Changes

Incompatible changes cannot be made to Spec Views that have installed clients. The clients must be demoted or a new Spec View must be spawned in which to make the changes.

When a new Spec View is spawned to make an incompatible change, new views of its clients usually must also be spawned where they can be changed to match the new spec. On the other hand, some clients do not need to be spawned because they are in a View

[16] Since the spawned view has no installed clients initially, by the precise definition of compatibility, any changes to the newly spawned Spec View are now compatible.

that can be modified.[17] It may also be the case that a developer does not want all clients to be affected immediately.

5.5.3.1. Synchronized Subsystem Development

In Gamma, the task of spawning client views to accommodate incompatible changes is entirely manual. In Delta, activities will be used to identify the subsystems that are to participate in *synchronized subsystem development*, calling out both the Spec and Load Views to use for each subsystem. Activities will be accepted by **View.Spawn**, **View.Make**, and others; these commands will apply to all the views designated by the activity.

If the supplied activity identifies a released or frozen view, a copy of that view will be spawned automatically before being changed. The supplied activity will be updated to reference the spawned view. The **State** directory of each view will contain a file that maps an activity file name into a pattern for naming views automatically spawned from the view.

5.5.3.2. Unsynchronized Development

In synchronized subsystem development, the developer is willing to state *a priori* what dependent subsystems are to be affected by the changes he is about to make, and he is willing to wait for views to be spawned in his clients before proceeding with his changes. By doing this, he is able to take advantage of the incremental compilation facilities of the system as he makes his incompatible changes.

Many developers will not want to go to all of this trouble just to get started on a change. They may go through many private revisions before they get one that is worth exporting.

The Delta system will also support this style of *unsynchronized subsystem development*. At the time an incompatible spec change is to be made, just the view to be changed (and

---

[17] Suppose I have a system composed of three subsystems ***A***, ***B***, and ***C***. The Spec and Load Views of ***A*** and ***B*** are all clients of ***C***, while only the Load Views of ***A*** are clients of ***B***. I need to make an incompatible change to subsystem ***B***, so I spawn a new Load View for ***B*** in which to make the changes, and I spawn a new Spec View for ***B*** to export the changes. Because the Load View for ***A*** is a client of ***B***, I must also spawn a new Load View for it and recompile the units affected by the changes made to ***B***. At this point, the modified version of the system shares subsystem ***C*** and the Spec View of subsystem ***A*** with the original version of the system since these are not affected by the changes made so far.

I now discover that I must make an incompatible change to subsystem ***C*** to complete the task I started. I spawn a new Spec View and a new Load View for ***C***, and because it is a client of ***C***, I spawn a new Spec View for ***A***. Since I already have new Views for subsystem ***B*** and a new Load View for subsystem ***A***, I do not have to create more copies to complete the task. All I have to do is change the Diana pointers in those Views to point to the new Spec View for subsystem ***C*** instead of the old one (i.e., perform only the MODIFY half of relocation).

maybe a test world) is spawned. The spawned view has no clients, so changes can be made freely until it is tested and ready for release.

After the incompatible Spec View has been released, each client can import it when it's convenient. In Gamma, importing a new Spec View meant spawning a new view, but in Delta, imports to an unreleased, unfrozen view can be changed using **View.Import**.

In unsynchronized development, the only units that are recompiled are those dictated by conventional obsolescence rules. The incremental compilation capability is unadvantageous.

5.5.4. Activity Stacks

Based on suggestions from current users of Gamma and because of the wider use of activities in Delta, extensions to Gamma activities are planned for Delta. The notion of a job activity is replaced by the notion of a *job activity stack*. Tools that need to "look through" an activity search the activities on the job activity stack from top to bottom until they find one that has an entry for the subsystem that they need.

The stack is at most five deep and is stored in the job profile. The deepest entry in the stack is logically the *system default activity*, which is actually always stored under the name **!Machine.System_Default_Activity**. On top of that are one or two activities associated with the session, which are defined by session attribute switches. During execution of the job, entries may be added to or removed from either the session or the job portion of the stack. Changes to the session portion persist after the job terminates. The job portion is rebuilt each time a job starts.

When a job is initiated, an activity associated with the window that initiated the job is pushed on top of the session entries. The activity associated with a window is the activity that was used to find the object displayed in the window. Many windows will have no associated activity.[18] The user can also associate an activity with a window explicitly.

Directory.Naming also supports some new abbreviations for naming views of a subsystem via an activity. They look something like this.

- To name the Load View specified by an activity, use the form *subsystem-name*'L(*activity*).
- To name the Spec View specified by an activity, use the form *subsystem-name*'S(*activity*).

[18] At this writing, nothing in the system is known to form such an association or depend on its being there, but differential worlds would have to when and if they are implemented.

- To name the Load View specified by the current job activity, use the form *subsystem-name*'L.
- To name the Spec View specified by the current job activity, use the form *subsystem-name*'S.

5.5.5. Moving Views Between Machines

The code archive package provides substantial performance improvements for distributing coded subsystems between machines.

Ada units cannot be relocated to a different machine.

If only compatible changes are being made, then the **View.Merge** command could be used to make the incremental view changes after the new units have been moved to the target machine.

5.5.6. Subsystem Operations

Here are the scenarios presented in Section 1.5 as they would be executed in Delta.

For synchronized development, creating a "workspace" involves creating a new activity that names the subsystems on the machine that constitute the system. The activity points initially to a frozen release of each subsystem in the system. It can be created in most cases by making a copy of the activity used to develop the baseline version.

For unsynchronized development, creating a "workspace" involves spawning one view. The activity contains only the subsystem being changed.

CHANGING BODIES
: Spawn a view if a mutable one is unavailable; update the workspace activity. Edit and recompile the body in the mutable view. Load the system under the workspace activity. Relocate the changed body back to the main view when done.

CHANGING SPECS
: Same as CHANGING BODIES. Reintegration will be more expensive since clients will be obsolesced when the changes are relocated back to the main view.

EXPORTING COMPATIBLE CHANGES
: Make and test changes in a Load View. **View.Merge** the spec release into the associated spec view, which propagates obsolescence to unfrozen dependents. **View.Make** all obsolesced units using the workspace activity. Then load and execute the system using the same activity.

EXPORTING INCOMPATIBLE CHANGES

View.Spawn a copy of the Spec View to be changed using the workspace activity (Spec View slot). The units in the dependent closure of the spec are relocated to their respective views. Client views are spawned as specified by the workspace activity. Make the change to the spec and propagate obsolescence to the dependents who are affected. **View.Make** the spec, its body and the obsolesced units. Reload and execute using the workspace activity.

DISTRIBUTING RELEASES

Code_Archive.Save the view. **Code_Archive.Restore** on the new machine.

6. Summary of Changes for Delta

- **Ada_Base**

  1. Add **Diana.Sm_Operator** values to operators defined in the **System** package. (DIANA)

- **Machine_Interface**

  1. Add I/O exception numbers and family values to **Exception_Names** package. (ACCESS CONTROL)

- **Kernel_Debugger**

  1. Add image functions for new exception representation for I/O and access exceptions to **Current_Exception.Name** and equivalent. (ACCESS CONTROL)

- **Environment_Debugger**

  1. Add image functions for new exception representation for I/O and access exceptions to **Current_Exception.Name** and equivalent. (ACCESS CONTROL)

- **Om_Mechanisms/Basic_Managers**

  1. Add storage of ACLs to segmented heap object manager. (ACCESS CONTROL)

  2. Add parameter to generic to indicate whether access checks should be done or not. (ACCESS CONTROL)

  3. Add code to do check and return bad status if access fails. (Alternative implementation places check in Directory.) (ACCESS CONTROL)

  4. Look at other manager operations (manager.operate) to see if they need access checking. (ACCESS CONTROL)

  5. **Job_Manager**: Store user id for access control purposes. Provide operations to set and get it. (ACCESS CONTROL)

  6. Add **Access_Implementation** package(s). Include cache of world default ACLs for creation and owner lists. (ACCESS CONTROL)

  7. Add **Group_Implementation** package for storing and managing short group numbers. (ACCESS CONTROL)

8. Change DDB manager to build and use the distributed dependency database. (DDB)

9. Add entries to DDB manager to make changes in relations after relocations. (RELOCATION)

- **Ada_Management**

1. Add instantiation parameter indicating desire to perform access check for segmented heap OM generic. (ACCESS CONTROL)

2. Make appropriate changes so attribute spaces get the right ACLs and no access restrictions. (ACCESS CONTROL)

3. Creation operation needs to be able to find the correct enclosing world default ACL, and check owner access. (ACCESS CONTROL)

4. Add top-declaration database (DDB) and image attribute spaces (EDITOR)

5. Add COPY and MODIFY procedures for relocation. (RELOCATION)

6. Export iterators for maps within attribute spaces (RELOCATION)

7. Add **Diana.Sm_Operator** enumeration and attribute. (DIANA)

8. Add attribute **Diana.Lx_Line_Count**. Change **Dirty_Tree** package to do the "right thing" with it. (EDITOR)

9. Add **Diana.Is_In_List** predicate. (EDITOR)

10. Redefine nodes to reduce the size of Diana spaces (PERFORMANCE)

- **Directory**

1. Add procedures for releasing Ada objects. (COMPATIBILITY)

2. Add procedures for reserving and freeing Ada objects (CMVC)

3. Probably some other changes, but don't know exactly what. (ACCESS CONTROL)

4. Add subclasses and efficient recognition of subsystems, views and differential views. (SUBSYSTEMS)

5. Change naming to recognize the **'L** and **'S** attributes. (SUBSYSTEMS)

6. Change **Promote** and **Demote** to call CG coupler to update Code Database after each promotion to coded or demotion from coded. (CODE DATABASE)

7. Add subprograms for getting/setting target dependent information into Compatibility Database Target Map **Compilation_Coupler** package. (COMPATIBILITY)

- **Code_Generator**

1. Change code generation for exception handlers for Io_Exceptions to handle flavored exceptions. (ACCESS CONTROL)

2. Add calls to Compatibility Database (COMPATIBILITY)

3. Add calls to Code Database. (CODING)

- **Semantics**

1. Build local reference map within Diana space and save it through compaction (DDB).

2. Determine and set proper value for **Diana.Sm_Operator** attribute on calls to predefined operators. (DIANA)

3. Attribute calls to derived subprograms according to Diana manual. (DIANA)

4. Add calls to get declaration number for a declaration from the Compatibility Database. (COMPATIBILITY)

- **Input_Output**

1. Probably some changes relating to getting opens and access checks done in the right place. (ACCESS CONTROL)

2. Probably some changes returning gracefully from operations after an access error status is returned. Need to raise **Access_Error** at some point. (ACCESS CONTROL)

3. Need to check owner access to world on create. Also need to find enclosing world default ACL on create. (ACCESS CONTROL)

4. Change raises of Io_Exceptions to raise appropriate flavors at various points in the code. (ACCESS CONTROL)

5. Change **Profile** to export fields for an Activity stack. (ACTIVITIES)

6. Change **Activity** and **Profile** to export the appropriate operations on activity stacks. (ACTIVITIES)

- **Tools**

1. Add **Access_List** and **Access_List_Tools** package. (ACCESS CONTROL)

2. Add **ACL_Utilities** package (unless native). (ACCESS CONTROL)

3. Addition of **Group** and **Group_Tools** packages. (ACCESS CONTROL)

- **Object_Editor**

1. **Library_Object_Editor** implementation of display and set operations for access lists. (ACCESS CONTROL)

2. **Library_Object_Editor** addition of interfaces to allow call to set access list. (ACCESS CONTROL)

3. **Library_Object_Editor** addition of elision level (or equivalent) to control display of access lists. (ACCESS CONTROL)

- **Ftp_Interface/Network**

1. Add FTP server code to set job identity for remote file operations. (ACCESS CONTROL)

2. Add FTP code to restore files properly relative to access control unless this is done via standard I/O calls. (ACCESS CONTROL)

- **Archive**

1. Change Source_Archive to include ACLs in the save. (ACCESS CONTROL)

2. Change Source_Archive to add options for replacement of ACL on restore. (ACCESS CONTROL)

3. Change Source_Archive to set ACL after restoration of object. (ACCESS CONTROL)

- **Native_Debugger**

1. Have Memory_Dump check for System_Maintenance access. (ACCESS CONTROL)

2. Worry about calls to code that constructs Diana pointers. (ACCESS CONTROL)

- **Subsystem_Tools**

1. **View.Spawn** will also take an activity and spawn the views designated by it.

2. **View.Spawn** will use relocation to be more efficient.

3. Add **View.Merge** and **View.Compare**.

4. **View.Import** can be used to refresh the imports for an uncontrolled view. Differentiated units in the new view to be imported cause obsolescence in the importing view.

5. **View.Import** also takes an activity and will update imports for all views in the activity.

6. **View.Make** can take an activity, which will cause it to make all of the views specified therein.

7. **View.Make** will also attempt to make obsolete aliases consistent with their view by copying and recompiling them into the view.

- **Compilation_Commands**

-

1. All commands accept an activity, which specifies a set of worlds to operate on.

Others

1. Any place a system task (VP=4) does an open, create, etc., explicit access checks need to be added because the access control for the system task will be based on Operator identity. (ACCESS CONTROL)

2. **Program.Run_Job** needs to be extended to allow setting the identity of the new job. (ACCESS CONTROL)

- Changes relating to determination of user identity. There may be places in the system where the current session is used to determine the user identity. These would need to be changed or an additional identity for a job for access control purposes would need to be added.

Appendix I. Sm_Operator

```
BOOLEAN_eq
BOOLEAN_ne
BOOLEAN_lt
BOOLEAN_le
BOOLEAN_gt
BOOLEAN_ge

BOOLEAN_and
BOOLEAN_or
BOOLEAN_xor
BOOLEAN_not

INTEGER_eq
INTEGER_ne
INTEGER_lt
INTEGER_le
INTEGER_gt
INTEGER_ge

INTEGER_plus
INTEGER_neg
INTEGER_abs

INTEGER_add
INTEGER_sub
INTEGER_mul
INTEGER_div
INTEGER_mod
INTEGER_rem

INTEGER_exp

FLOAT_eq
FLOAT_ne
FLOAT_lt
FLOAT_le
FLOAT_gt
FLOAT_ge

FLOAT_plus
FLOAT_neg
FLOAT_abs

FLOAT_add
FLOAT_sub
FLOAT_mul
FLOAT_div

FLOAT_exp

Universal_Integer_Real_Mul
Universal_Real_Integer_Mul
Universal_Real_Integer_Div

Universal_Fixed_Mul
Universal_Fixed_Div

FIXED_eq
FIXED_ne
FIXED_lt
FIXED_le
FIXED_gt
FIXED_ge

FIXED_plus
FIXED_neg
FIXED_abs

FIXED_add
FIXED_sub
FIXED_Integer_Mul
Integer_FIXED_Mul
FIXED_Integer_Div

DISCRETE_eq
DISCRETE_ne
DISCRETE_lt
DISCRETE_le
DISCRETE_gt
DISCRETE_ge

String_eq
String_ne
String_lt
String_le
```

```
String_gt
String_ge

String_String_cat
String_Character_cat
Character_String_cat
Character_Character_cat

One_DIM_Array_eq
One_DIM_Array_ne
One_DIM_Discrete_Array_lt
One_DIM_Discrete_Array_le
One_DIM_Discrete_Array_gt
One_DIM_Discrete_Array_ge

One_DIM_Boolean_Array_not
One_DIM_Boolean_Array_and
One_DIM_Boolean_Array_or
One_DIM_Boolean_Array_xor

One_DIM_Array_Array_cat
One_DIM_Array_Element_cat
One_DIM_Element_Array_cat
One_DIM_Element_Element_cat

Multi_DIM_Array_eq
Multi_DIM_Array_ne

Record_eq
Record_ne

Access_eq
Access_ne

Formal_Access_eq
Formal_Access_ne

Private_eq
Private_ne

Formal_Private_eq
Formal_Private_ne
```

Where

```
BOOLEAN   = Boolean | Other_Boolean
INTEGER   = Integer | Short_Integer | Short_Short_Integer |
            Long_Integer | Long_Long_Integer | Formal_Integer |
            Universal_Integer | Other_Integer
FLOAT     = Float | Short_Float | Short_Short_Float |
            Long_Float | Long_Long_Float | Formal_Float |
            Universal_Real | Other_Float
FIXED     = Duration | Formal_Fixed | Other_Fixed
DISCRETE  = Character | Enumeration | Formal_Discrete
DIM       = Dim | Dim_Formal
```

Index