# The Minimal Sierra Environment: Product Description

**Charles Haley**
**Version 2.2**

## 1 Introduction

This document describes the Sierra product from the user's perspective. Descriptions of the implementation are avoided except in cases where the user must know these details. Another way of saying this is that this document is written from a marketing perspective.

The point of this exercise is to converge on a description of the first release of Sierra and as such this document is not intended to be a description of a long term vision. We very much need help from lots of people to converge a description of a good product that meets our delivery goals (beta July 1992). Reviewers, please keep this in mind.

Note also that the vocabulary used in this paper probably isn't what we really want to use. The terms used here, however, are probably understood by reviewers. Future revisions will introduce new, improved, terms along with their definitions.

Before starting the description we must first agree on some these terms. We must also agree on an outline the problem domain. This detail is provided in section 2. Sections 3 through 6 describe some functionality assumptions, the user interaction model and a synopsis of some functional requirements. Section 7 contains general questions; note that the other sections contain questions as well.

## 2 The Problem Domain

### 2.1 Definitions

We are building a product to support the construction of computer software. To this end, a short description of this process is included in order to establish common terms.

The assumption is made that the process of software development has the following phases, applied with varying degrees of rigor and religion. It is realized that these descriptions are inadequate, and in many cases wrong. It is also realized that these phases are not encountered in strict order; development is often iterative.

1. Design. During this phase the problem is studied and decomposed. Various solutions might be outlined. The result of this phase might be documents, diagrams, formal specifications, ideas, or some combination of the above.

2. Initial Implementation. In this phase the design is transformed into code. This code may be executed in limited contexts, but not usually as part of a complete system.
3. Integration Implementation. During this phase code is being written and tested in the context of the desired system. The result of this phase is a system the programmers believe is the final product, meeting functionality, quality, and performance goals.
4. Acceptance Testing. During this phase the product is tested against some acceptance plan.
5. Maintenance. During this time bugs are fixed in the product, functionality put off is added, and new functionality is added.

Note that the distinction between these phases is often blurry, and perhaps non-existent.

One more term needs definition: programmer. In this document a programmer is a person who:

1. Specifies what code (as opposed to functionality) needs to be written.
2. Writes code.
3. Tests code.
4. Verifies that the code meets the design.
5. Documents the code.
6. Specifies the relationships between various units of code.
7. Constructs running examples of the system from the code. Perhaps constructs releases of the system from the code.

## 2.2 Sierra Coverage Of These Phases

It is anticipated that the first phase is covered by Rose. Sierra and Rose work together to cover the second phase. Phases 3 and 5 are addressed by Sierra. Phase 4 is assisted by Sierra to the extent that the methodology being used requires computation. In some instances, the combination of Rose and Sierra address #4.

## 2.3 What Is Sierra?

Sierra is a tool used by programmers. The coverages discussed above all fall into the realm of programmer support. It is our goal to make the process of programming as easy and error free as possible.

To accomplish this task, Sierra must:

1. Assist in the mapping of the design onto an implementation
2. Assist in mundane tasks of realizing the implementation. These are the physical entering of the program, the construction of the executable, enforcement of local programming standards, and the specification and maintenance of linkages between the various components.

3. Aid the interactions between programmers. The product should help ensure that records are kept, changes are made in a controlled fashion, and systems are constructed from combinations of components that make sense.
4. Provide tools that help verify the goals are being met. These include tools to help check correctness, performance, and mapping onto the design.
5. Be extremely easy to use, so that mere mortals can actually realize the above benefits.

The remainder of this document discusses how we might achieve this functionality.

# 3 Some Assumptions

We need to discuss some basic assumptions before charging off into the weeds.

## 3.1 Focus On Time To Market

Time to market is a major concern. We must do everything possible to ship a quality product as rapidly as possible. We must resist the temptation to embellish the product beyond what is required for a first release product being put into a (relatively) new marketplace.

The worst scenario is that we dally long enough for the competition to enter into their second release phase. We will then be behind the market instead of equal to or ahead.

## 3.2 Focus On User Interfaces

The user interface provided by Sierra is extremely important. It contributes to ease of use, initial sales, and continuing acceptance. We must make it easy for the novice to approach and use our product. We must also provide the facilities that an expert would want to improve productivity.

## 3.3 C++ Declarations And Definitions

C++ introduces two notions that are important to the Sierra product. These are:

1. A Declaration. A declaration is used to make a symbol known to the compiler. In can specify other information, such as function parameter profiles and return types. These are often forward references to allow the construction of 'pointer to mumble' types.
2. A Definition. This uniquely defines the object. For example, a function is defined when the code for the body is defined. A class is defined when the member information is supplied.

Where declarations are allowed there can be many. It is perfectly legal to declare the same function in many '.h' files, or to declare the same class several times. However, there can be only one definition for a declaration.

With one exception, there can be at most one declaration for any definition in any given view. The exception is 'main'; we allow more than one main program in a view. The practical effects of

this restriction is that we can always find the defining occurrence, and that there is no ambiguity regarding what object files are to be included when building an EXE.

### 3.4 Code Generation And Linking

The Unix 'make' facility will be usable as a mechanism for driving the compiler and the linker. This provides a level of Unix integration. However, since **`makefiles`** are extremely arcane, we will provide tools that hide their generation and maintenance.

## 4 Interaction Model

There are initially 4 major points of user interaction in the Sierra product: the program editor, **`makefile`** maintenance, the desktop, and/or the shell (command line interface). As part of developing the second release of Sierra, a more graphical interface will be added.

One proposal for the graphical interface is discussed in the companion document "A Proposal For A Graphical Front-End To Sierra". This topic is not further discussed in this document.

### 4.1 The Program Editor

The program editor must provide the following capabilities:

1. Motif Compliance.
2. Basic Text Editing. We must refine this over time to ensure we have a minimum level of functionality.
3. Support For Aliasing. If a user is editing something, and through some means requests that the object be displayed again, the image currently being edited must be displayed.
4. Support For Currency. If the underlying object is changed, the editor must show the new image.
5. Support For Decoration. The editor must be able to decorate the image to show errors, usage information, debugger information, and other feedback.
6. Support For Definition, Show_Usage, etc. The editor must provide for the various traversal commands, such as definition, show_usage, show_unused.

The editing subsystem, if not the editor, must also support the notion of lists of file names. These lists would be used for displaying the results of the moral equivalent of comp.make, and for the likes of show_usage.

For more information see the companion document "The Minimal Sierra Environment: User Interaction Paradigms".

## 4.2 Support For 'Make'

'**Make**' shall usable as a primary mechanism for sequencing the compiler to install units, generate object files, and control the linker to build an executable. Note that this requirement does not preclude the offering of an alternate mechanism if such a mechanism is better suited to the operations of the environment. Such an alternate mechanism is not further discussed in this section. This section describes the tools we provide to help the user control the use of '**make**'.

There are three distinct phases to the program building process. The first is generation of installed units. The second is code generation, or production of the '.o' file. The third is linking, or production of the EXE file. All of these operations are specified in **makefiles**. We provide a makefile maintenance facility to assist the user in generating and maintaining these **makefiles**.

The following operations are available to accomplish these tasks:

1. Given a view, construct the installation phase parts of the **makefile**. There needs to be some way to install a single unit, assuming its suppliers are installed. There also has to be a way to install, in the right order, all of the units in the view.
2. Given a view, construct the compilation phase parts of the **makefile**. This is done by analyzing all of the source files in the view, extracting all of the dependency information. Again, there has to be a way to code one unit assuming the dependencies are met, and a way to code all units in the right order.
3. Given a main program, generate the linking portion of the **makefile**. Alternatively, the user can ask that this work be done for all of the main programs in the view.
4. The user can supply special **make** rules with an object. For example, the user can specify that a parser generator be run to create the source. Or whatever. These extra rules are associated with the object somehow, perhaps as files. We need to work out what this means to dependency analysis. First blush opinion is that these rules override any system generated rules.
5. An 'activity' can be associated with the main program. The activity conditions **make**'s interpretation of the makefile such that the correct '.o' files are linked in to the executable. The activity has no effect on the compilation phase.

## 4.3 The Desktop - Manipulating The File System

The following file systems operations shall be available:

1. Traversal between directories.
2. Traversal from object (directory or file) to the appropriate editor.
3. CM operations. All of the CM operations must be available from the desktop, either via menus, drag & drop, or both.
4. Invocation of the makefile maintenance utilities from within a view.
5. Execution of a program, with and without the debugger.

6. Drag & drop. For example, we should be able to check out and in objects by dropping them on the appropriate icons. An object is printed by dropping it on the printer icon. Similar for editing. A complete list of these operations needs to be built. Operations also have to be available from menus. Some might be available from the mouse.
7. Display 'checked out/in' state somehow - perhaps as a 'locked' object. This might be as simple as reporting the read/write status.
8. Be able to show the Rational state of an object. Can be done using pop-up windows if this is the only method available.
9. Provide menu operations for 'enclosing view' and 'enclosing subsystem'.

# 5 Other User Interaction Issues

## 5.1 Command Line Interfaces

Most operations are also available from a shell command line. These report errors in the most reasonable way available, given the medium. Some of the operations available are:

1. All CM subsystem, view, inporting, and source operations.
2. The CM linking and **`makefile`** maintenance operations.
3. Compilation.
4. Definition, allowing use of Rational naming (attribute, link pack, etc)

## 5.2 Customization

To the extent practical, we must insulate our users from the arcana of 'X resource files'. However, we must support such files, as they are the standard storage mechanism. Exactly how we do this remains to be determined.

## 5.3 Makefile Currency

Operations must be provides that allow a user of **`make`** to verify:

1. That the **`makefile`** exists.
2. That the dependencies in the **`makefile`** are correct.
3. That the list of objects to be linked together is correct.

An interesting question is "When do we do these checks?" If the cost is low enough, we should do them at the time **`make`** is invoked.

In the event we find a problem and we don't automatically fix it, the user must be given the option of building a new, correct, makefile.

# 6 Functionality Requirements

## 6.1 General Requirements

1. NLS Enabled. We support the NLS method used by the host platform. If the platform supports double-byte languages, we must as well.

   For Sierra this requirement means that we accept NLS comments in the programming language and that we can readily translate our error messages. Sierra will use the platform NLS support for displaying dates, times, and any other nationalizable output.

2. Runs on Sun SPARC and IBM RS/6000 at first release. Supports C++ 2.1, possibly with version 3 template support if time permits.

3. Provides a very graphical UI. Assuming a 1992 shipment, the system must support monochrome and color Suns (and grey scale when these arrive), and grey scale and color RS/6000s. Where color exists we should use it to our benefit.

4. Provides a complete shell based command line interface for those who disdain GUIs.

5. The system must be easy enough to use that a Unix-literate programmer can perceive value in the system after no more than 30 minutes of study, requiring that person to read no more than 15 pages of material.

6. Non-Unix literate programmers must be able to perceive value within 1 hour, requiring the reading of no more than 20 pages.

7. It is permissible that this system require a Unix literate person to install it. However, the installation shouldn't require a 'guru'. We must follow a consistent set' of the emerging standards for installation, including configuration scripts, use of environment variables, etc.

8. We must support multiple users. It is permissible, but quite undesirable, to support only one user in any one view. As part of being multi-user, the system must participate in the voluntary locking schemes provides by Unix, and heed the locks.

9. Tools must be provided to put the system back into a consistent state. It is permissible that these tools operate by reducing everything to ground state, then rebuilding. The requirements are that the tools be complete, that they produce a consistent result, that they be operable by anyone, and that they not fail in the face of inconsistencies.

10. The system must operate over NFS. It is permissible, but undesirable, that a subsystem be contained on one Unix volume. There can be no such requirement for collections of subsystems.

11. It would be nice to support load balancing across homogeneous processors, especially at 'code generation' time. It is permissible to require that the underlying platform provide appropriate tools, such as the 'on' command supplied by Sun, HP, and IBM. This facility must be automatic and reliable to be included in the product.

12. If the user aborts a tool in progress, the tool must leave the system in as consistent a state as possible. Practically, this means that we must keep the windows of vulnerability small, and preferably the window should exist while running in the Unix kernel.

13. Sierra must make use of whatever the platform supplies to shorten the ECD cycle. This could be incremental linking, shared libraries, optimization control, or whatever.
14. Short of Sun paying us lots of money, the first release of Sierra is Motif based. If at all possible, Sierra should use the platform Motif libraries. We should not statically bind our own Motif libraries into Sierra unless we can produce a reliable product no other way.
15. Sierra must support concurrent use licensing. In addition, it must be possible to license subsets of the product, such as the compilation system, the CM system, and the editing system, allowing us to subset the product.

## 6.2 Compilation

1. Syntax and semantics are checked Rational components. These checks must be conditioned by the language accepted by the code generator (platform compiler).
2. It must be possible to add design rule checks to the semantics phase. What about the parsing phase? To the extent reasonable, the compiler should protect itself from bad code in these checks.
3. Program building (installation, coding, linking) can be done using **`make`**. Rational provides tools to help this process.
4. Rational provides a 'CC' command that provides 'smart recompilation' to the make-based system.
5. 'Target Key' and multiple versions of a compiler, both Rational and platform, must be supported.
6. It must be possible to copy a view such that the result is in exactly the same state vis-à-vis compilation as the source. It is permissible to require usage of a Rational supplied operation to do the copy. This operation cannot invoke the compiler, and cannot take significantly more time than a vanilla 'cp' operation. If some other command is used, smart recompilation will detect that the view should have arrived compiled, and change state indications (if they exist?) appropriately.
7. The compiler should support whatever is required to achieve rapid turnaround during the ECD cycle. This is some combination of smart recompilation, incremental linking, shared libraries, or some other scheme.
8. If a user has write access to a view, that user can promote and demote any object in that view regardless of whether or not that user has write access to client or supplier units in other views.

## 6.3 Editor

1. The editor must provide for integration with the rest of Sierra. In particular, it must support definition, show usage, aliasing, and currency. This is more important than editing features.
2. Support for incremental parsing. Sierra need not use this feature in the first release, but the support must be there for Rational II.

3. Annotations of some sort for errors, show_usage, etc. The mark must be in line; the user can see these marks while browsing the source. The display of the data under the annotation can be displayed in a pop-up or a separate window. If a separate window, this should be a sash type window in the same frame as the object containing the annotations. The user can request to see the error message associated with the annotation, or to see the annotation associated with the error.
4. There must be support for syntactic and semantic completion. Sierra need not make use of this feature in the first release, but the support must exist for Rational II.
5. Support must exist for incremental pretty printing. Again, Sierra need not implement this feature in the first release, but the support must be there for Rational II.
6. The editor must provide for structural traversal. Sierra need not provide this in the first release, but the support must be there for Rational II.
7. Support for invocation of the compiler and the Make facility.
8. Definition, using the mouse. The answer must be exact. It isn't acceptable to run a static analyzer to get the result. This operation should complete in under 1 second with the defining occurrence displayed in a window.
9. Show_Usage, using the mouse or a menu. It is OK to use a pop up menu here to get the various flavors of Show_Usage. The answer must be exact, and as complete as possible in the face of objects being in various stages of compilation.
10. If an object is currently being displayed by an editor, and if for some reason the user requests that the object be displayed, a new window will not be created. The original window will be brought forward and used. If required, the cursor will be moved to the position requested by the user. We need to decide what to do about currency.

## 6.4 CM

1. Controlled objects. Must be able to control objects with/without saving source. There is some discussion of a declarative model for specifying which objects are controlled; exactly how this works needs to be specified.
2. Line Of Descent support. Same comment as above vis-à-vis a declarative model.
3. Target Key Support. We must be able to specify the compiler, both Rational and platform, that is to be used for a view. It is not necessary that we allow a view to contain objects written in multiple languages or to support objects compiled with multiple versions of any one compiler.
4. We must support multiple versions of any one compiler.
5. Support for imports. User can import any view, released or not. If an imported view is released, it is compatibility-checked against the rest of the (released) import set. Since compatibility will be checked by the Rational linker at EXE-build time anyway, this can be considered an acceleration of the detection of some errors. Circular imports are allowed.
6. Support for system architecture specification. The user can specify which subsystems are allowed to import each other.

7. Support for path and subpath operations, renamed appropriately.
8. Importing of frozen, read-only views is permitted. In particular, a user need not have write access to a view to import or stop importing it. A more general statement can be made: a user need not have write access to any other view to perform arbitrary actions on any given view. Of course, it may be that the user must have access to the given view.
9. Operations must exist that verify the correctness and consistency of any state information.
10. Subsystems must be movable. However, the repair of the clients can be manual, as long as operations are provided that make this easy.
11. CM must do what is required to support rapid turnaround during the ECD cycle. This might be building special libraries for incremental linking or doing something special to enable shared libraries.
12. The CM system must support construction of executables. The method chosen must support compatibility checking to the extent possible to ensure that the '.o' files being linked together make sense. In addition, the user must be able to link in '.o' files that Sierra otherwise knows nothing about, and reference libraries ('.a' files) that are not under Sierra control, on a main by main basis.

    Rational provides an 'LD' command that does complete compatibility and completeness checking of the '.o' files before starting the link. This operation can be considered part of smart recompilation in that is saves a lot of time chasing compatibility problems and 'undefined symbol' diagnostics. This check can be disabled on a per-invocation basis (only if it is really slow!)

## 6.5 Debugging

We must support the platform debugger. However if at all possible the debugger should:

1. Have a look and feel consistent with the rest of Sierra.
2. Have a look and feel consistent with the Rational II debugger.

## 6.6 Rose Integration

See the companion document "The Minimal Sierra Environment: User Interaction Paradigms" for a description of Rose integration into Sierra.

# 7 Note Pad For Random Thoughts

Can **make** do a 'make all' across subsystems?

What are we going to do about deleted controlled objects? Rename of controlled objects?

What is unit state? How is it displayed?

Can we control EXE files? What does it mean? What about binary files?

Should we implement the makefile stuff as a makefile per object? This way we can keep the update process simpler, invoke make with the individual file to do single unit promotion, invoke a make of makes to make the world. This scheme might let us solve the currency issue by regenerating the makefile every time. If the user supplies an object makefile, then we don't touch it.

What are the rules regarding cross subsystem obsolescence? Can we import working views? Under what circumstances? What does this mean to the compiler? To the linker? Do we have to do something special to support mutually referencing object libraries ('.a' files)?

# The Minimal Sierra Environment: Components Description

**Rich Reitman**
**Charles Haley**
**Version 2.1**

## 1 Overview

This document describes the key components and features of the next generation product from a bottom up perspective. It is biased toward limiting the scope of the product to minimize development time while preserving those characteristics of the environment that provide the most value. In particular, the benefits of permanent Sienna/Diana trees, as well as the ability to do recombinant testing with small turn-around time for small changes, are thought to be of fundamental value to Rational users. We wish to deliver this core functionality for the C++ language to beta customers in July of 1992.

This document is a companion to the document titled "The Minimal Sierra Environment: Product Description", and is limited in scope to the core features called out by that document. This document does not include any consideration of layered products. It is not to be considered a detailed design document; these will be provided by the various development groups.

One motivation for this document is to focus us on product areas that are of immediate and tangible benefit to a customer. If we can't easily explain to a prospect why they want some particular feature, or explain why delaying the release in order to get some functionality is of value to them, we will look hard at not doing the work. We want to focus on how to use existing tools, adding value where reasonable.

Another very important aspect of this project is providing support and technology for the Rat/II project. We believe the components we are building meet their needs, and at the same time doing so on a more agressive schedule. More detail work needs to be done, both on exact functionality and schedule, but we are on the right track.

## 2 COTS Product Inclusion

We are using COTS (Commercial Off The Shelf) products wherever reasonable. We are doing this to save development time (reducing time to market), increase use of standards by using products that follow the standards, increase the quality of our product, and increase the probability of being able to integrate with other tools by forcing us to do it.

Currently there are eight candidates for inclusion. These are:

1. A Desktop Manager. A DTM is a combination file manager, program manager, and process manager. We are probably going to use a product called X.Desktop3 by IXI. XDT3 is bundled with every RS/6000 and is available on Sun, HP, Intel, and other platforms. This product should cost us somewhere between $100 and $200 per seat on platforms where it isn't bundled (Sun, HP).
2. Source Control. We are currently using RCS from the Free Software Foundation for source control storage. This product is free.
3. Help. We will probably end up using Frame, although X.Desktop3 includes a hypertext help facility as well. [WHAT DOES FRAMEVIEWER COST? DESKHELP IS FREE.]
4. An OODBMS. We might consider using an OODBMS for the various databases we maintain. Some of the candidates are unit state, links, CM configurations, and dependencies. Currently we are not planning to use an OODBMS.
5. A License Server. We will use the Highland Software license server. The OO group has already negotiated a paid-up license for both Sun and IBM platforms. Per seat cost to us: free.
6. An Encapsulator. It is likely we will use an encapsulator product to integrate the platform debugger into our product. The same encapsulator might be used for the makefile maintenance applications. One candidate is X.Deskterm from IXI. Per seat cost should be in the $10 to $50 range.
7. A GUI Builder. This product will be used to construct our top-level windows, menus, dialog boxes, and the like. There is no run-time charge; cost per seat is $0.
8. Motif. We are using the Motif libraries in our product. If we ship binaries of our product and do not include any OSF files except as incorporated in the binaries, the per seat cost to us is $0.

It should be noted that there are other COTS products we (CT) are implicitly using. These are:

1. The platform C++ and C compilers. We in CT aren't building code generators. There is a cost to the customer for these components, in the $2000 range.
2. The platform Debugger. We might investigate integration with Saber and GNU as well. The debugger is usually included with the platform compiler.
3. Other platform tools, such as MAKE, the linker, and Unix itself. These are usually included with the workstation.

Summarizing the above, the current contemplated cost per seat of the COTS products is between $110 and $250, plus the cost of Frameviewer (for help) if we use this product. The cost to the customer is our product plus the platform C++ compilation system (approximately $2,000 list).

# 3 Per Seat Pricing

Since we have a requirement that we must support concurrent use licensing (per seat pricing with enforcement), we will use the Highland Software license server. We anticipate using licensing for the following components:

1. The compilation and OM systems. We do this to allow selling the smart recompilation system as a batch compiler.
2. CM and associated software. This allows us to sell our CM system as a separate component. CM requires that the compilation system be installed as well.
3. The user interface. This component gets you definition, show usage, traversals, etc. It requires the compilation system. It does not require the CM system.
4. The graphical interface. When we get around to shipping the graphical browser (2nd release), this will also be separately licensed. It requires all of the above to function.

Note that the above isn't an expression of policy. It merely gives us the flexibility to price our product in various ways.

It isn't clear what the technical options are for license acquire and release. For example, it might not be possible to implement the policy of "acquire the license at first use and keep it until logout." It would be helpful to determine what the desired policy is so we can investigate its implementation.

# 4 Object Management System

The object management system provides the foundation for the rest of the environment. The core of its features are integration with the underlying operating system (UNIX), organization of the permanent objects of the system (including mechanisms for common operations), special purpose files such as link packs and dependency database storage to aid in the implementation of other environment components, certain naming operations such as attribute naming, and national language support (NLS).

The kinds of tasks to be performed are the design and implementation of:

1. Naming. We would want to provide common mechanisms for item naming, subsystem and view rooted naming, and object-handle based resolution.
2. Traversal. Common mechanisms for traversing within and between views and subsystems.
3. Persistent C++ Classes. Mechanisms for creating, opening, and deleting permanent objects.
4. File Management. Operations that facilitate the creation, opening, closing, and deletion of files. These operations will follow the voluntary locking protocols supported by NFS, which gives us some level of concurrency control.

5. Common Databases. The unit-level DDB, some CM information, and unit-state DB will be maintained through interfaces supplied by the OMS.

# 5 CMVC

CM provided by the first release of Sierra will have the following characteristics:

1. A Delta-Like view model. This is a flat model - views live in subsystems.
2. View-to-View importing. Implemented in a similar fashion as the current shell-based subsystem tools.
3. Configurations. Similar to Delta to the extent supported by the underlying Unix source control system.
4. Lines Of Descent. Supported to the extent the underlying Unix source control system allows. In addition, adding a declarative-based specification for controlling which lines are in which view would be a plus.
5. Releases, both "configuration only" and full. Do we want to support something like a compressed TAR image of a release?
6. Activities. This is a mechanism for building executables that use views different than the ones imported.
7. Any functionality required to support rapid turn-around. Possibilities are the building of shared libraries, tables for incremental linking, and the like. See Execution, below. Note that the CM system is responsible for linking programs.
8. Control of the platform linker. What are we doing about the compatibility requirement? About minimal '.o' inclusion? About foreign '.o' and '.a' inclusion?
9. Integration with other source control systems, such as RCS, SCCS or SCLM. The first release will be integrated with RCS. We aren't yet sure if we will have the time to integrate with any other system in this time frame.

It is felt that most of this functionality could be achieved using the current shell-based tools and RCS. However, we will probably code at least some of the system to meet the concurrent use licensing requirement, and to ensure an appropriate level of robustness and performance.

We should note that differential libraries will not be provided in this release, and potentially not at all. They have been eliminated in order to simplify:

1. Access Control Management. If objects aren't linked, we won't get into situations where a shared object is subject to two different access control policies. Standard Unix cannot handle this situation.
2. Compilation Management. Many Unix tools use the file's modification time to control processing. It is common to use the `touch` command to change the modification time of the file in order to affect compilation. This is made complicated if files are linked together.

3. The product. Not doing this work saves time.

In fairness, we should point out the 'cons' of not doing differential libraries. Some are:

1. Space Consumption. Making copies requires more space.
2. Time To Copy. Making a release (or any new view) will require making a copy. This takes time.

# 6 Compilation

The Rational compilation system consists of the following basic components:

1. The Precompiler. This component is responsible for parsing, preprocessing in the 'cpp' sense, and constructing semanticized trees.
2. Change Analysis. This component, perhaps rightly called a subcomponent of #1 above, implements the declaration numbering policy, builds and checks import keys, and provides compatibility checking operations to the rest of the environment.
3. The Rational CC command that sequences the above and provides a command line interface.
4. Editor Support. This component supplies the parsing and tree support that the editor needs to do image/tree traversal, structural traversal, and eventually pretty printing.

Components 1 through 3 form the basis of the batch smart recompilation product. Component 4 is only used by the Rational user interface. Note that the compilation system doesn't provide any special support for linking; this support will be provided by the CM system.

The Rational smart recompilation product is capable of being driven by 'make.' In the first release this may be the only method available for driving the compiler. Using 'make' gets us the following benefits:

1. Integration with Make. Doing this allows the customer to use Make to build non-Rational items, specify that operations must be done before compile (such as run a parser-generator), and add other dependencies.
2. A Command-line interface for those who don't want to use the desktop.
3. Use of existing customer knowledge. Customers (mostly) know how to use Make.
4. Possible integration into existing customer tools.

The 'cons' include:

1. We must provide tools to build and maintain Make files.
2. Using Make puts us at the mercy of its obsolescence checking. In other words, time stamps become quite important.

To understand how this works we need to discuss how Unix C compilers operate today. The CC command is conceptually a shell script that invokes the various passes of the compiler in turn. It

starts with the preprocessor (CPP), saving the output into a temporary file. This output is then fed into the 1 or more passes of the compiler proper. The results of compilation are then passed to the linker.

We are inserting our own processes in front of CC, also replacing CPP. There are several parts of this process, discussed roughly below.

## 6.1 Change Analysis

The first task for the Rational CC command is to determine if the object really needs to be compiled. To do this it:

1. Checks the existing image and tree for consistency. If there are any consistency problems, the tree must be rebuilt. Exactly how this is to be done isn't yet determined.
2. Check the imports. If the current suppliers are compatible with the previous ones, then the tree is OK as is.
3. Check that the object file (a '.o' file) is consistent with the tree. If it is, we are done. If it isn't, we build a preprocessed image file (a '.i' file) and then invoke the platform compiler on this file to generate the '.o' file.

## 6.2 Consistency

One of the problems we have to solve is how to verify consistency between the image, the tree, and the object file. The following is a strawman for how we might do this.

First, note that we assume the source is an unadulterated Unix text file. As such, we can't store any keys, time stamps, or other identifying information in the text. Also note that the Unix time stamps are not trustworthy in the face of copies, archives, FTP, and the like. Given these issues, the following (mostly) checks consistency:

1. Store the text's modification date in the tree. When we want to check consistency, compare this date against the modification time of the text file. If they are the same, assume they are consistent.
2. If the date is different, compute a white-space independent checksum for the image, and compare this with one stored in the tree. If they match, assume the tree and the image are consistent. This computation might be done as part of preprocessing the image. Note that this process gets us a simple check if the only changes to the image were in comments or white space.
3. Whenever we change the tree, increment a serial number stored in the tree. Store this serial number in the object file whenever one is generated. This can perhaps be done by storing the value in the name list, inserting a static declaration in the program before it is compiled, or by stuffing the value in a data cell after compilation. Check object/tree consistency by comparing this number.

To ensure that copied units arrive compiled, we provide a copy operation that orders the copies in modification time order, oldest first. As trees are copied, we check to see if the image has been already copied. If so, and if the modification time stored in the tree was the same as what the image used to have, update the stamp in the tree to the new value. Note that this requires changing some bits in the tree as it passes through.

If, however, the 'wrong' copy is used, the penalty shouldn't be large. The methods outlined above should discover that the files are indeed consistent even of the time stamps indicate otherwise.

It is understood that these are probabilistic checks.

### 6.3 Target Compilation

If we find we must build a new object file, we go through the following kinds of steps.

1. Preprocess the unit. The step builds a text file that will be given to the compiler. During this step we might only pass through parts of the include files actually necessary to compile the source file.
2. Invoke the compiler by invoking the platform CC command. Assuming it is successful, do anything required to register the new '.o' with the OMS.
3. Fix up anything required for consistency checking.

## 7 Execution

We execute by running the output of the linker. Note that the process of linking is controlled by the CM system.

As noted in the introduction, we want to do what we can to reduce the turnaround time for testing. We might be able to use the platform's shared library features, incremental linking features, or both to achieve this. Exactly what we can do and how to do it requires investigation.

## 8 Debugging

We will use the platform's debugger. We intend to encapsulate this debugger (put a skin around it similar to what one can do with the HP Encapsulator), but at this point don't know if this is practical for resource reasons. The major reasons for attempting to encapsulate are:

1. Ease of use. If we can make the debugger easier to use, this is good.
2. Integration with our compilation system. Doing this allows the user to more easily use Definition, Show_Usage, our editor, and other environment features.
3. Provide a user interface similar to what the Ada Business Unit is providing, potentially simplifying the documentation process, and perhaps sharing code with the ABU.

There is nobody working on this problem at this time. This is a major hole.

# 9 Editors

The combination of the requirements that: 1) integration with the rest of the environment is more important than whizbang editor features, and 2) time to market must be as short as possible, lead us to the following strategy:

1. Base our editing capability around the standard Motif text widget.
2. Implement the appropriate call backs from the text widget to get change reports, definition requests, etc.
3. Extend the text widget using standard methods to get in-line decorations (underlining).
4. Use XWindows mechanisms for currency control, communication between editors, and aliasing.
5. Simplification of other editors. We expect to use GUI builder technology to put together the initial versions, then examine how we can use or modify the Rose technology to achieve the same purpose. The emphasis is on rapidly building the framework so we can flesh out the issues with architecture, ease of use, and technology insertion.
6. Do nothing special to integrate other editors, such as Emacs or VI.

We understand that we will have to implement some minimal level of editing functionality over and above that provided by the text widget. The text widget comes with:

1. Basic display of characters in a window. Scroll bars are supported.
2. Character insertion and deletion.
3. Cut and paste.
4. Optional word wrap.
5. Traversal using keyboard (arrow keys, home, end, etc.).
6. Some level of undo?
7. Tab expansion?

We will certainly have to add:

1. Search and replace, both global and local.
2. Initialization and finalization (read from file and write to file).
3. Menu operations to access Cut, Copy, and Paste.
4. Operations to 'explain' decorations, traverse between decorations, and remove decorations.
5. Definition.
6. Go to line #/column #.
7. Structural traversal, at least for the ABU.

We might choose to add:

1. Printing (fancy, plain, or both)
2. Automatic indenting.
3. Keyboard macros, potentially including menu operations, and potentially able to be saved permanently.
4. Last command redo.
5. Regular expressions.
6. Tab elimination.

# 10 Rose Integration

The companion document "The Minimal Sierra Environment: User Interaction Paradigms" describes the sorts of Rose integration we are trying to achieve.

# The Minimal Sierra Environment: User Interaction Paradigms

**Rich Reitman**

This document outlines the kinds of user interaction in the minimal next generation environment. By minimal I mean to focus on those capabilities that must be in the environment in order for it to be considered as a possible product. Market analysis may determine that additional kinds of user interaction are required. This document is primarily concerned with the C++ environment, although some Ada specific characteristics are mentioned.

All of the kinds of user interaction are X/Motif based. They employ a combination of mouse movement, key clicks, menus, and dialog boxes to provide the user with a modern interface to the Rational environment. To the maximum extent possible, the environment presents the user with a common look and feel.

## 1. Desktop Manager

The environment is integrated with a desktop manager (such as X.Desktop) that provides directory browsing. The operations available from the desktop manager menu is sufficient to perform most environment operations. In particular, compilation, source control, and importing operations are available from the menu.

In addition, the desktop provides a "drag and drop" interface that allows the user to manipulate environment objects directly. One method of using "drag and drop" is to drag a set of objects and drop them on a command (i.e. program). This causes the command to run with the set of objects as the command's parameters. Check_In, Check_Out, and Make are examples where this interface is quite useful. An alternative use of "drag and drop" is where the set of objects is dropped on another object, rather than a command. This causes an operation to be performed that is (hopefully) intuitively what the user would expect. A standard example of this sort of "drag and drop" is to drop a set of objects on a directory; this causes the objects to be copied (or moved) to the directory. We anticipate that this method of drag and drop will be used for environment operations where both a source and a target are needed. For example, to import library L1 into library L2, the user could drop L1 onto L2.

## 2. Language Editor

We collectively refer to the Rational language specific (e.g Ada, C++) text editors as the language editor. In addition to the usual text editing capabilities, this editor provides a level of integration with the environment that is not possible with standard text editors such as emacs and vi. The language editor menus provide simple ways to perform environment operations that pertain to a single program unit, such as Check_In, Check_Out, and Compile. Various flavors of the Delta

definition operation are also available, preferably via the mouse. The language editor has the capbability of providing semantic completion and image formatting (including syntactic error correction); it is anticipated that the initial C++ editor will make limited use of these capabilities. In addition, the language editor is able to annotate the text (perhaps using underlining) to visually show characteristics such as error message, using occurrences, and unused declarations. For the minimal environment it is much more important that the editor be integrated with the environment than it be as complete as commercially available editors (e.g. emacs).

## 3. Rose Integration

The C++ environment is integrated with the Rose design tool so that users can use their designs to drive their implementation and can capture their implementation in designs. Rose module specifications can be used to specify the binding between class specifications and the files in the C++ environment that contain their concrete realization; in the absence of module specifications the environment has an alternative method to determine this binding.

Traversal to and from a C++ declaration and the corresponding declaration in Rose is provided as an extension of the Language Editor definition operation. Visiting a Rose class specification from a C++ declaration is done as a manner similar to definition. Visiting a C++ declaration from a Rose class has the same effect as having done a definition from a C++ using occurrence of that declaration.

It is also possible to generate Rose design diagrams from a set of C++ files (reverse engineering). The Rose class specifications for each of the class declarations in these files are built in such a way that subsequent traversals between Rose and C++ will work. It is highly desirable that a designated Rose class diagram be updated to contain the new classes generated; currently this capability has been given very low priority by the OO business unit.

Rose users can also automatically generate C++ files from their designs (spec/code generation). The generated files will capture all the relevant information in the Rose class specification in such a way that subsequent traversal between Rose and C++ will work. These features, along with the ability to perform traversals between designs and C++ code, provide the capabilities of a C++ class browser.

Although it is desirable that inconsistencies between design (Rose) and implementation (C++) be automatically detected and/or prohibited, this capability is not required in the minimal environment. Integration between Ada and Rose will provided in the minimal environment by the Ada business unit.

## 4. Show Usage

When show usage determines that there are multiple files that (may) contain a using occurrence of the specified declaration, the environment shows this list of files in a Motif window. The user is able to traverse from a file in the list to the Language Editor for that file. As a result of this traversal, the using occurrences are highlighted (e.g underlined) in the image. It is necessary for the

list of files to be (optionally) augmented with the images of the constructs in which the using occurrences was found, so that a complete presentation can be given in the Show Usage image. This presentation must be able to include line numbers, to that the user can use a non-Rational editor (e.g. emacs) to visit the using occurrences.

## 5. Compilation Error Summary

After performing a compilation in which multiple files failed to compile, the environment shows the list of files with errors to the user in a manner analogous to that of Show Usage. Traversal to each of the files, in a way that shows the errors, is provided. A more complete presentation of all the errors in the Compilation Error Summary image, along with traversal to the point of a particular error (in the Language Editor image), is possible. As was true for Show Usage, there should be a form of display that shows the line numbers of each of the errors, so that the summary can be used in conjunction with a non-Rational editor.

## 6. Debugger Editors

The Rational Ada debugger editor is a presentation of the Rational Ada debugger, which is only useful for program compiled with the Rational Ada compiler. The C++ debugger editor is an encapsulation of the platform standard C++ debugger. These debugger editors should have a similar look and feel and be Motif-based. As much as possible, the appropriate Language Editor should be used to display source, identify source locations, and identify program objects.

## 7. CMVC Editors

There are a variety of cmvc interactions that could be facilitated by having application tailored user interfaces, such as graphics and/or dialog boxes. Examples of such interactions are importing, change histories, and source differences. Although these editors are desirable, they are not considered necessary for the minimal environment.

## 8. Environment Customization

The user needs to be able to easily customize the environment. This customization includes setting environment "switches" (X resources?), configuring editor menu bars and menus, and customizing actions associated with menu items and "drag and drop". A uniform, direct manipulation method of performing this customization would be highly desirable, but is not considered a essential in the minimal environment.

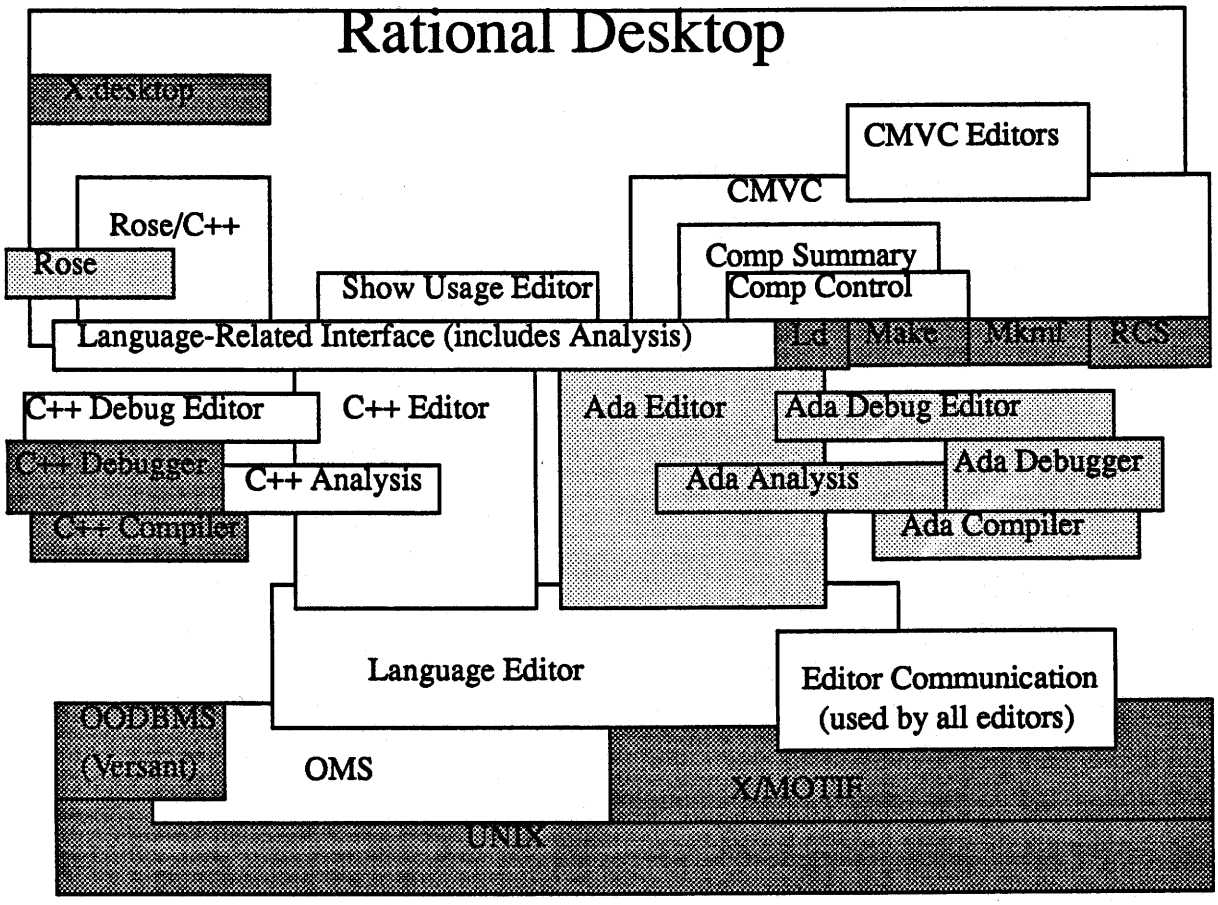
Another form of customization is reqquired to select the version of the compiler-related tools to be used as well as the design rule checking to be employed. It is envisioned that when design rule checking is first employed, the user will be able to turn on or off a fixed set of Rational defined checks; eventually more flexiible design rule customization would be desirable. The interface to

this capability is likely to be dialog-box based. Note that none of these capabilities are expected to be in the minimal environment.

# Architecture Diagram

**Rich Reitman**

This document is an illustration of the key implementation components of the next generation environment. It includes existing components, whether from commerical or internal sources, as well as new components that are to be built by Rational.




externally available component

internally available component (outside C++ business unit)

C++ business unit component

# The Minimal Sierra Environment: Description Of Major Milestones

**Charles Haley**
**Version 1.1**

## 1 Introduction

This document describes the three major milestones leading to first customer ship of the Sierra product. These milestones are called Demo, Alpha, and Beta.

Each of these three milestones is subdivided into the components that each of the groups are expected to supply, and the level of functionality in these components.

## 2 Demo (Target 1/15/92)

The purpose of this milestone is to verify the basic concepts, functionality, usefulness, and fitness for market. As such, we concentrate on trying to get a complete implementation for some small, subset, product.

It is expected that using the demo system, one can:

1. Create two subsystems.
2. Create a working view in each subsystem.
3. A working view in one subsystem imports a working view in the other.
4. C++ body and header files are created in each of the working views. One of the bodies in a working view depends on a body in the other working view. These bodies and headers are created with the Rational editor. These bodies must be non-trivial, but need not contain overly complex C++ code. There must be at least 2 header files and 2 bodies in each view. One view contains a `main` program
5. The above bodies can be individually compiled from the Rational editor. Errors are indicated via decoration. The user can traverse between the decorations, viewing the errors. It is permissible to allow only the decoration -> error transition in the demo system.
6. The complete collection of bodies can be compiled and linked using the CM system.
7. The result program is executed.
8. One can perform definition within the files. In particular, one can move from declaration to definition and use to definition. It is OK if this only works on an interesting subset.

9. The Rational editor is used to perform trivial editing.
10. Objects can be checked out and in.
11. Rose Integration, consisting of being able to traverse between Rose diagrams and C++ source, both directions. Support from the OO group is required to meet this objective.
12. The command line interfaces are not required.

## 2.1 Editor

The editor group supplies:

1. The Rational editor, capable of creating and editing files. The user interface must be similar to what we expect to ship.
2. The editor is capable of underlining, traversing between underlines, and showing the messages under the underlines.
3. Definition is supported
4. The editor can invoke the compilation system.
5. Show_Usage and Show_Unused aren't required at this time.
6. Rose Integration (in conjunction with the OO group).
7. Aliasing and currency support are not necessary.

## 2.2 Compilation

1. The compilation system can build semanticized trees for non-trivial programs, including precompiled headers. It is OK if the programs aren't competely semanticized, as long as an interesting subset is.
2. Interfaces to the editor exist to support definition.
3. Change analysis, smart recompilation, and consistency checking need not be included at this time.
4. Compilation after importing must be supported, even if only at the level of 'recompile everything'.

## 2.3 OMS

1. Link pack support, both inter- and intra-subsystem.
2. Object locking.
3. CM support as required.
4. Naming as required

## 2.4 CM

1. Construction of subsystems and working views. The user interface must be similar to what we expect to ship. The implementation can be the shell scripts if this is appropriate.
2. Controlled objects with saving source. Check out and in. Multiple LOD are not required at this time.
3. Importing. It isn't required to support circular importing at this time.
4. Support for platform linking with '.o' files coming from multiple subsystems. It isn't required to support minimum '.o' inclusion or compatibility.

# 3 Alpha (Target 4/15/92)

The alpha system is expected to be functionally complete, modulo bugs. As such, one can:

1. Create many subsystems, with many working views in the subsystems.
2. Create releases from working views, both foliated and configuration only.
3. Create lines of descent within subsystems/views.
4. Perform all allowed importing, including circular relationships where allowed.
5. Multi-user within a view. Editor, CM, and compilation support locking.
6. Edit, compile, and debug the entire C++ 2.1 language.
7. Definition and show_usage in all cases.
8. All editor operations, such as structural traversal.
9. Necessary editing features, such as search & replace.
10. Smart recompilation in the face of imports, editing, and copied views and subsystems.
11. All interfaces are in place, including command line interfaces.

There is little point in outlining the development group components, since these are described in other documents. There are other groups, however, that come into play between demo and alpha.

## 3.1 Documentation

1. A documentation plan showing what documentation is required, when.
2. Preliminary outlines of the CM and Editing user's guides (or whatever we want to call them).
3. Personnel forecasts. It is likely that we will need these forecasts significantly in advance of alpha, and perhaps as early as Demo.

## 3.2 Product Test And Release

1. A test plan detailing what kinds of testing we will do between alpha and beta.

2. A release strategy, detailing what will get released, when. Also details what the components of the product are.
3. A plan detailing how the product will be installed on a customer machine, and what development will be required to achieve this plan.
4. A preliminary support plan, detailing how the product will be supported once it gets to the field.

### 3.3 Marketing

1. Preliminary beta test plans.
2. Preliminary field rollout plans, detailing what channels of distribution we intend to use, how we feed product into those channels, how the channels are supported, and what material and programs we need to support these channels.
3. A preliminary announcement plan. The plan should include where we intend to announce, follow-up trade shows, seminars, or whatever we intend to use to get the word out about Sierra. This plan should also include pricing strategies.
4. A preliminary marketing materials plan, including any applicable of collateral materials, sales training, distribution support, advertising, public relations, competitive analysis.
5. A preliminary administration plan detailing how we intend to take orders, fulfill the orders, manage licenses, etc.
6. A 'partners' plan. We should have detail on which other companies are our partners (or we want to be our partners). The plan should include details on the cooperative arrangements.

## 4 Beta (Target 7/15/92)

At this point we believe:

1. Product development is complete.
2. Development required to install the product is complete.
3. Documentation to support beta is complete at a preliminary level (not at production).
4. A beta test plan is complete.
5. All of the preliminary marketing plans are complete, and some well under way to implementation.
6. A support infrastructure sufficient to support beta exists.

## 5 G/A (Target 10/15/92)

# A Proposal For A Graphical Front-End To Sierra

**Charles Haley**
**Version 1.1**

## 1 Introduction

This document describes a user model I would like to see. It is a proposal; the ideas are presented for purpose of discussion.

User interactions with the system are divided into 4 rough categories. These categories describe how the user manipulates source files, executables, structural relationships, and the file system. Each of these categories has what is called a 'controlling interface' through which the user interacts to get the job done.

### 1.1 Manipulating Source - The Editor

The text editor is the primary interface for entering and modifying the text that makes up a program. In addition, it is the primary interface for browsing between units in the program, and between these units and any pictorial representations of the program that might exist (Rose Integration).

In the editor a user can:

1. Create and edit source files.
2. Browse between sources (definition & usage)
3. Browse to and from design (to Rose picture)
4. Construct Rose diagrams from source (reverse engineering).
5. Construct source from Rose diagrams (source generation). [Is this a Sierra feature or a Rose feature?]
6. Compile source. Errors are shown in a second (sashed) window with marks in the source. One can put the cursor on the error message and go to the point of error, or put the cursor in the source and go to some error, and the error window will adjust to display the appropriate message.

   Remember that we use **`make`** to sequence the compiler. The process of constructing a **`makefile`** is discussed in section 1.3.4.

   Note that this window where errors are displayed can be used for more general purposes, such as explaining the meaning behind marks, prompting, or snapshot messages.

7. Build an executable (EXE). This option is only available if the user is displaying a main program (Is this true? A good idea?).
8. Invoke certain CM operations, such as check in and check out.
9. Definition and Usage that are ambiguous at the object level cause a window to open, displaying a list of the candidate objects.

## 1.2 Manipulating Executables - The Debugger

We need to figure out what this is. Are we simply supporting the platform debugger? Are we trying to encapsulate it? Does it interact with our editor? With activities? What is the effect of changing the source of a program while it is being debugged? Are there any multi-lingual issues that must be faced?

## 1.3 Manipulating Structural Relationships

This section describes editors that are used to look at and manipulate subsystem, view, and object relationships. The relationships discussed are:

1. Subsystem ↔ Subsystem. There are two sorts of relations here. The first is a structural relation; one subsystem is allowed to import another. The second is a client/supplier relationship; some view in the subsystem actually imports some view in another subsystem. Note the assumption that we support specification of structural relations.
2. View ↔ View. This is a client/supplier relationship, showing imports.
3. Object ↔ Object. This graph displays the object client/supplier relationships (#include/with).
4. Subsystem ↔ Views/Objects. This is a 'contains' relationship. A subsystem contains some number of views and indirectly, some number of objects.

A display of one of these relationships is called a 'projection.' The discussion below describes these projections in more detail.

In all cases we can traverse to the directory display for some selected object. If the object isn't a directory, we traverse to the editor for that object, or optionally the nearest enclosing directory.

It should be noted that the Rose subsystem and module diagram features may be used for these projections. If for some reason we don't use this code, we should at least use the visuals (icons, line drawing styles, etc.).

### 1.3.1 Subsystem And View Relationships

We display the subsystem relationships as a network of boxes. The following information and operations are available using these boxes:

1. Show the subsystem interconnects, derived from the import relationships of the views in the subsystem. These are shown with a particular line style. The supplier -> client display is left to right. [What do we do about circular relations?]
2. The permitted, or structural, connections are displayed as another line style. We might want to encode the combination of the permitted and import line styles into a third to reduce clutter.
3. A subsystem box can be expanded to show the views in that subsystem. In this case the lines in and out of the subsystem terminate at labeled marker points on the subsystem box. Lines are drawn from these marker boxes to the views, indicating the existence of a client or supplier relation. These lines are labeled with the name of the supplier or client view. This may be a permanent label or a popup, depending on screen clutter and readability.
4. The user can create client (import) relationships by drawing or moving a line to one of the marker points. A popup menu appears, displaying the names of the views that can be imported from the subsystem associated with the marker; selecting a view creates the import relation. A practical result of this rule is that the user must first have a structural link between two subsystems, creating the marker point, before an import relationship can be defined. We cannot modify the supplier relationships.

   Note that we can also create an import relationship by drawing a line directly from the client view to the desired supplier view in some other subsystem if both are expanded. The structural link must exist.

   When editing imports a CHECK button appears somewhere. Pushing this button checks compatibility through the import tree.
5. Clicking on a marker or a subsystem -> subsystem line takes the user to that subsystem.
6. Clicking on a line from a view to a marker takes the user to the supplier or client view. If necessary, the subsystem is expanded to show views, then the desired view is selected.
7. The user can construct other paths, subpaths, and releases. How this is done isn't worked out yet. We also need to work out how we specify the LOD relationships and any new imports that might be required.
8. We support zoom and pan.

### 1.3.2 Object Relationships

This projection displays relations between objects in any one view. We use the same diagram style as the subsystem diagram above: an outer box indicates the subsystem and view, and there are labeled marker points on the box to indicate transitions (in and?) out of the view. The labels give both the subsystem and view name in some way (using pop-ups? Using point and arc labels?).

In this projection the user can:

1. Display the basic linkages between objects. Some iconic form of the object is displayed with lines showing the client and supplier relations.
2. We might also want to display a 'completes' relationship somehow. This relationship shows which '.C' files contain declarations for definitions in some '.h' file.
3. Specify a main program in the view. The objects used by that main are highlighted (or the rest are grayed). If the main program has an activity associated with it, the supplier's mark points are relabeled with the name of the view from the activity.
4. Construct activities, A main program is first selected, and the system displays the graph described above. The user then asks that an activity be created. The user can edit the activity by using a pop-up menu on the marker or by drawing a line directly to the other view.

The restriction that this projection operate in one view is arbitrary, intended to simplify the problem. It may not be tenable to include the restriction.

### 1.3.3 Subsystem ↔ View/Object Relationships

This is really a CM projection. Using this view one can look at the relationships between the objects in views. This view applies to a subsystem. It displays as a diagram in two parts, displayed in separate windows (or perhaps panes). One window shows the views in the subsystem. The second window shows the objects in the subsystem. Objects are shown as stacks of whatever icon is being used; a stack indicates more than one line of descent (LOD).

Given this diagram, the user can:

1. Select a view. The objects in that view are highlighted.
2. Select more than one view. Objects in both views are highlighted. If these objects don't share an LOD, more than one item in the object stack is highlighted.
3. If two views are selected, the user can select an object that shares an LOD in the two views. This produces a difference listing/editor.

We probably want some way to filter the above projections to show only objects that share an LOD, objects that don't share an LOD, views that reference the same generation and those that don't, etc.

### 1.3.4 EXE Relationships (Makefile Construction)

Recall that '**`make`**' is used as the primary mechanism for sequencing the compiler to install units, generate object files, and control the linker to build an executable. This section describes the tools we provide to help the user control this process.

There are three distinct phases to the program building process. The first is generation of installed units. The second is code generation, or production of the '.o' file. The third is linking, or production of the EXE file. All of these operations are specified in **`makefiles`**. We provide a program builder facility to assist the user in generating and maintaining these **`makefiles`**.

The following operations are available from the program builder:

1. Given a view, construct the installation phase parts of the **`makefile`**. There needs to be some way to install a single unit, assuming its suppliers are installed. There also has to be a way to install, in the right order, all of the units in the view.
2. Given a view, construct the compilation phase parts of the **`makefile`**. This is done by analyzing all of the source files in the view, extracting all of the dependency information. Again, there has to be a way to code one unit assuming the dependencies are met, and a way to code all units in the right order.
3. Given a main program, generate the linking portion of the **`makefile`**. Alternatively, the user can ask that this work be done for all of the main programs in the view.
4. The user can supply special **`make`** rules with an object. For example, the user can specify that a parser generator be run to create the source. Or whatever. These extra rules are associated with the object somehow, perhaps as files. We need to work out what this means to dependency analysis. First blush opinion is that these rules override any system generated rules.
5. An 'activity' can be associated with the main program. The activity conditions **`make`**'s interpretation of the makefile such that the correct '.o' files are linked in to the executable. The activity has no effect on the compilation phase. Activities are built in the object relationship editor, see section 1.3.2 above.