```
  SSSSSSSS    UU      UU    BBBBBBBB        SSSSSSSS     YY      YY
  SSSSSSSS    UU      UU    BBBBBBBB        SSSSSSSS     YY      YY
SS            UU      UU    BB      BB    SS             YY      YY
SS            UU      UU    BB      BB    SS             YY      YY
SS            UU      UU    BB      BB    SS               YY  YY
SS            UU      UU    BB      BB    SS               YY  YY
  SSSSSS      UU      UU    BBBBBBBB        SSSSSS           YY
  SSSSSS      UU      UU    BBBBBBBB        SSSSSS           YY
        SS    UU      UU    BB      BB            SS         YY
        SS    UU      UU    BB      BB            SS         YY
        SS    UU      UU    BB      BB            SS         YY
        SS    UU      UU    BB      BB            SS         YY
SSSSSSSS      UUUUUUUUUU    BBBBBBBB      SSSSSSSS           YY
SSSSSSSS      UUUUUUUUUU    BBBBBBBB      SSSSSSSS           YY




TTTTTTTTTT    XX      XX    TTTTTTTTTT                     333333
TTTTTTTTTT    XX      XX    TTTTTTTTTT                     333333
    TT        XX      XX        TT                       33      33
    TT        XX      XX        TT                       33      33
    TT          XX  XX          TT                               33
    TT          XX  XX          TT                               33
    TT            XX            TT                             33
    TT            XX            TT                             33
    TT          XX  XX          TT                               33
    TT          XX  XX          TT                               33
    TT        XX      XX        TT             ....      33      33
    TT        XX      XX        TT             ....      33      33
    TT        XX      XX        TT             ....        333333
    TT        XX      XX        TT             ....        333333



*START* Job DESIGN Req #976 for EGB    Date 29-Apr-85  9:37:29 Monitor: //, TOPS
File RM:<SYSTEM.SPEC>SUBSYS.TXT.3, created: 26-Feb-85 13:12:10
        printed: 29-Apr-85  9:37:48
Job parameters: Request created:29-Apr-85  9:37:27   Page limit:261   Forms:NORMAL
File parameters: Copy: 1 of 1   Spacing:SINGLE   File format:ASCII   Print mode:ASCI
```

DRAFT 1

2.4. Subsystem Development Paradigm.

2.4.1. General.

While Ada and the Rational Programming Environment can support a wide range of programming methodologies and project management strategies, the language and the environment are particularly suited to those methodologies based upon techniques such as hierarchical decomposition, object-oriented design, levels of abstraction, information hiding, data abstraction, etc. In this section (2.4) we introduce the Rational Subsystem Paradigm, which is representative of a family of related methodologies that have been developed over the past decade. The methodology described here is tailored to Ada and the R1000, and provides a framework that can be adapted or extended to address the requirements of a particular project.

We will introduce both the methodology and associated programming environment support by considering key activities in various phases of the development life cycle. For purposes of discussion, we present a very simple view of the development cycle. In practice, development will be very iterative, and at different levels the same software will be in all of the phases described here. Thus, development activities overlap and the facilities discussed find use in every phase of development. We will consider the following phases:

a. Preliminary design.
b. Detail design and implementation.
c. Test and integration.
d. Maintenance and on-going development.

Note that we do not address requirements analysis, but begin with design tasks. After considering these phases we will briefly consider development in a distributed environment and support for multiple targets, two topics which will be addressed more fully later.

2.4.2. Preliminary Design.

2.4.2.1. Decomposing a System into Subsystems.

In the subsystem methodology, a large system is decomposed into a hierarchy of subsystems. For the moment, we will view each subsystem simply as a collection of one or more Ada packages which implement some portion of the system. The system should be decomposed in accordance with good design practices and software engineering principles. For decomposing large Ada systems into subsystems, it is particularly important to recognize two dimensions of decomposition. In the "vertical" dimension, it is important to decompose the system, or any portion of the system, into levels of abstraction, with separate subsystems for major layers. This layering results in a more manageable and maintainable system. In the "horizontal" dimension, the system (at any particular level of abstraction) should be modularized into logical entities, preferrably in an object-oriented manner.

Essentially, the decomposition of a large system into subsystems is an extension of the process of decomposing a large Ada program into packages.

In addition to reflecting good design practices, the decompostion of a system into subsystems must reflect organizational and project management considerations. For many projects, a subsystem will correspond to the amount of work that can be reasonably allocated to a single person, or to a small team. Distribution of activities between different development groups with differing expertise (and perhaps different geographic locations) may also influence the decomposition. If the system is to be bundled and unbundled in different product configurations, that separation should be reflected in the subsystem structure. Generally, a subsystem will serve as the field replacable unit for purposes of software repair, release and distribution. Other organizational constraints on system decomposition will vary according the to particular project and development team.

The decomposition into subsystems must identify the subsystems, define (at least at a high level) the contents of each subsystem and specify the interfaces between subsystems. Then the design effort can focus on the individual subsystems, although there will continue to be some evolution of the system structure as the design matures.

2.4.2.2. Subsystem Interfaces.

A subset of the packages in a subsystem will be exported. The visible parts of all the exported packages form the abstract interface which the subsystem presents to higher-level subsystems. This abstract interface should hide implementation details from higher-level subsystems, while completely capturing the facilities to be provided by the subsystem. Again, good design practices based on information hiding, abstraction, etc., should be appliec in designing subsystem interfaces.

A subsystem will import lower-level subsystems to use in its implementation. This "using" relationship, where one subsystem uses another in its implementation or specification, must form a strict hierarchy (no cycles). (Support for mutually referencing subsystems at the same level of abstraction, where the implementations refer to each other, is planned, but introduces elaboration and loading complexities which will be ignored for the moment).

2.4.2.3. Subsystem Design.

Once the interfaces have defined, it is possible to design the subsystem itself. The design of the individual subsystem should conform to good software engineering practices, but is largely driven by the specific application and the system design goals. Each subsystem should be designed to be indepently tested and maintained to the greatest extent possible.

Precisely specifying the abstract interface for a subsystem and then constructing the subsystem on top of other subsystems, brings us to the next phase of development.

2.4.3. Detailed Design and Implementation.

2.4.3.1. Subsystems as Libraries.

In the environment, each subsystem is represented as a single library. The library contains the specs which make up the abstract interface for the subsystem, contains the bodies for all those specs, contains other library units required to implement the subsystem, and contains a number of managed objects which store information relating to the subsystem.

Section 2.2 introduced the concept of a policy being associated with a library. Libraries used to implement the subsystem paradigm are created (implicitly or explicitly) with the configuration policy and are control points. The configuration policy initializes a set of state objects associated with the library, including a subsystem configuration, compilation switch files, elaboration information, and history information.

2.4.3.2. Subsystem Configurations.

Although not explicitly declared, creation of the library creates a configuration object associated with the library (see 2.2.3.3), known as the subsystem configuration.

Configurations are themselves objects, which have versions. Recall that a configuration is a mapping from objects to versions of the objects. The R1000 provides support for two important kinds of configurations, subsystem configurations and system configurations.

A (version of a) subsystem configuration defines a consistent view (release) of a single subsystem and can be viewed as a mapping from the objects in the subsystem to the version of those objects participating in this view of the subsystem. A release of a subsystem may not have been "released" in any formal sense, but rather represents a user visible version of the subsystem.

The subsystem configuration information for a library may be retrieved using operations provided on the library itself. The subsystem configuration object is not declared explicitly, so that only the configuration policy (and no user program) manipulates the configuration. The validity of the configuration information is vital to the integrity of the library.

2.4.3.3. System and Session Configurations.

A system configuration is not associated with a particular subsystem, but rather with a universe of subsystems. A (version of a) system configuration defines a consistent universe of subsystems and can be viewed as a mapping from the subsystems in the universe to the releases of those subsystems which are participating in this view of the universe. A system configuration is required for all operations involving more that one subsystem, and all command or program execution.

Each active session on the system has a session configuration, which is a system configuration specifying the users view of the universe. This determines the default version of all objects the user manipulates. Thus, a session supports a single consistent view of the universe including only libraries that are consistent with each other. In particular, the libraries in the session configuration must have been compiled against compatible versions of each other. Other versions can be accessed by explicitly specifying version names (possibly through the use of other configurations).

For a given subsystem it is possible to specify a default release. If such a default is specified, then referencing this subsystem with a system configuration which does not designate any release of this subsystem attempts to add the default version to the system configuration. If the default is compatible with the configuration, and the configuration is open for update, the operation succeeds and all references are directed to the default version. Otherwise, the operation fails, no release of the subsystem participates in that system configuration, and all references will fail. Note that this is a temporary binding, since only the open (temporary) copy of the configuration is updated. For a session configuration, that binding may be explicitly committed and made permanent, or may be explicitly updated to reflect a newer release. Usually, the binding will be temporary and last until logoff. In this way the user will tend to get the latest release of a subsystem at the time of first reference, and will keep a consistent view of that subsystem throughout the session.

In this example, we have created a new library, creating an initial (fairly empty) release. In creating the control point we had the option of specifying the name of the first release. Let us assume we named it KK_0, representing the first major release of a subsystem named Kernel. This release is automatically established as the release for our session, since we created the library. For existing libraries, we must explicitly establish the release for our session (which is preserved as session state). Operations on the library are with respect to the specified release. For example, the first operation on the newly created library might be to create several new Ada units. This would update the current subsystem configuration to reflect the new objects (and the current versions of those objects).

2.4.3.4. Compilation and Semantic Consistency.

A component of the subsystem state automatically created with the library is a set of switches. These switches are primarily to control compilation options, and are passed to any compilation that occurs in the library. Switches can be set on a per release basis. Consistent with the directory model (2.2.4), we can compile, elaborate and execute units in the subsystem. The configuration policy allows the system to view the library as a single set of units, and ignore version issues. The compilation facilities (both interctive and batch) use the session configuration to extract the release configuration, and then use the release configuration for compilation. Thus the release configuration must include all units (including imported units) required to establish the full compilation context.

The configuration policy maintains one extremely important invariant with respect to compiled (installed) units. This invariant is the basic property of a consistent subsystem configuration. All of the installed units in any given release are guaranteed to be semantically consistent. That is, for any two units A and B in the release, for any semantic attribute in A referencing some other unit C, semantic attributes in B must reference the same version of C (and so on recursively through the entire transitive closure). Furthermore, referenced version of C must be the version that appears in the release containing the versions of A and B under consideration.

2.4.3.5. Exporting Subsystem Interfaces.

We have assumed that the example library did not import units from other libraries. Further assume that all we have done, so far, is write the visible parts for this subsystem and compiled those specs. Then the release we have defined can be viewed as an export release which others could compile against, even though no implementation of this subsystem exists. For any spec which is intended to be exported from a subsystem to another subsystem, the user must incicate that it is to be exported by including a pragma Subsystem_Interface (note that for this purpose, units containing bocies of inlined procedures and macro-expanded generics, as well as completions of incomplete types from open private parts, would have to be exported and include the pragma subsystem_interface).

2.4.3.6. Freezing a Subsystem Release.

In order for this release to be imported by another subsystem, it must be frozen. Freezing a release puts it in a state where none of the versions of objects which belong to the release may be modified. At this point, the library Kernel might appear as follows.

```
    library Kernel is
        package A is separate;
        package B is separate;
    end;
```

In this example, the packages A and B (or Kernel.A and Kernel.B) form the abstract interface, and we have constructed a minimal frozen configuration for use by the clients of Kernel. Clients of the kernel could use this configuration for all compilation, but would not be able to execute until we had constructed a complete implementation release for the Kernel which client could use to construct a system configuration for execution.

2.4.3.7. Constructing a Subsystem Implementation.

To do any new work in Kernel, we must first spawn a new release. We can create a new release called KK_0_0, indicating that it is the first minor release of the KK_0 major release (the notions of major, minor and mirco releases are not part of the subsystem paradigm per se, but are used here as an explanatory device given that the audience is familiar with Rational release mechanisms). This operation will create a new version of the subsystem configuration, and establish that as the default for our session. Initially, this new configuration references the same versions of the same objects as the configuration (KK_0) constructed earlier.

Assume that in addition to adding bodies to A and B we plan to add a package C that is not to be exported. Adding these declarations creates a new (logical) version of the library itself so that the stubs can be added, since the previous version was frozen. The implementation configuration is updated to reflect the new version of the library and the addition of new units (A'body, B'body, and C). Further assume that the implementation must be written in terms of the lower-level subsystem Machine_Interface. We can update the context clause for the Kernel library to reference Machine_Interface. We now have in place the structural elements for the implementation of a first version of the Kernel subsystem. This results in the following view of the Kernel library.

```
    use Machine_Interface;
    Library Kernel is
        package A is separate;
        package B is separate;
        package body A is separate;
        package body B is separate;
        package C is separate;
        package body C is separate;
    end;
```

Note that another user, relying on the release for Kernel which we created earlier (KK_0), would still see the view of the library presented earlier, not this new view of the library that we are developing.

2.4.3.8. Importing.

Compilation of the new units would produce semantic errors at this point (assuming the new units in Kernel reference units in Machine_Interface), because the Kernel configuration we are using does not indicate which version of the units from Machine_Interface should be used. Recall that the release configuration must include the complete context for compilation in the library.

We can perform an import operation which augments the current release of Kernel with a release of Machine_Interface. If our session configuration already includes a release of Machine_Interface, that release will be used to augment the current Kernel release, unless we explicitly name some other release of Machine_Interface.

The system will first check that the designated release of Machine_Interface is frozen. Then all of the units in the Machine_Interface release which contain the Subsystem_Interface pragma are added to the Kernel release, by copying a modified form of the imported specs into a sublibrary and then updating the subsystem configuration to include those specs. Copying the specs into the Subsystem_Imports relativizes semantic references (with respect to the current library), and will fail if the specs are not consistent with other imported specs. Copying the specs also may prune the context clause and the private part for those targets which allow closed private parts (see 2.4.5.). Optionally, we could have specified that the specs were to be copied in on demand, rather than all at once.

Having augmented the Kernel subsystem configuration, compilation of the new units (which import units from Machine_Interface) will proceed properly. At this point the Kernel library and the Subsystem_Imports library look like the following.

```
    use Machine_Interface;
    Library Kernel is
        package A is separate;
        package B is separate;
        package body A is separate;
        package body B is separate;
        package C is separate;
        package body C is separate;
        Library Subsystem_Imports is separate;
    end Kernel;

    Library Subsystem_Imports is        -- Kernel.Subsystem_Imports
        Library Machine_Interface is separate;
    end Subsystem_Imports;
```

2.4.3.9. Local and Global Diana tools.

There are two classes of tools that use Diana on the system, local tools and global tools. The first class relies only upon the subsystem configuration and references only units within the library (including the nested spec libraries). The compiler is probably the most important member of this first class. The second class uses a system configuration to look through references to specs to reach the "real" version of the particular unit. The editor and debugger are members of the second class. (Some discussion remains on whether the editor is in the second class or is in the first class with operations that explicitly look through to the "real" version).

In this model, compilation is more efficient, but relies only upon local information. Debugging and editing make less frequent use of semantic attributes, but provide a complete and consistent view of a more global universe. For example, using DEFINITION in the editor will take the user to the real spec, so that he may than use OTHER_PART to see the body. The user would only see the (truncated) specs in the spec sublibraries in the case where his session configuration does not include a real version of the referenced subsystem.

One can proceed in this manner, designing and implementing the subsystem as a consistent set of Ada units. As soon as a first (possibly incomplete) release of the subsystem has been compiled, one will want to test and debug that release before proceeding to add functionality or otherwise change the subsystem. The issues of execution, testing and debugging are addressed in the following section.

2.4.4. Test and Integration.

We will continue the example of the previous section to illustrate the model for test and integration. Let us assume that the Kernel subsystem we have constructed is to be a SHARED_ONLY subsystem, and that the Machine_Interface subsystem is a SHARED_ONLY subsystem which the owner has already elaborated (i.e., our session configuration references an elaborated release of Machine_Interface). Let us further assume that our initial test plan involves the following steps:

Step 1. Construct a Kernel Test library which will hold test programs as we write them.

Step 2. Execute several simple commands which exercise facililites from Machine_Interface that the Kernel needs.

Step 3. In the Kernel Test library, construct a more comprehensive test which excercises key Machine_Interface facilities in a manner similar to actual kernel operation.

Step 4. Elaborate the Kernel subsystem, using the debugger as needed to analyze problems.

Step 5. Having successfully elaborated the Kernel, execute several simple commands which provide a preliminary indication that the kernel has initialized itself properly.

Step 6. In the Kernel Test library, construct a more comprehensive test which verifies proper kernel initialization checks correct operation of simple facilities.

Step 7. The owner of Machine_Interface has produced a new release and would like to test his subsystem using the Kernel and Kernel Tests produced earlier.

These steps are only a few of the many required, but they illustrate key characteristics of the subsystem paradigm. Even before discussing the individual steps, it is clear that the R1000 supports an interactive and incremental approach to test and integration.

2.4.4.1. Constructing a Test Library.

First we construct our test library nested within the kernel library. This test library will consists of test programs which execute on top of the exported Kernel specs. Thus the Kernel Library now looks like the following.

```
    use Machine_Interface;
    Library Kernel is
        package A is separate;
        package B is separate;
        package body A is separate;
        package body B is separate;
        Library Subsystem_Imports is separate;
        library Test is separate;
    end Kernel;
```

2.4.4.2. Establishing the Test Configuration.

Step 2 in our plan involves executing a few simple commands to exercise facilities in Machine_Interface that the Kernel would rely upon. All execution requires a system configuration, and in this case our session configuration is adequate since it should include an elaborated Machine_Interface. The key issue here is that we must have properly established our session configuration so that it includes an elaborated release of Machine_Interface which supports the facilities we need for testing the Kernel.

This may be an unnecessary ("old granny") step, but much time is wasted in test and integration because of improperly established test configurations. When bringing up new and untested code, one wants to remove all other sources of error, in order to quickly track down real problems and not chase ghosts. While the subsystem paradigm is designed to help eliminate many of these problems, cross subsystem coordination requires manual intervention and is subject to some error. Therefore, the system facilitates quick, interactive verification at key steps, so that errors which do occur can be detected early in the process. Once the user is confident that his session configuration is being established properly, this step may be eliminated.

Recall that for SHARED_ONLY libraries the library is elaborated as a whole. If we visit one of the specs in Machine_Interface, it should be in the elaborated state at this point. If it is not, we must (possibly in cooperation with the owner of Machine_Interface) either update our configuration to reflect an already elaborated release of Machine_Interface, or create an elaborated release for our own use.

We can then write short commands which call specs exported by machine interface. Even if these test commands do not exercise the most interesting facilities, they give us quick feedback that Machine_Interface is properly elaborated. If there are new facilities that have just been added for our use, we might try to exercise those to make sure they at least exist. If any errors are uncovered, we can construct minimal test cases which produce the problem, and then work with the owner of Machine_Interface to resolve the problems.

2.4.4.3. Constructing a Test Program.

Step 3 is a continuation of Step 2. While Machine_Interface presumably has a set of test programs, we may want a test program that further verifies specific properties that we depend upon. We add this to the Kernel Test library and compile against the specs imported from Machine_Interface. The test library is not a shared library, and we can call the test program as soon as it has been coded. The library Kernel.Test would look like the following at this point.

```
    use Machine_Interface;
    Library Test is
        procedure Test_KMI is separate;
    end;
```

From a command window we can execute Test_KMI and review the results. If the test produces a log file we can save a "golden" copy of the file in the test library, and have each execution compare its results to the golden results. We can later add test drivers in the test library which invoke this test along with others and produce a summary of the results.

2.4.4.4. Elaboration of a new Subsystem.

Step 4 actually involves executing the new code we have written in the Kernel library. We can elaborate the current release of the Kernel, or we can produce a new release of the Kernel which differs from the previous only in that we will promote it to elaborated. Elaboration information is retained as part of the release automatically. When elaborating a SHARED_ONLY subsystem we have the following three options with respect to the persistence of the elaboration.

1. The subsystem remains elaborated from the time it is

explicitly elaborated until it is demoted or until the system goes down, whichever comes first.

2. The subsystem remains elaborated until it is demoted, and it is automatically elaborated after a crash at the time the system is brought up.

3. The subsystem remains elaborated until it is demoted, and it is automatically elaborated after a crash at the time of the first reference to the subsystem that requires it to be elaborated.

While we are first debugging the subsystem, option 1 is most appropriate, since we do not expect anyone else to be using the subsystem. Once we have released a version of the Kernel for widespread use, we must chose between option 2 and option 3. The system elaborates the new subsystem with respect to a particular system configuration, in this case our session configuration. Since the constructions of any system configuration verifies subsystem compatibility, we are certain that we are elaborating against compatible versions of the lower level subsystems. This consistency checking is addressed further in 2.4.5., since the main issues deal with upward compatible changes.

If any problems are encountered with the elaboration of the Kernel (quite likely if there is much new code), we can use the interactive debugging facilities to investigate the problems. Interactive debugging is further discussed in section xxxx. Once we have successfully elaborated the subsystem, we can move on to the next phase of testing.

2.4.4.5. Test and Release of a new Subsystem.

Step 5 in our test plan involves executing a few commands to check that the newly elaborated Kernel is properly elaborated and that basic facilities work properly. This gives us quick feedback and lets us interact with the subystem directly to determine its health. If the Kernel exports operations which perform internal consistency checks, those are probably the first operations we invoke. From a command window, we can invoke any operation exported by any package in the subsystem, including exported packages (Kernel.A and Kernel.B) and internal packages (Kernel.C). Having executed some of the visible operations, perhaps with the debugger, we move on to the next step.

Step 6 involves adding to the Kernel Test library a more comprehensive test of the initialization of the Kernel. The test library would now look like the following.

```
    use Machine_Interface;
    use Kernel;
    Library Test is
        procedure Test_KMI is separate;
        procedure Test_Initialization is separate;
    end;
```

Once this test program has been coded, we can execute it and determine the results. Again this test can be structured so that the results are reproducibly verifiable. We can continue in this vein, executing simple commands that excercise the Kernel directly, writing more test programs, and building up our test library, and then cycling back to design and implementation of additional Kernel facilities.

In preparation for moving on to step 7, let us assume that these first two test programs work correctly abd we and freeze an elaborated release of the Kernel and an unelaborated release of Kernel.Test. We can establish these releases as the defaults, which other users will see if they do not specify particular releases. In a very minimal sense, the subsystem has been released. A more formal release process can be supported, including more comprehensive test and documentation procedures. For the moment, assume we are constructing an informal, internal release.

2.4.4.6. Recombinant test and integration.

Now assume that while we were developing the Kernel, the owner of Machine_Interface has just frozen a new release of his subsystem. In addition to running his own regression tests, he now has a customer (the Kernel) who has code which executes on top of Machine_Interface. He may wish to run the Kernel tests, since they may actually exercise Machine_Interface in more or different ways.

The subsystem paradigm allows the combination of different versions of subsystems that have compatible interfaces. Thus the owner of Machine_Interface should be able run his new subsystem with the previously released Kernel, which is known to execute properly with the previous release of Machine_Interface. This property will hold for the R1000 as an execution vehicle in the face of a fairly wide range of upward compatible changes in the different versions of the subsystem specs (including different private parts, adding new functions, etc., see 2.4.5.). Non-R1000 targets which follow fairly simple conventions for linking and loading may also be able to support this aspect of the subsystem paradigm, although a smaller (possibly empty) set of upward compatible changes will be supported (see 2.5.).

In our example, the session configuration for the owner of Machine_Interface currently includes the new release of Machine_Interface. We have released Kernel and Kernel.Test libraries, so he can get those releases automatically. However, in this case he cannot add the elaborated release to his session, because it is elaborated against a different Machine_Interface. However, if he designates the Kernel subsystem and demotes it to coded, he implicitly spawns a new release (which is added to his session configuration), which can be elborated on top of his new Machine_Interface. Then he can execute the Kernel tests and ensure that the new release of Machine_Interface supports the current release of the Kernel. He may even include this as part of the standard regression testing procedure for new releases of Machine_Interface.

Note that relatively little interaction is required between the developers of different subsystems. There may be many releases of Machine_Interface or many releases of the Kernel, but they need not be coordinated as long as they are spec compatible. In practice, interface issues or subtle bugs may arise, requiring joint debugging and coordinated fixes.

So far we have focused on testing an individual subsystem in the context of other subsystems. System testing can simply be viewed as testing the "top" subsystem in terms of all the lower subsystems. For system testing, and even subsystem testing, one can create configuration objects which capture meaningful configurations of subsystems. The system will enforce consistency and ensure that the systems are configured properly.

More on regression testing, system test, test tools, etc. Someday.

2.4.5. Maintenance and On-Going Development.

So far we have been considering very simple scenarios involving new subsystems with very few versions. Much more complex issues arise during maintenance and on-going development of a subsystem which has one or more released version which must be supported. In particular, support for incremental and upward-compatible changes becomes essential (since one is "fixing" existing code rather than writing new code), source managment becomes a major issue, and tracking of history and other information becomes more important. Let us continue with the example of previous sections to illustrate these issues.

2.4.5.1. Incremental and Upward Compatible Changes.

Recall that at the end of step six in the previous section we froze a version of the Kernel. Now assume that based on our first round of incremental testing we wish to fix several problems and add several facilities in the package Kernel.C. We spawn a new release, KK_0_1. All of the units are initially shared with the previous release (KK_0_0). If we make changes in package C, only that package will have a new version which is in the new Kernel release and not in the old release. The Kernel subsystem configuration we are using is updated to reflect these new versions of package C. When we have made our changes, probably using the incremental compilation facilities, we can test them immediately by elaborating this new release and repeating a form of the test cycle described in the previous section.

Changing exported specs provides a more interesting example. Assume that several clients are now using the frozen version of Kernel released earlier (KK_0_0), and we wish to change exported Kernel specs to produce a new release (KK_0_1) that is compatible with the code the clients have produced, but which includes new facilities that support future client development. In particular, we wish to be able to have the clients run against KK_0_1 (when we release it) without the clients having to recompile any of there subsystems. In fact, old frozen versions of the clients which ran against KK_0_0 should run against KK_0_1, while new versions of client code may be developed using the new facilities of KK_0_1.

The system actually goes further, in that clients can import the new specs into their subsystems without causing any recompilation. The only compilation involved is that associated with changes the client might make to use the new facilities provided by the new kernel specs.

We cannot demote the specs to source and make arbitrary changes and have the changes be upward compatible. However, if we make incremental changes, the system will produce a new version (reflected in the KK_0_1 subsystem configuration) and properly maintain the version so that it is upward compatible. In particular, it will not allow incremental deletions without informing us that such a change would not be upward compatible. Incremental additions would be supported, and the system would properly maintain semantic and code generator attributes such that the change is upward compatible.

For the R1000, most additions are upward compatible, and if we have specified that the spec has a closed private part, then no client was allowed to rely upon the information in the private part and all changes which affect only the private part are then upward compatible. (The complete set of rules for upward compatibility for each target (including the R1000) will specified separately.)

The system implements upward compatibility by restricting the set of changes allowed in the spec, and managing specific compiler attributes. Proper management of compiler attributes requires that all compatible versions of a spec be maintained (especially modified) on a single machine which maintains the set of related versions of the specs. Each compatible version of a spec is related to the original and shares a compatibility key which is used for cross-subsystem compatibility checking. Anytime a system configuration is constructed, exported specs are checked against imported specs (for all subsystems in the configuration) to ensure that they have the same compatibility key.

In our example, we could add changes to Kernel.A and Kernel.B (exported packages) using the incremental facilities of the environment, and then test those changes as described above. Once we are satisfied that the changes have been made properly, we can freeze and release KK_0_1. If we make that release the new default, and encourage clients update their system configurations properly, then users will be using KK_0_1, which should support all old facilities, plus the new facilities we have provided.

2.4.5.2. Source Management.

The facilities discusses so far are adequate for supporting a single development path, where there is a sequence of releases, each release superceding the previous release. Given that the system has been decomposed into small subsystems where a very small team is working on each subsystem, and given that only a single development path need be supported, no additional source management support would be required. However, in the face of maintaining one or more released versions while supporting one or more active new development paths involving more people and possibly multiple target machines, the user will required more substantial source management support.

basic model is to know which devel paths are related, inform user when he is making a change that it will impact other paths, support policies that restrict changes which impact other paths, and support merging and (semi)automatic propogation of changes to other paths.

source mgmt -- serial releases, divergence and parallel devel, multi (lower-level) specs and multi targets.

2.4.5.3. History.

keep all relevant info at object, release, subsystem, and project level. support construction of tools that operate on this info.