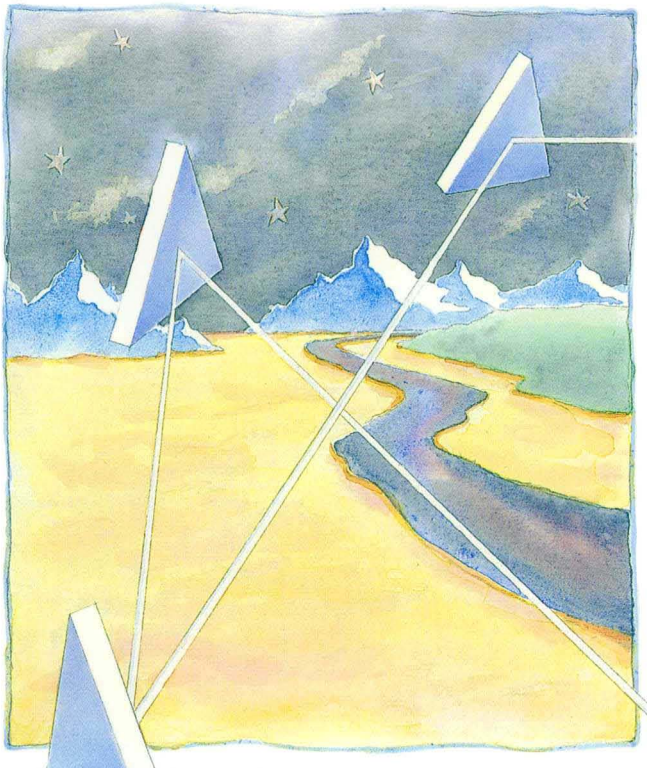


Engineering in the DSEE Environment

008790-A00

apollo

Engineering in the DSEE Environment

Order No. 008790-A00

Apollo Computer Inc.
330 Billerica Road
Chelmsford, MA 01824

First Printing: July 1986
Latest Printing: July 1988

This document was produced using the Interleaf Technical Publishing Software (TPS) and the InterCAP Illustrator I Technical Illustrating System, a product of InterCAP Graphics Systems Corporation. Interleaf and TPS are trademarks of Interleaf, Inc.

Preface

Engineering in the DSEE Environment is written for experienced users of the Domain® Software Engineering Environment (DSEE™) product. It presents an engineering perspective on the DSEE facilities and describes three engineering projects maintained by the DSEE environment. By reading this book, you will increase your understanding of how DSEE facilities work and are implemented. You will see how other engineers have used DSEE facilities to solve some of their engineering problems. With your enhanced knowledge of DSEE facilities, you'll also be able to extrapolate DSEE solutions for your own engineering problems.

The Organization of This Manual

We've organized the information in this manual as follows:

Chapter 1	Describes the five DSEE managers, their functional components, their uses, and some details of their implementation.
Chapter 2	Presents a case study of an engineering project converting to a DSEE environment.

Chapter 3	Presents a case study of an engineering group using DSEE facilities to manage the development of a multi-targeted operating system.
Chapter 4	Presents a case study of an engineering group using DSEE facilities to manage simultaneous development and maintenance of software.
Appendix A	Presents an abbreviated version of the system model used by the CAD tools project (described in Chapter 2).
Appendix B	Presents an abbreviated version of the system model used by the OS project (described in Chapter 3).
Appendix C	Presents an abbreviated version of the system models used by the DSEE project (described in Chapter 4).

How to Use This Manual

Reading Chapter 1 will enhance your understanding of the DSEE software and its functional components. You may appreciate the insights into the product's implementation that this chapter provides.

Each one of the case studies in Chapters 2 through 4 is designed to stand alone. You can either read all three or you can read only the ones that pertain to your own situation.

Each of these chapters contains "highlight" sections that examine topics of general interest to DSEE users. These sections, which are boxed and shaded, may provide you with quick answers to general questions. A separate table of contents lists highlight sections.

As you read each case study, refer to the related appendix. There, you will find a scaled-down version of the project's system model. The sample models in the appendixes will not only help you understand the systems, they will also aid you in writing your own system models.

Related Documents

For an introduction to DSEE facilities, read *Getting Started with the DSEE Environment* (008788).

For detailed information on DSEE commands, the system model and configuration thread languages, DSEE administration, and definitions of DSEE terms, refer to the *Domain Software Engineering Environment (DSEE) Command Reference* (003016).

For detailed information on the DSEE callable interface, refer to the *Domain Software Engineering Environment (DSEE) Call Reference* (010264).

For detailed information about the Domain system, consult one of the following manuals:

- *Getting Started with Domain/OS* (002348) describes the basics of the Domain system.
- *Using Your Aegis Environment* (0011021) is a detailed guide to using the Aegis environment.
- *Using Your BSD Environment* (0011020) is a detailed guide to using the BSD environment.
- *Using Your SysV Environment* (0011022) is a detailed guide to using the SysV environment.
- *Aegis Command Reference* (002547) provides detailed information about the Aegis™ shell commands.
- *BSD Command Reference* (005800) describes the BSD shell commands supported by Domain/OS.
- *SysV Command Reference* (005798) describes the SysV shell commands supported by Domain/OS.
- *Domain Display Manager Command Reference* (011418) describes the Domain Display Manager.

Problems, Questions, and Suggestions

We appreciate comments from the people who use our system. In order to make it easy for you to communicate with us, we provide the Apollo Product Reporting (APR) system for hardware and software-related comments, and the Reader's Response form for documentation comments. By using these formal channels you make it easy for us to respond to your comments.

You can get more information about how to submit an APR by consulting the appropriate Command Reference manual for your environment (Aegis, BSD, or SysV). Refer to the **mkapr** (make apollo product report) shell command description. You can view the same description online by typing:

$ man mkapr (in the SysV environment)

% man mkapr (in the BSD environment)

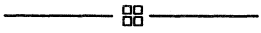
$ help mkapr (in the Aegis environment)

Alternatively, you may use the Reader's Response Form at the back of this manual to submit comments about the manual.

Documentation Conventions

Unless otherwise noted in the text, this manual uses the following symbolic conventions.

literal values	Bold words or characters in formats and command descriptions represent commands or keywords that you must use literally. Bold words in text indicate the first use of a new term, a command or keyword, or the name of a sample DSEE object or pathname.

user–supplied values	Italic words or characters in formats and command descriptions represent values that you must supply.
`command arguments, configuration thread text, and model text`	In examples, arguments to commands, configuration thread text, and system model text appear in this typeface.
user input	User input to a prompt is presented in color.
system model block types	Within the text the names of system model block types appear with initial capital letters (Aggregate, Element, External, and Model). The word "element" with an initial lowercase letter refers to a DSEE library element.
< >	Angle brackets enclose the name of a key on the keyboard.
CTRL/	The notation CTRL/ followed by the name of a key indicates a control character sequence. Hold down <CTRL> while you press the key.
. . .	Horizontal ellipsis points indicate that you can repeat the preceding item one or more times.
. . .	Vertical ellipsis points mean that irrelevant parts of a figure or example have been omitted.
	Change bars in the margin indicate technical changes from the last revision of this manual.
	This symbol indicates the end of a chapter.

Contents

Chapter 1 DSEE Concepts: an Engineering Perspective

Chapter 2 Case Study 1: Converting to a DSEE Environment

Chapter 3 Case Study 2: Developing a Multi–Targeted Operating System

Chapter 4 Case Study 3: Maintaining Released Software in a DSEE Environment

Highlights

Figures

Tables

Chapter 1

DSEE Concepts: an Engineering Perspective

This chapter presents the components and functions of the DSEE environment. Here we take an engineering perspective on the software, examining the components of the DSEE software, how you use them, and some details of their implementation.

Introduction

The Domain Software Engineering Environment (DSEE) manages large-scale development efforts involving engineers, technical writers, managers, and field support personnel. The DSEE environment provides source code control, configuration management, release control, task management, form management, and user-defined dependency tracking with automatic notification. All DSEE facilities are fully integrated with one another. You use the same interface to perform all DSEE software development functions.

Because the people that DSEE software supports and their data are typically spread among many locations, the DSEE facility must recognize and support distributed development environments. The underlying Domain architecture aids in the support of distributed development environments by providing network-wide virtual address space, transparent remote file access, and remote paging. DSEE software uses the following Domain facilities:

- D3M™, a distributed database management system, to store historical information
- Reliable, immutable files to store deltas and tasks
- Server processes to watch for asynchronous events
- **spm**, the server process manager, to distribute system builds to remote nodes
- A store-and-forward interprocess communication mechanism to safeguard against problems occurring when the network becomes temporarily partitioned

The dedicated window in which the DSEE software runs provides you with most of the tools you need for a complete programming environment.

What Is a Programming Environment?

The phrase **programming environment**, while used in many contexts, generally refers to an operating system environment and a collection of tools or subroutines. Different programming environments have different goals. Some manage the complexities of building releasable products. Others help engineers write, test, and debug their software. Yet other systems track the evolution of software.

Our concept of a programming environment is a facility that has all these tasks as its goal. The procedures of tracking software evolution, developing working software, and generating useful products are not isolated from one another. Most software engineers must perform all of these functions. In addition, they must coordinate all of their activities.

The DSEE facility is a collection of five fully integrated components of a total programming environment. Integrating these components allows the DSEE facility to automate many of the tasks involved with software engineering. This integration makes the whole DSEE product more valuable than the sum of its parts. DSEE software facilitates the peripheral tasks that can consume much of an engineer's time: tracking source evolution; structuring, building, and rebuilding complex software systems in development; composing releases of software for bug fixes, beta test distributions, and major upgrades; and making sure that each step that needs to be performed in the execution of a task is not forgotten.

The DSEE software separates these components from debuggers, editors, compilers, and test software. As a result, the DSEE environment is extremely flexible. You can use DSEE facilities to aid in the development of software in any language, for any target machine, written with any Domain–supported editor.

DSEE facilities are useful for everyone involved with large–scale engineering projects, not just engineers. Technical writers, engineering project managers, and the field support personnel involved with projects managed with the DSEE environment can all use DSEE facilities to assist them in their work. It aids them in their own tasks as well as automating parts of their interaction with the engineers involved with the project.

DSEE Managers

The five components of the DSEE facility, called managers, are listed below. Subsequent sections describe them in detail.

- The **history manager** controls source code and provides complete histories of versions
- The **configuration manager** builds systems and components, detecting the need to rebuild components and performing the builds only when necessary
- The **release manager** saves configurations and helps relate released software to the sources that built the configuration

- The **task manager** relates source code changes made throughout the network to particular high-level activities, holds general project related information, and provides templates for redoing common tasks

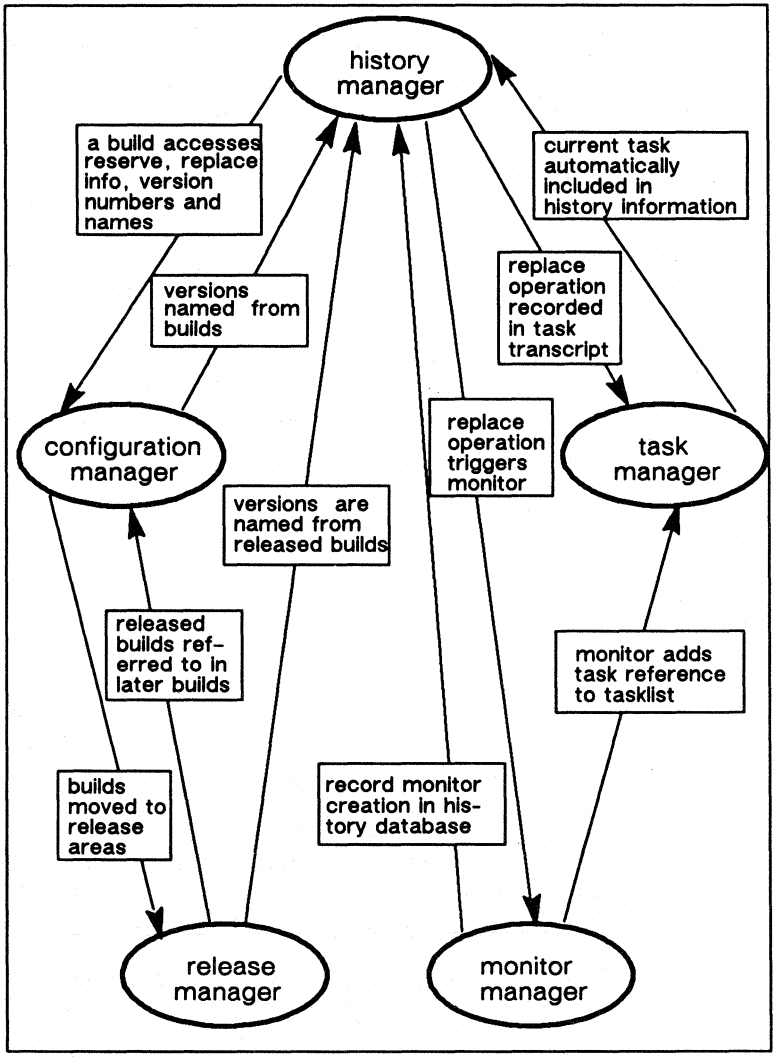


Figure 1-1. DSEE Managers and Their Interaction

- The **monitor manager** watches user-defined dependencies and alerts users when such dependencies are triggered

The Figure 1-1 illustrates the DSEE managers and their interaction.

History Manager

The history manager provides source code control within the DSEE environment. The history manager lets you store many versions of a program module and access them easily, while the history manager records all information on the module's evolution. The history manager is also the fundamental facility of the DSEE environment. All other DSEE managers interact with and depend on the history manager.

In this section we examine the history manager's components, use, and some aspects of its implementation.

Components

The DSEE history manager stores related components of a software development effort in a database known as a **library**. A library is a directory you create and manipulate with DSEE commands.

Each library contains **elements**. An element corresponds roughly to a file. An element may represent a program source module, a document that needs to be run through a formatting program, or any similar unit of text that is created and modified with an editor.

The Domain system uses a unique DSEE history manager file type to store elements. Unlike ordinary files, DSEE elements have successive **versions**. Each time you modify an element, the history manager makes a new version of the element, assigning it a new version number. Each version of an element has a unique version number. (Because the Domain system understands the structure of element files, you can read different versions of elements through the Display Manager and access them with any tool that uses the IOS streams facility, as we explain in "Implementation Details" later in this section.)

You can refer to a particular element version using its version number. The history manager displays the version number whenever it prints information about the version (for example, when you request information concerning an element's history).

Although the version numbers assigned by the history manager allow access to specific versions of elements, they aren't very effective at identifying particular versions for people. It's unlikely, for example, that you will always remember that version 2 of the element **sort.pas** was the version was used in Software Release 1.45 of a product. Even more unlikely is that you will remember all the different numbers of the constituent versions of all the elements used in Release 1.45: version 2 of **sort.pas**, version 5 of **sqrt.pas**, version 10 of **alph.pas**, and so on.

The history manager therefore allows you to assign a **version name** to any version of an element. Using our previous example, you could name all of Software Release 1.45's constituent element versions SR_1.45. This allows you to access the shipped version of every element used in the release with an easy–to–remember name. You can assign the name SR_1.45 to version 2 of **sort.pas**, version 5 of **sqrt.pas**, version 10 of **alph.pas**, and so on.

So far, we have simplified our discussion of element versions by assuming that each element's development is linear. This is frequently not the case. The history manager supports competing lines of development within an element. Each one is called a **line of descent**. If you view an element's development as a tree structure, the central path of development is called the **main line of descent**. Other paths, or alternate lines of descent, are called **branches**.

You might create a branch from an old main line version in order to fix a bug in that version without incorporating more recent changes to the element. Alternatively, at the start of an extended period of parallel development, you might create branches with the same name off the most recent versions of a set of elements. Finally, you might create a temporary branch in order to work on an element that someone else has reserved currently.

The history manager has a merge facility that you can use to incorporate modifications made on one line of descent into ongoing work on another line of descent. The history manager does much of the merging automatically, determining what to incorporate in the new version by checking changed sections of text against a common ancestor of the versions being merged. When the history manager cannot make an automatic decision in a merger, it asks you to tell it what to do. It then lets you edit the new text before going on with the merger process.

Using the History Manager

You **reserve** a line of descent of an element when you want to create a new version of the element. When you reserve a line of descent, the history manager gives you an editable copy of the line of descent's most recent version. The manager also ensures that no one else can create another version on that line of descent of the reserved element. If another user attempts to reserve the same line of descent, the history manager reports that this cannot be done and tells the other user that you have reserved this line of descent.

When you finish changing and testing the copy of the element, you **replace** the element's line of descent. The replacement operation creates the new version of the element.

Whenever you reserve or replace a line of descent, the history manager asks you to describe the changes you intend to make or have made. The history manager records this information, along with the date and time, the ID of the node you're using, and your user ID, in the history database associated with the library.

At any time you can determine which elements in a library are reserved, by whom they are reserved, and why they are reserved. You can also retrieve the history of one particular element. This history consists of the sequence of replace operations that created successive versions of the element as well as the operations that created alternate lines of descent. Using the history manager, you can also review the interleaved history for an entire library since a particular date (for example, the date of an earlier release).

Another use of the history manager is version comparison. This facility compares two versions of an element line by line and tells you exactly what differences exist between them.

Implementation Details

There are several aspects of the DSEE history manager's implementation that are noteworthy. In this section we examine the following aspects of the DSEE history manager:

- How historical information is stored
- How element versions are stored
- The integration of the history manager and the operating system
- The history manager's ability to handle system failure

Storing Historical Information

The history manager stores all of the data concerning a library, including element histories and other control information, in a database associated with the library. The database is controlled by the D3M database management system. This facility supplies the sophisticated data structuring facilities needed to represent the data concerning a library. Storing the information in a database also guarantees that the history manager can recover if system problems occur while the information is being updated, and ensures that many users can access the information concurrently.

Storing Element Versions

Because DSEE is designed to support large systems over a long period of time, and on moderately sized disks, it stores only the incremental differences (known as **deltas**) between successive versions of an element.

The use of deltas saves much space. Statistics on typical Pascal modules managed by the history manager showed that each new version makes the delta file about 1% to 2% larger. In other words, 50 to 100 versions of a module can be stored in the same amount of space as two plain text copies of that module.

In addition to its use of deltas, the history manager saves space by compressing leading blanks in source files to a single byte that contains a space count. Statistics on Pascal modules stored by the history manager showed that 20% of each module consists of leading blanks. The combination of storing deltas and compressing blank spaces leads to an interesting phenomenon: an element maintained by the history manager that has five to ten versions is often smaller than a single text copy of that element.

The DSEE history manager uses **interleaved deltas**. In other words, there is only one object containing all of the versions of the element. Intermixed control records allow the history manager to extract any version of the element in a single pass over the file.

History Manager Integration with the Operating System

The ability to construct any version in a single pass over the interleaved delta file (as discussed above) is critical to the history manager's management of DSEE objects. The file system offers ordinary, unmodified programs transparent access to any version of a DSEE element via the Domain system's IOS streams facility. Performance studies showed that a program can read any version of a typical DSEE element with less than 20% overhead relative to reading a plain text file.

Direct file system support is provided for the DSEE history manager. This enables the Display Manager, shells, and ordinary applications (like compilers and text formatters) to read any version of a source element directly from the library. No special step is needed to copy the version into a plain text file. You can access element versions from outside the DSEE environment in two ways:

- By specifying the particular version you want by line of descent and version number or by version name (called **extended version pathnames**).
- By using DSEE commands to establish a per–process **version map** that indicates that you want a particular version of the element. (The DSEE configuration manager, which we describe in the next section, relies upon this feature of the history manager to build systems.)

Element files, like all file system objects in the Domain system, are stamped with an object type unique identifier (a 64-bit type UID). There are several predefined object type UIDs, including ascii_file, object_file, bitmap, mailbox, and dsee_history_manager_file (called **casehm**). For each object type there exists a corresponding type manager implementing standard type operations on objects of that type of file (for example, open, close, get_record, put_record, seek, etc). From the operating system, you can perform only read operations on dsee_history_manager_file objects; updates to DSEE elements can be performed only with DSEE commands.

When the Domain input/output subsystem receives a request to open a stream on a system object, it allocates and initializes a file-descriptor and then alerts the appropriate type manager to complete the open operation. (The object's type UID determines the appropriate type manager.) The dsee_history_manager_file stream manager determines and records the desired version information in the file-descriptor. As subsequent calls are made to obtain records from the file, the DSEE type manager implements the appropriate behavior, which includes applying deltas and determining the next record in the desired version.

By default, the dsee_history_manager_file stream manager accesses the most recent version on the main line of descent of an element. However, you can use either of two procedures to access versions other than the most recent on the main line of decent. One is to use extended version pathnames which identify specific versions by line of descent and version number (for example, **//alpha/one/ps_lib/ele_1.c/bugfix/[3]**) or by version name (for example, **//alpha/one/ps_lib/ele_1.c/[sr5]**) in the pathname of the element file. Extended version pathnames are supported by the Domain input/output subsystem and thus can be accessed by programs running outside the DSEE environment.

The other way to access element versions other than the most recent on the main line of descent is to use DSEE commands to create a per-process version map that indicates that you want some alternate version of an element. Per-process version maps are inherited by all child processes. (We discuss version maps, which are created by the DSEE configuration manager, later in this chapter.)

Recovering after Failures

The history manager coordinates updates to the library database with updates to element files in such a way that a user-level operation appears to be executed in one complete unit: it either succeeds entirely or leaves no trace in the library.

In a distributed environment "partial" failures are more likely to occur than in a local environment. For example, the node on which your library is stored might fail while you are replacing an element. The DSEE software provides reliable recovery for partial failures. The database management system that the history manager employs uses journal files and semaphores to implement transactions.

Configuration Manager

Configuration management is an important aspect of a programming environment. However, different people have different ideas of what configuration management is. Before describing the DSEE configuration manager, we discuss the DSEE view of configuration management and how it differs from other views.

After defining DSEE configuration management, we present the configuration manager's components and their use, and some highlights of the configuration manager's implementation.

What Is DSEE Configuration Management?

In general, **software configuration management** is thought of as having three components:

- Source code modification control and tracking
- Product version identification
- System building

The DSEE history manager provides comprehensive source code control, and the release manager oversees product version identification. The DSEE configuration manager's primary concern is system building.

The DSEE concept of **system building** is the act of translating the appropriate versions of the constituent elements. The configuration manager relies on two things to build a system: a system model and a configuration thread. A **system model** is a sort of blueprint of the system's components, their interrelations, and their translation rules. A **configuration thread** specifies which versions of elements to use in the build.

This separation of version specification from component definition is one of the DSEE configuration manager's distinguishing characteristics. The configuration manager builds programs from desired element versions. When building a program with the configuration manager, you specify only the desired element versions and translator options; the blueprint of the constituent components needn't be redefined for every build. The configuration manager finds the results of previous builds that satisfy your specifications, executing the appropriate translation rules to derive the remaining objects.

The configuration manager associates a record of the versions used during the build with the **derived objects** (that is, the output of translation rules) it produced. The configuration manager thus assumes responsibility for managing derived objects. The configuration manager can manage any type of derived object (object code, listings, microcode, object code for other target machines, etc.).

The DSEE configuration manager is capable of building systems for multiple target machines. It can distinguish between derived object modules based on the versions of the sources that were used to build them and the translator options used.

The DSEE configuration manager makes it possible to:

- Avoid the common error of building an inconsistent program by failing to rebuild components that were indirectly affected by modifications to other elements
- Build different versions of a system without having to copy versions of source files and derived objects
- Work simultaneously on development and maintenance environments using common source libraries and a common pool of derived objects
- Switch easily between development and maintenance activities

- Establish concurrent and noninterfering multi-user debugging and testing environments
- Create a permanent record of the element versions used in a released configuration of a program
- Use this permanent record to establish a process environment for programs accessing DSEE elements
- Repair bugs in a prior release while continuing development activities
- Gain easy access to the source modules that make up a particular system configuration and to precise documentation about that configuration
- Build systems configured with any desired versions of elements managed by the history manager
- Use multiple nodes to build system components concurrently
- Manage software projects in which several source modules are used in multiple products by ensuring that all systems requiring shared elements are built with the appropriate versions of those elements

Components

We have already introduced two of the components of the DSEE configuration manager: system models and configuration threads. In this section we explain both in greater detail and discuss three other aspects of building a system: model threads, bound configuration threads, and binary pools.

System Model

As we explained above, a system model is a kind of blueprint for a system. In effect, the system model can be said to define the system. Thus, before you can build a system, you must define its system model.

Having a model of a system has several advantages. Describing, as it does, the whole structure of a system, it serves as a reference on the system for project members. It also frees individual members of a project team from needing to remember all the details about the structure of the project's software; they need only consult the model for details. Finally, use of a uniform methodology of system description (like that provided by system models) makes it easier for engineers working on one project to understand the construction of other projects' systems.

The system model gives a static description of the structure of the system in terms of its buildable components, their constituent elements, and their hierarchical relationships. The system model also gives the translation rules to be used to produce the derived objects, such as the object and listing files. Translation rules can include translation options, which you can decide to use or not for any given build of the system.

The system model is written in a block-structured system model language. The block structure mirrors the hierarchical relationship among buildable components and provides scoping for certain kinds of declarations. The system model describes each component's dependencies (that is, the elements, files, and other components it depends on; the source libraries it requires; and the translators it requires). The model describes the necessary translation rules (the compiler, binder, or formatter command lines DSEE must use to derive, or build, the components). The dependency relationships in the system model state the order in which DSEE must derive the system's low-level and intermediate-level components.

Because the system model serves as a blueprint of your system, you edit your model only when your system requires a structural change.

Figure 1-2 demonstrates the hierarchical block structure of a system model. The brackets on the left-hand side of the text outline the component blocks within the model. Note that some blocks are nested within one another. This nesting results in the model's hierarchy. (This illustration does not present a working system model. You can find several examples of system model source code in the appendixes of this manual.)

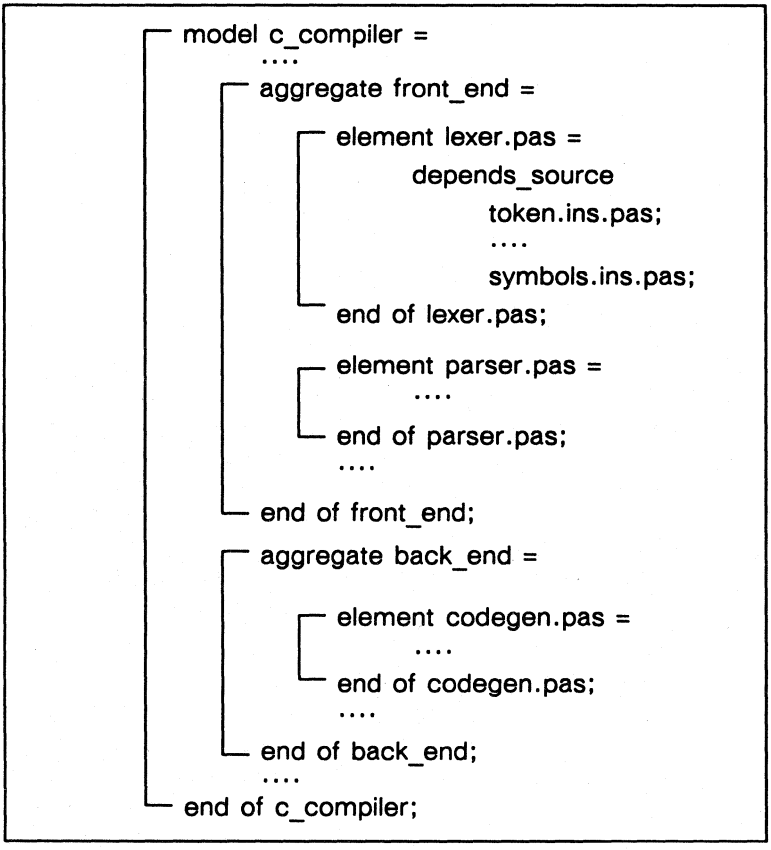


Figure 1–2. An Example of a System Model's Structure

The System Model Language

The system model language provides a syntax for describing the behavior of translators. The **translation rule** for a component is essentially a stylized script of shell commands that contains the compiler, binder, or formatter command lines needed to build the component. As a result, you can use any translator in a system model, even a translator not supplied with the Domain system, provided you use the proper system model syntax to express the translator's behavior.

The language has constructs for indicating where in a translation rule the configuration manager should substitute the pathnames of the relevant derived objects. Such embedded constructs are necessary for two reasons. First, only the configuration manager knows the pathnames of derived objects, and even then only at build time. Second, only you know where those pathnames should ultimately appear in any shell script derived from the translation rule. (Remember that the configuration manager has no special knowledge of the translators used to build a system.) The language provides similar constructs for indicating where translation options specified by the user at build time should be substituted. Aside from these embedded constructs, the configuration manager treats a translation rule as uninterpreted text.

Here is an example of a translation rule. It shows the construct that indicates where the configuration manager should substitute derived object pathnames (**%result**) and the construct for specifying translation options given at build time (**%option**). Another construct, **%source**, indicates where in the translation rule the configuration manager should substitute the name of the component being translated.

```
translate
    /com/pas %source %option(-dbs) -b %result
%done;
```

As we indicate above, a system model describes two aspects of the system in detail: how its components are translated, and how the components relate to one another. The structure of the system is indicated by the structure of the model, as described using the system model language. The system model language has syntax for describing each system component's **dependencies**: the source files and elements, the translation tools, and the other components on which the individual component depends. The configuration manager uses this structure of dependencies to determine what derived objects it needs to rederive at build time. A change to one of a component's dependencies between otherwise identical builds causes the configuration manager to rebuild the component and any other system components that depend on it. (We discuss the configuration manager's build process in more detail later in this section.)

The system model language allows declarations that are common to a set of buildable components to be factored out. For example, if every buildable subcomponent of component utilities depends on a certain element (perhaps an include file), the system model can state this succinctly with a declaration at the block level for utilities. Placing the declaration at the block level for utilities defines the scope of the declaration. The set of components to which a factored declaration applies may be further restricted by a regular expression for the component names. The regular expression is written just as it would be for use in any other Domain system operation or application, using the same wildcards. Therefore, it is a simple matter to specify a single translation rule to be used for every buildable component whose name ends with a certain extension (for example, **.pas** or **.c**).

The system model language has a conditional compilation facility that makes it possible to construct a system model that can be used to build structurally variant configurations of the same system. This feature is useful for such applications as systems that need to be configured differently for different target machines. Conditional compilation is controlled by switches that you supply when you set your current system model.

Other system model language constructs enable you to modularize your system model, breaking up the system model source code into a **root model**, which represents the main body of the system model, and **model fragments**, which are segments of system model source code incorporated into the model when it is compiled. System model modules can be maintained as DSEE elements and provided with the same degree of version control as system model components through constructs known as **model threads**. Because model threads share many characteristics with configuration threads, we will discuss model threads and model modularity after we've explained configuration threads.

Configuration Thread

The system model describes the structural aspects of the system. It does not specify which versions to use for its constituent elements during any given build. The configuration thread is a separate component that describes which versions of sources should be used to build the system. The configuration manager uses the configuration thread to bind a system model to a specific set of versions for your builds.

You write configuration threads using a high-level configuration thread language. You don't have to (and typically won't) identify the exact version that the configuration manager should use for each and every element in the system. Thread syntax can indicate, for example, that you want to build with the most recent version of the main line of descent of every element in the system, with the one line rule:

```
[]
```

Moreover, many configuration thread language constructs are unbound to precise version specifications. Suppose, for example, you have created a new version on the main line of descent of an element since you last used the configuration thread containing the line in the example above. The configuration manager recognizes that the new version is the most recent version of the element on the main line of descent and uses that in the build rather than the older version. You don't have to edit your thread for each build (although you probably change your thread more often than you do your system model). As a result of the high-level nature of the language, most configuration threads are only a few lines long.

A configuration thread consists of an ordered list of rules. Each rule contains a predicate that identifies the elements to which it applies and a specification of the version to use for those elements. The predicate may stipulate that the rule applies only to the element with a given name, or more generally to elements whose names match a given regular expression. The version specifier may stipulate that the rule applies only to elements that are currently reserved by the user from the history manager, or, alternatively, only to elements having a branch with a given name. The version specification may be static (for example, one version specification might tell the configuration manager "use version 42 of the element," or it might say "use the version named Release_9.0") or dynamic (for example, rules that tell the configuration manager "use my working copy of the reserved element" or "use the most recent version of the element on the main line of descent").

The order in which rules appear in the configuration thread is significant. To determine which version to use for a given element, the configuration manager selects the first rule whose predicate is satisfied by the element. Configuration thread rules are therefore usually listed in order of increasing generality of their predicates.

Here are a couple of figurative examples of configuration threads to illustrate the importance of order in configuration threads. Suppose you are a developer working on the next release of a system. You would typically want to use the most recent versions on the main lines of descent of all elements, except that you would want to use your working copies of any elements that you currently have reserved. Your configuration thread might state:

1. If I have the element reserved, then use my working copy.

2. Otherwise, use the most recent version of the element.

If you were working on a bug fix to a past release of the system, you would probably also want to use your working copies of any elements that you currently have reserved. However, instead of the most recent versions of other elements, you would want to use the same versions that went into the past release that you are fixing, unless you had created a branch off of one or more of the elements in order to fix the bug. In such a case you would want to use the branch versions. Your configuration thread might state:

1. If I have the element reserved, then use my working copy.

2. If the element has a branch named **Release_9.0_bugfixes**, then use the most recent version of that branch.

3. Otherwise, use the versions that were used in Release 9.0.

The last rule in the above example illustrates a useful capability of the configuration thread language: you can specify that you want the configuration manager to build your system using the same element versions used in an earlier release simply by referencing the release by name. (Releases are created and named with the DSEE release manager, discussed later in this chapter.)

To simplify the discussion, we have assumed that a configuration thread specifies, for each element used in the system, a single version to be used whenever the element is needed during a system build. In fact, configuration threads may specify that different versions of an element should be used in different contexts. For example, if buildable components **set_grid.pas** and **init_design.pas** both depend on the element **banner.ins.pas** (an include file), the configuration thread can specify that the configuration manager should use one version of **banner.ins.pas** when building **set_grid.pas** and a different version of **banner.ins.pas** when building **init_design.pas**. This facility makes it possible to use a bug fix version of an element in exactly those contexts that need the bug fix, leaving the remaining contexts to use the element version that doesn't contain the bug fix.

Figure 1–3 illustrates how two engineers can build entirely different versions of the same system through configuration threads. Engineer A is using a thread that tells the configuration manager:

1. For the element **color.pas**, use the most recent version on the main line of descent.

2. For every other element, use the version named **V2**.

Engineer B has element **bit.pas**'s main line of descent reserved in his working directory and is using a configuration thread that tells the configuration manager:

1. For each element used in the system, if I have one of its lines of descent reserved, use the version in my working directory.

2. Otherwise, use the version named **V2** of each element.

As Figure 1–3 shows, each engineer's builds use a different set of element versions.

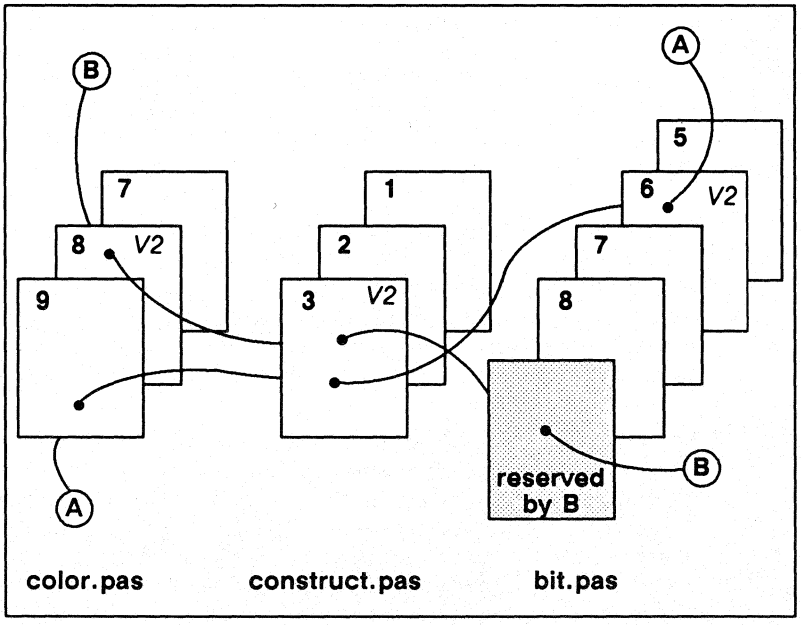


Figure 1–3. Pictorial Representation of Configuration Threads

We close this section by mentioning that configuration threads may also specify translation options for some or all buildable components. Thus, if you wanted to use a debugger option of a compiler when building derived objects with all reserved source modules, your configuration thread might include a rule that states "If the component depends on a reserved element, compile with the -debug option."

Model Thread

In the same manner that you can fragment parts of your source code into portions that are shared by several modules, you can fragment parts of your system model source and share the portions among different systems. These **system model fragments** are referred to in the **root system model** with **%include** directives. When the configuration manager compiles your system model, it incorporates the fragments in the compilation.

Extracting certain parts of a system model into included fragments not only lets you share parts of a system model among projects; it also makes it easier to develop and maintain a system model. Just as it's easier to debug and manage a program that is separated into modules, it's easier to debug and manage a modularized system model.

The DSEE environment gives you much the same control over model fragments as it gives you over software modules. All portions of a system model can (and should) be maintained as DSEE library elements. When you compile your system model, the configuration manager employs a model thread to identify the constituent versions of model fragments to use in the compilation.

Since a model thread serves the same function in relation to a system model that a configuration thread does in relation to a system, it's not surprising that model threads and configuration threads are much alike. The model thread language for version specification is nearly identical to the configuration thread language for version specification. In general, the languages differ in those characteristics unique to their respective uses.

In addition to specifying versions, model threads can also identify variables for conditional processing of the system model during compilation. These "target" variables (named for the model thread rules in which they appear) tell the system model which parts of the model to compile, and which to ignore, during a given compilation.

Bound Configuration Thread

Configuration threads rarely specify explicit versions for every element in the system. Instead, they contain dynamic references to versions like "the most recent version on the main line of descent." The configuration manager must evaluate the configuration thread at build time in order to determine, for each element in the system, which version to use. This "binding" of an element to a version at build time is stored in a **bound configuration thread**, or **BCT**. A BCT reflects the hierarchical nature of a system model, since your build may call for different versions of the same element in different contexts.

In addition to specifying explicit versions for all elements in the system, a BCT also specifies translation rules for each of its buildable components, including any translation options called for by the configuration thread. Model versioning information is part of the BCT, too; this and conditional compilation information (also included in the BCT) are derived from the model thread. A BCT is thus a complete specification for building a configuration of a system.

The BCT serves as a valuable record of what went into such a configuration. As we shall see later, the configuration manager examines existing BCTs to determine whether it needs to rebuild components during a build. Comparing the BCTs of two configurations tells you how built systems differ from one another. You can also refer to BCTs to see exactly which source versions and build options were used to create particular builds. This is helpful when you're debugging a system.

Binary Pools

The configuration manager places all the products of a system build (BCTs and derived objects) in **binary pools**. Each system has one or more binary pools associated with it.

An optional specification recognized by the system model language tells the configuration manager where you want it to store a component's derived objects. Each system has one default pool associated with it. However, you can, if you wish, define and use any number of other binary pools in addition to, or instead of, the default pool.

Different systems can share the same binary pools. For example, the system for a Pascal compiler might use two pools, one to hold the derived objects for its front-end components and another to hold the derived objects for its back-end components. The same back-end pool might also be used by the system for a C compiler, assuming that the two compilers have a common back-end.

In order to reuse derived objects produced by earlier builds, the configuration manager must be able to tell what versions and translation options were used to produce each derived object. One approach to solving this problem would be to store such version and option information in the derived objects themselves. This, however, would require special cooperation between the configuration manager and the translators, and would therefore effectively restrict use of the configuration manager to translators supplied with the Domain system. For this reason the configuration manager associates with each derived object a BCT. As we pointed out above, the BCT provides an exact specification of the versions and translation options that were used to produce the derived object.

At any given time the binary pool may contain several sets of BCTs and associated derived objects for the same buildable component. This occurs when developers build different configurations of the system, as our developers A and B did in Figure 1–3. Because A and B are building the same system, they share the system's pools. By sharing the pools, they build only those derived objects that have unique configurations (like A's use of **color.pas** and B's use of **bit.pas**) and share the derived objects that are identical for each configuration. (In our example, this would be the derived objects produced by translating element **construct.pas**.)

The configuration manager deletes derived objects from the pool as they fall into disuse, in accordance with a user–specified limit on the number of derived objects per component the pool can hold. If inserting a new derived object for a given component would exceed this per–component limit, the configuration manager first deletes another of the component's derived objects, using a least–recently–used replacement algorithm. Should the deleted derived object be needed again in the future, the configuration manager can rederive it from its constituent element versions, which are always available from the history manager.

In addition to identifying a limit on the number of derived objects per component that a pool can hold, you also specify a minimum age for deleted derived objects. This ensures that the configuration manager will not delete a derived object unless it has resided in the pool for more than a given period (for example, 24 hours).

Figure 1–4 illustrates the concept of a binary pool.

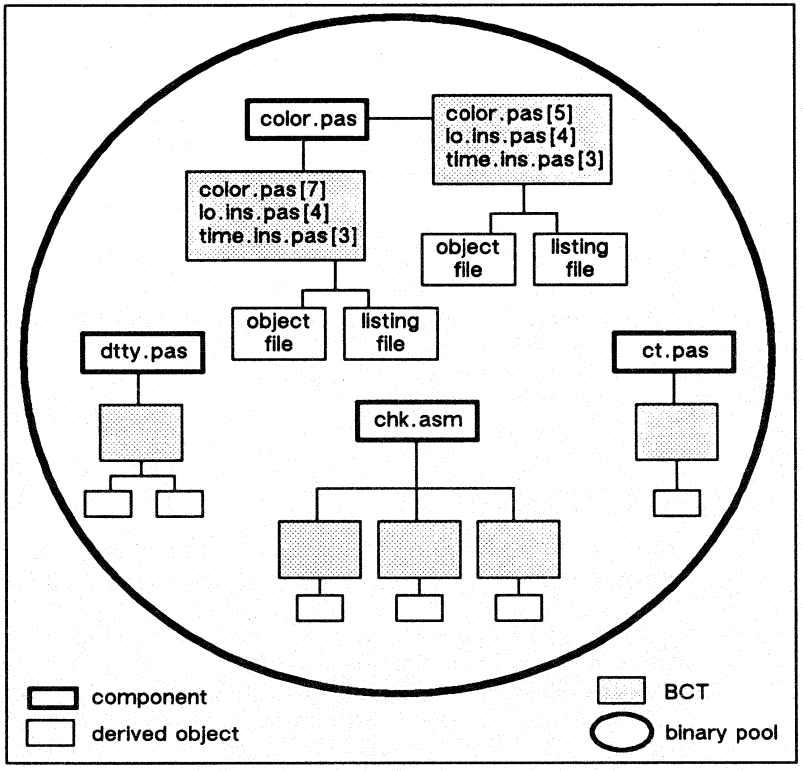


Figure 1–4. A Binary Pool

Derived objects built from working copies of reserved elements are typically not of interest to anyone but the user who has those elements reserved, so it makes little sense to place them in a central pool devoted to sharing. Moreover, such derived objects cease to be of interest even to the user who built them as soon as he or she edits the working copies and builds new derived objects. Therefore, such derived objects ought to be deleted more quickly than derived objects in the central pool would be. For these reasons the configuration manager provides each user with a **reserved pool**, into which it places all derived objects of components that depend on one or more reserved elements (and, in turn, the derived objects of any

components that depend on them). By default, a reserved pool has a limit of one on the number of derived objects it can hold that are produced by translating one component. Therefore, every time a new derived object is inserted into the reserved pool it replaces any existing derived object for the same component (if the age limit of the pool has been reached).

The configuration manager controls the derived objects in binary pools. If you want to access one or more derived objects, you can export copies of, or links to, the derived objects.

Using the Configuration Manager

In this section we discuss how the components of the configuration manager interact during the building process as well as several aspects of the building process that you can control for optimal performance.

The Build Process

Once you have selected a current system model and a current configuration thread, you can issue a command to build all or part of the system. When you do, the configuration manager first evaluates your configuration thread in order to determine which element versions and translation options to use. Any dynamic configuration thread references (like "use the most recent version on the main line of descent") are resolved to specific versions. The result is the **desired BCT**. The desired BCT specifies version and translation rules for the component being built, its subcomponents, their subcomponents, and so on.

The configuration manager checks the desired BCT against the BCTs in the system's binary pools to see whether there are any derived objects that it can reuse for your current build. The configuration manager reuses the derived objects whose BCTs match components of the desired BCT, then builds the remaining components in accordance with your desired BCT.

Notice that in the preceding discussion the configuration manager determines what needs to be built based on desired versions; no mention is made of elements having *changed*. The configuration manager does not build something because some element or working copy has changed, but only because the desired BCT calls for a derived object that does not currently exist in the pool. This behavior enables the configuration manager to support multiple concurrent configurations while sharing sources and derived objects whenever possible.

There are two common reasons that the configuration manager can't find an existing BCT to match all or part of the desired BCT. The first is that you have just created a version of some element for which your configuration thread requests the most recent version. The second is that you have edited your working copy of a reserved element called for in your configuration thread. However, the configuration manager does not distinguish these cases of changing the targets of dynamic references from any other case (for example, when you are trying to rebuild an old configuration whose original derived objects have dropped out of the pool due to old age).

Once the configuration manager determines that it needs to build a component it performs three steps. Figure 1–5 illustrates the process.

First, it forms an **actual translation rule**. This involves substituting into the translation rule template (declared in the system model) any pathnames of derived objects, such as those for subcomponents, pathnames of source elements being used and any translation options you request. The actual translation rule is a script that builds the component.

Second, the configuration manager creates a new process and establishes a process context in which the desired element versions are automatically retrieved whenever elements are read by programs executing in that process. This is accomplished by setting up a process–specific version map in accordance with the desired BCT. The configuration manager cooperates with the IOS streams facility to construct the version map. Because there is direct operating system support for process version maps, they function with any translator (including translators not supplied with the Domain system) that runs on the Domain system (provided the translator uses the Domain system IOS streams facility to read files or the built–in language input/output facilities of C, FORTRAN or Pascal).

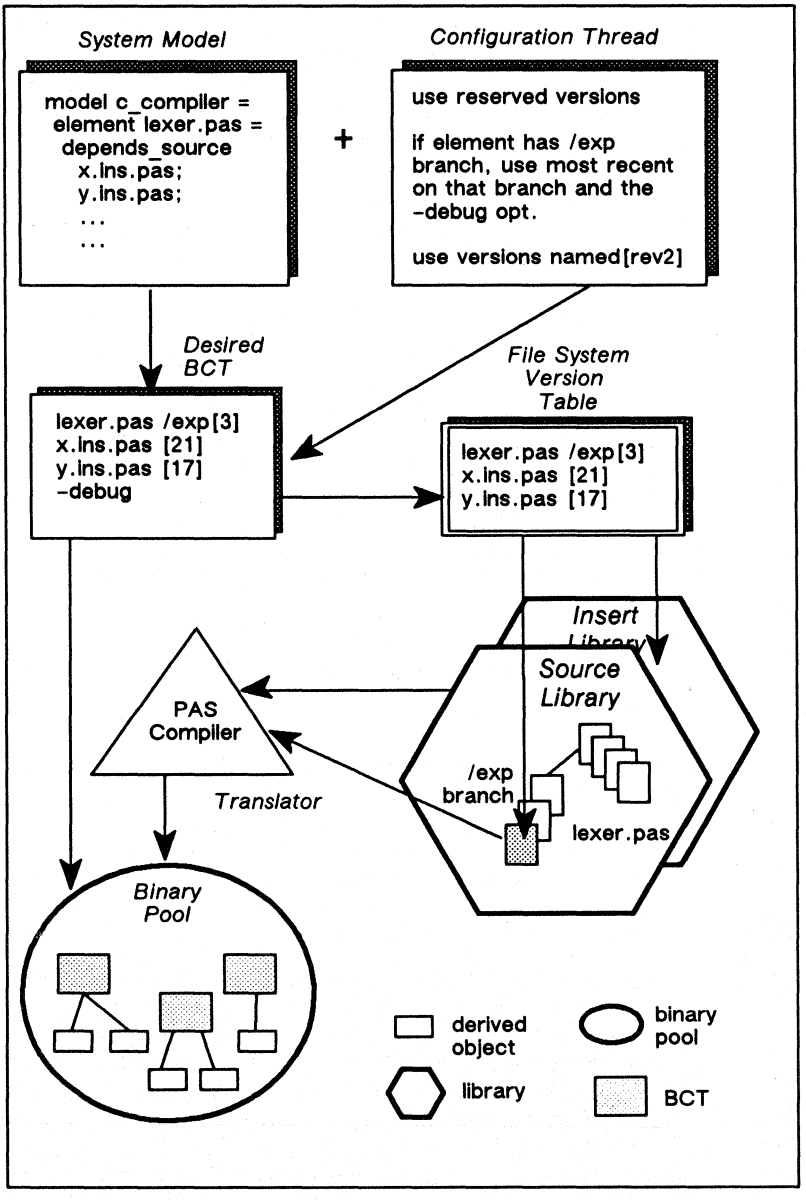


Figure 1–5. Overview of the Building Process

Third, the configuration manager invokes the actual translation rule in the specially prepared process environment. If the translation succeeds, as indicated by the exit status of the shell executing the actual translation rule script, the configuration manager places the BCT in the binary pool and associates it with the derived objects just produced. If the translation fails the configuration manager cleans up any partially built items and aborts the build. When a build fails, the configuration manager continues to build other components that do not depend on the component that failed. This might mean that the modules of a program that have no compilation errors are built, but the whole program is not bound together if any module fails to compile.

Parallel Building

By default, the node at which you invoke a build is the same one that performs the build. However, you have the option of choosing another node to build your system. Even more significantly, you can choose to have your system built by many different nodes; when possible, the builder nodes can execute your component builds in parallel.

Parallel building on many nodes significantly reduces the time required to build a system. You can expect a parallel build of your system to be three to ten times faster than a serial build. (The degree of parallelism in your system influences the speed of a parallel build versus a serial build. The degree of parallelism inherent in your model—that is, the degree to which component builds are independent of one another—is a major determinant of your system's degree of parallelism. Another determinant is the length of your translations. Longer translations provide more time for builds to be started in parallel.)

Parallel building also maximizes use of network computing resources. In most networks, many nodes are more than 90% idle at any given time. Their users are either not using the nodes at the moment or are performing tasks that are not resource intensive, such as editing. Parallel building takes advantage of this idleness by putting nodes to work building system components. Foreground users of the nodes rarely notice any disturbance. (We will discuss this concept in more detail later in this section.)

To build your system in parallel, you must first identify a list of candidate builder nodes. This list can contain the names of as many as 1000 nodes. At the same time that you provide your list of builder candidates, you identify the degree of parallelism (up to 20 simultaneous builds) that you want to achieve.

When you subsequently initiate a build, the configuration manager constructs a **partial ordering** of the components that need to be built. This partial ordering specifies which builds can be performed in parallel and which builds can't be started until other builds are completed.

The next step is for the configuration manager to form a translation script for the first component from the partial ordering. The configuration manager then examines the nodes on your list of builder candidates and determines which node to use for the component build based on idle time percentage—that is, the ratio of unused compute cycles to total compute cycles—over the last minute or so. Next the configuration manager creates a new process on the build server with a version map that causes translators to transparently read the desired versions of source elements. The translation script is then executed in the new process, and its output is accumulated in a temporary file. (This output is accumulated before being displayed in the output window to avoid confusing you. Otherwise, translator messages issued during simultaneous builds would be interleaved in the output window, making them hard to sort out.)

Once the build is started, the configuration manager goes on to the next component from the partial ordering. If it can be built in parallel with the first build, the configuration manager repeats the process. If the next component depends on a build in progress, the configuration manager waits to submit the new build until the other build completes.

During build execution, the configuration manager shows you the accumulated output of translation rules and configuration manager messages about invoking and completing builds. The configuration manager also shows you the current status of the distributed build: how many component builds are required, how many have completed successfully, how many have failed, and how many are being executed at the moment. This information is updated continuously throughout the build.

If a builder node crashes during a parallel build, the configuration manager treats the situation like any failed build: it deletes any partial results from the binary pool and does not build any other components that depend on the unbuilt component.

When many nodes are executing translation rules, a primary concern is that pathname resolution is consistent: for example, you probably want the same version of a C compiler to be compiling all of your C source code. Rather than forcing you to incur the overhead of maintaining a consistent set of sources and tools on every build server candidate, the configuration manager uses a **reference node**, or common local file system root, that you choose, to resolve relative pathnames. Use of the reference node ensures that, no matter which node is performing a particular build, such references as **/bin/cc** and **/usr/include** in the translation rule will always be resolved to a single, specific location.

Another important issue in parallel building involves node choice. The configuration manager selects builder nodes on the basis of their relative idleness: nodes that are more idle than others are selected first. However, nodes that are relatively idle when you initiate your build may become active later during the build.

The configuration manager avoids the problem of sending builds to active nodes (and thereby inconveniencing other users of the network) by keeping statistics about the idle time percentage of each candidate builder throughout the build. If a builder node becomes busy with another resource-intensive process during a build, the configuration manager will not submit another build to the node after the current build is completed. (The impact of the build on the other process running on that node is minimal, typically lasting less than a minute.) If a once-busy node becomes idle during the build process, the configuration manager realizes this and considers the node a good candidate for upcoming builds.

Promoting Derived Objects

In our discussion of binary pools, we mentioned that the configuration manager creates a binary pool, called the reserved pool, to hold the results of building components that depend on elements that you have reserved. Placing the derived objects of elements under development in the reserved pool ensures that these derived objects don't compete for system pool space with derived objects that need to be shared.

When you finish testing your changes to the elements that you have reserved, you don't want to have to rebuild the components that depend on them after you replace the elements' lines of descent. Therefore, the configuration manager **promotes** the derived objects and BCTs associated with an element you replace from the reserved pool into the appropriate system pool when you issue the **replace** command. The configuration manager updates the version descriptions in the associated BCTs to refer to the new version of the element in the library.

The configuration manager promotes a derived object from the reserved pool only if you've replaced all of the reserved lines of descent used to construct that derived object.

Avoiding Unnecessary Rebuilds

In a large system an element may be listed as a dependency for hundreds of components. For example, an include file that contains global declarations may be needed by most of the source modules in the system. Changes to such an element would normally cause the configuration manager to rebuild most of the system's components if the configuration thread calls for the most recent version of the element. This is frequently unnecessary because the changes are such that only a few components are affected. Additions of new declarations to an include file like the one mentioned above, for example, may affect only a few components that depend on the file.

When testing your changes in such a situation, you wouldn't want the configuration manager to rebuild every component of the system. That would take time and machine resources. Instead, you'd want the configuration manager to rebuild only those components that you know are affected by your changes, and to reuse existing derived objects that you know are equivalent for the unaffected components.

The configuration manager allows you to declare **equivalences** for components that it would otherwise build. The configuration manager interprets your declared equivalences to mean that, for certain components, derived objects built from the new element version are interchangeable with ones built from another version.

You can specify that an equivalence is in effect for either the duration of your current build or until the equivalent derived objects are purged from their binary pool. Equivalences in effect only for the current build are called **overrides**.

The configuration manager stores long-term equivalences in the binary pool as BCTs that have no derived objects of their own but refer to the derived objects associated with other BCTs. When you build the system, the configuration manager treats the equivalent BCT like any other BCT in the pool. If it matches all or part of the desired BCT, the configuration manager reuses the derived object to which it refers. Storing equivalences in the pool makes them available to other users, too, who may not be in a position to determine which components were affected by your changes. You can request that the configuration manager ignore existing equivalences for a particular build.

Figure 1-6 shows how an equivalent BCT stored in a binary pool can point to another BCT's associated derived objects.

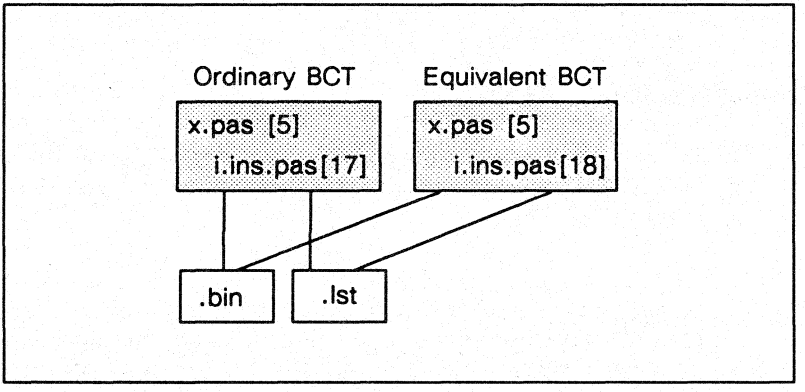


Figure 1-6. Pictorial Representation of an Equivalence

The configuration manager provides another way to avoid unnecessary builds: by declaring **noncritical dependencies**. This capability is provided by the system model language. It allows you to state that, for almost all cases, the configuration manager shouldn't rebuild a component simply because the element listed as a noncritical dependency of the component has changed. If, while checking a desired BCT against the BCTs in the system pool, the configuration manager finds one that matches the component's desired BCT in all respects except for the specification of the noncritical dependency, the configuration manager uses the derived objects associated with the closely matching BCT rather than rebuilding the component.

Your development team might declare dependencies on global include files to be noncritical, since you would not normally want the configuration manager to rebuild all components when you add new declarations to the include files. Usually, you have to change the source modules that are affected by new declarations in the global include files anyway, so that these modules use the new declarations. When you build your system, the only desired BCTs that differ from existing BCTs by more than the change to the noncritical dependency are the ones involving modules affected by the include files' changes.

Implementation Details

The DSEE configuration manager is a sophisticated and complex facility. It provides virtually comprehensive control over system builds.

At first glance, it might seem that the configuration manager must be very slow if it is providing all of the control described in this section. However, we have implemented several optimizations in the manager to ensure that the capabilities it provides do not come at too high a cost of speed.

The most important optimizations we incorporated in the configuration manager's design are based on the rate and magnitude of change in your configurations from build to build. The configuration manager stores your last desired BCT because, under most circumstances, each successive build you do differs from your last build by only a few factors. Each time you do a build, the configuration manager reuses those parts of your last desired BCT that are appropriate to the current build.

Also, because you're likely to do many consecutive builds using the same lines of descent, the configuration manager keeps track of what the most recent versions are on those lines of descent. As a consequence, the configuration manager doesn't have to spend a lot of time during each build resolving "use most recent" specifications in your configuration thread.

Release Manager

Most of the builds you perform during the normal course of development produce derived objects that are of short-lived utility. They quickly become unused as new element versions are created, and they and their BCTs eventually disappear from the binary pool.

The situation is very different when you build a system to release for distribution. In this case, you want to have permanent copies of some or all of the derived objects and a permanent record of the element versions and translation rules used to build them.

The DSEE release manager stores derived objects that you want to release, along with their associated BCTs. Using the release manager, you can preserve all or part of a build and keep copies of the system's dependencies.

Integration of the configuration and release managers means that you can, in your configuration thread, refer the configuration manager to the BCT of a released build for version and option specifications. This is very useful when you are doing maintenance to a released system, or when you simply need to rebuild a released system.

Components

The sole component of the release manager is the **release area**, a directory that the release manager creates, at your request, to store a build. A release area has two subdirectories: one to hold the derived objects and BCTs of each built component of build you are releasing, and one to hold source copies of the dependencies you want to retain.

When we discussed the configuration manager above, we noted that you cannot access derived objects directly unless you export them. The configuration manager manipulates the objects for you. However, when you release a build, you can access its derived objects. The release manager copies them into the release area and uses component names to name them.

Using the Release Manager

As mentioned, the release manager creates a release area at your request, structuring the subdirectory tree structures and placing derived objects, BCTs, and dependencies in the appropriate directories. You can later add to the release and examine the release's contents.

We have already discussed the primary uses of releases. The principal use is as distribution-ready software. Another important use of a release is as a reference in later builds.

When we explained configuration threads, we noted that you can refer to the BCT of a previous build in your configuration thread. Such a configuration thread rule (known as a **build-ID-based rule**) can refer to a build in a release area. This is particularly helpful when you are fixing bugs in a prior release. You can structure your thread to give you builds that use exactly the same element versions and translation options that went into a released build for all the components of the system that don't require changes simply by referring to the name of the released build.

Task Manager

The DSEE history manager provides a convenient way to record descriptions of the modifications to an element when a new version is created. In large systems, however, there are few modifications that affect only a single element; most significant enhancements and many bug fixes require changes to several elements. It is desirable to have a mechanism for remembering all of the modifications that were performed as part of one higher-level task.

Many, but not all, of the steps taken to accomplish a task modify elements. For instance, adding an enhancement to a system may also require updating the system's design specification, user manual, and online help files, in addition to changing the program code stored as elements. Some steps may involve offline activities, such as giving a talk about the enhancement, constructing floppies for the enhanced system, and telephoning customers. In short, the software development process involves much more than just programming. Therefore, a practical software development environment should support more than just programming.

The DSEE task manager provides a way to plan and track the low-level steps involved in some high-level activity. The task manager can maintain a record of these changes automatically, making it easy to determine at a later time exactly what was done as part of the task.

Components

The task manager has three components, which we define and discuss in this section:

- Tasks
- Tasklists
- Forms

Tasks

A DSEE **task** is a structure used to plan and record the low-level steps involved in a high-level activity. A task consists of a user-supplied title, which describes the high-level activity, a list of active items, and a list of completed items (also known as the **task transcript**).

Active items represent anticipated steps that are yet to be taken. Together, these steps form a plan of action. **Completed items** represent steps that have been taken. Many completed items are former active items that you have checked off (marked as completed). The history manager also signals the task manager to add completed items to the task transcript automatically, as we discuss in the section entitled "The Task Manager and the History Manager."

Tasklists

DSEE **tasklists** contain references to tasks. A tasklist serves as a list of high-level activities that need to be done. Each user has a personal tasklist. Additionally, each library contains two tasklists—one for active tasks and one used mainly for completed tasks. (The latter tasklist is called the library's **master tasklist**.) You can also create any number of additional tasklists.

Several tasklists can refer to the same task, since several people might be involved in the completion of one task. In this case, each user sees items completed by other users immediately, since tasklists contain references to tasks, rather than tasks. You can add task references to other users' tasklists, subject to access control considerations (discussed later in this chapter under "Security and Protection").

Forms

The task manager lets you develop and use standard task **forms**. Once you have written a form, you can use it to create new tasks, editing the text as appropriate for each specific task. A form outlines a series of steps that you must perform for a certain type of task that you do frequently. For example, you may find that you are executing the same sequence of procedures for each bug fix. If so, you might create a form that lists these steps. You can use this form as a basis for each distinct bug fix task that you have.

You can create a form from scratch, or you can use an existing task as a foundation for a form.

Using the Task Manager

The task manager automates a large part of your use of tasks and tasklists. It works with the history manager, so that task steps that involve element modifications can be recorded as completed items automatically.

Because accomplishing a task can involve more than just modifying elements, you can't rely on the task manager to keep them up to date automatically. Some things you will probably want to do yourself. The task manager provides the facilities you need to manipulate tasks and tasklists. You create and modify tasks yourself with the task editor, and you add and remove tasks from tasklists using DSEE commands.

The Task Manager and the History Manager

The history and task managers work in close communication with one another. In particular, when you replace a line of descent of an element, the replacement operation affects the text of your current task, and the current task affects the information recorded in the element's library's history database.

The task manager adds a completed item to the task whenever you replace an element's line of descent in any library. Like the entry that the history manager makes in the library's history database when a line of descent is replaced, the completed item contains your name and network location, the date and time, the name of the element that you replaced and the version that was created, and your description of the change. (Several users can have the same current task simultaneously. In this case, the task transcript reflects the activity of all such users.)

When you replace a line of descent, the history manager includes the name of the current task in the record it makes in the library's database as a result of the operation.

Creating and Modifying Tasks

You can create tasks in two ways: you can create them from existing forms, and you can create them with the task editor. Once you have created a task, you can modify it using the task editor.

You create a task from a form by adding an option to the **create task** command. You specify, as an argument to the command option, the form you want the task manager to use as a boilerplate for the one you're creating.

The **task editor** is an interactive, menu-based editor that you invoke with DSEE commands. With it, you can create or modify a task title, add, change or delete active items, assign active items to particular people, change the priorities of active items, and move active items to the completed items list. (This last procedure is called checking off an item.)

Using Tasklists

Tasklists are ordered lists of tasks to complete. They serve as the mechanism by which you refer to specific tasks. To set, edit, or delete a task, you refer to it by its number in your current tasklist.

Each time you create a task, the task manager asks you on which of several tasklists you want the new task recorded. Later, you can add the task to other tasklists: your own, or someone else's, as long as your access rights permit it. (See the section on the DSEE protection mechanism for more information on access rights.) For example, you might want to add a reference to a task that appears on a tasklist of open bugs for an upcoming release to your own personal tasklist, too, since you are responsible for fixing that particular bug.

While you are working, you can examine tasklists to see which tasks you still need to perform. When you finish your part of a task, you remove the reference to the task from your tasklist. When no tasklists refer to a task and all of its active items have been completed and checked off, the task is completed. The master tasklist that contained a reference to that task, however, keeps the task record. This reference serves as a historical reference of the task.

Implementation Details

Because you use DSEE software in a distributed environment, we gave close attention to the implementation of task and tasklist operations that may involve more than one node in the workstation network. For example, if you create a new version of an element, the event is recorded in the task transcript of your current task. However, the library where the history manager creates the new version may be on a different node than the library where the task is stored. This could present a problem if the network is partitioned when the new version is created: specifically, how is the task transcript updated?

The task manager uses a reliable (store-and-forward) message passing utility to guarantee that the update occurs. There is a delay between the creation of the new version and updating the task transcript if the network is partitioned. Otherwise, the update occurs immediately.

The store-and-forward mechanism is used similarly in other operations that access objects on different nodes in the network. Besides providing reliable delivery of messages to other nodes on the network, the store-and-forward utility provides the capability for sending messages across inter-network gateways. Therefore, the DSEE architecture allows the same task to be referred to by users on more than one local network.

Monitor Manager

As we mentioned earlier, the DSEE configuration manager monitors system include dependencies and detects when parts of the system need to be rebuilt. These are **syntactic dependencies**, so named because the syntax that outlines the structure of the system defines the dependencies.

There is another type of dependency tracking, however, that is not addressed by the configuration manager. This type of dependency is more people-oriented than build-oriented. It involves letting people know what's happening to source code and text in which they have an interest. Such dependencies are called **semantic dependencies**.

A good example of a semantic dependency arises in the interface between a development team and the technical writers supporting them. The writers need to know when functional specifications and design notes change so that they can keep on top of the developing product. Unfortunately, when changes are most critical, engineers are at their busiest. Under these circumstances, it's likely that they will occasionally forget to let writers know about design changes.

As a second example of a semantic dependency, suppose you write a module that depends on functions in another module, but this dependency isn't reflected by the system structure. It's not only useful for you to know when the module you depend on changes, it's also worth notifying anyone who wants to change that module that the changes might impact your work.

The DSEE monitor manager notifies users when particular elements are changed. It also lets users know when elements they are modifying might affect others. The monitor manager works in concert with the task manager and the operating system to provide a flexible means of communication.

Components

The sole component of the monitor manager is a **monitor**. A monitor watches elements and lets users know when the elements are modified. Activating a monitor can also trigger execution of a list of shell commands.

A monitor consists of a title that describes its purpose, a list of the elements that it monitors, a **task template** to be instantiated (that is, copied as an instance of a task) when the monitor is activated, a list of the tasklists to which the instantiated task should be added, and a list of the shell commands that should be executed. However, monitors needn't instantiate tasks; you might, for example, create a monitor that simply sends mail to you or another user whenever it is activated. (This is useful when you want to notify someone who doesn't use DSEE facilities of a change to an element.)

Using the Monitor Manager

For the most part, you work with monitors in two ways: you create them, and you activate them. We discuss each separately.

Creating a Monitor

When you create a monitor, you specify the following:

- Which elements you want to monitor
- Who you want to activate the monitor
- What you want to happen when someone activates the monitor

The following paragraphs discuss these specifications.

The elements that you want a monitor to watch are called the **target elements** of the monitor. Target elements of a monitor all reside in the same library. You can use regular expressions and wildcards in your list of target elements. This not only allows you to monitor a group of elements in one library easily; it also ensures that any element created in that library in the future whose name matches your regular expression will be monitored.

When you create a monitor, you can specify that it can be activated only when you replace a line of descent of a target element. Alternatively, you can specify that the monitor can be activated only when someone else replaces a line of descent of a target element. If, for example, you want to set a monitor on an element to watch for access by other users, but you also access the element frequently yourself, you would want to specify that the monitor is only activated when someone other than you reserves one of the element's lines of descent.

Specifying what you want to happen when someone activates your monitor is a matter of describing the task template (if any) that you want instantiated when the monitor is activated and the tasklists the new task is to appear on, and/or writing shell commands that you want executed when the monitor is activated. Task templates are quite similar to task manager components like forms and tasks. Shell commands are passed directly to the shell. Therefore, you can use conventional shell command syntax (including regular expressions and wildcards) when composing them.

Shell commands can also contain **activation strings**, which serve as substitution arguments. The monitor manager defines unique activation strings for such items as the name of the element whose use triggered the monitor, the name of the user triggering the monitor, and the number of the new element version created. When your monitor is activated, the monitor manager replaces any activation strings with the accurate information.

Activating a Monitor

When you reserve a line of descent of a monitored element, the monitor manager informs you that the element is monitored and shows you the description of the dependency that the person who set the monitor is trying to track. This ensures that you don't modify the element without first considering the dependency.

When you replace a line of descent of a monitored element, you activate its monitor. This causes the task manager to create a new task using the template (if any) that the person who set the monitor created. The task manager adds a reference to this new task to every tasklist listed in the monitor. The DSEE tasklist alarm server can send notification to personal tasklist owners notifying them that the task manager has added a new task to their tasklist.

When someone activates a monitor, the monitor manager passes any shell commands in the monitor on to the shell. As mentioned, the monitor manager first replaces all activation strings with the appropriate information (for example, the name of the person who triggered the monitor). The shell then executes the commands as it would any other command.

Implementation Details

As with certain task manager operations, monitor activation can involve accessing objects on different nodes in the network. Network partitioning may cause some of the nodes to be temporarily inaccessible. The store-and-forward message passing utility (which we described in the "Implementation Details" section of the discussion on the task manager) handles this complication gracefully, allowing monitor activation to work across network partitions. When the network is no longer partitioned, the store-and-forward mechanism ensures that the monitor messages arrive at the correct destinations.

Integration of the Managers

In the discussion of each manager, we note how it interacts with other managers to provide you with a comprehensive work environment. If you turn back to Figure 1–1, you can see the lines of communication between the managers again. Now that you know more about each manager's components and behavior, you'll appreciate the integration depicted by Figure 1–1 more.

Figures 1–7a and 1–7b show the DSEE environment's integration from a more detailed level. In them, you can see how each manager's components interface with other components.

The library code_lib shown in Figure 1–7a contains the elements labeled **bar.c**, **cxt.c**. and **dtty.c**. (Element **cxt.c** has a branch line of descent.) All version and branch activity is recorded in the library history database). Monitor creation is also recorded in the database, as shown by the arrow from one of the two monitors that the library owns. Monitors can add new task references to tasklists, as the arrow between one of the monitors and the personal tasklist show. The current task is also part of the tasklist. Arrows between the task and the database demonstrate the passing of information between the two: when an element is replaced, the history database records it in the current task's transcript and records the name of the current task in its own record of the replace operation.

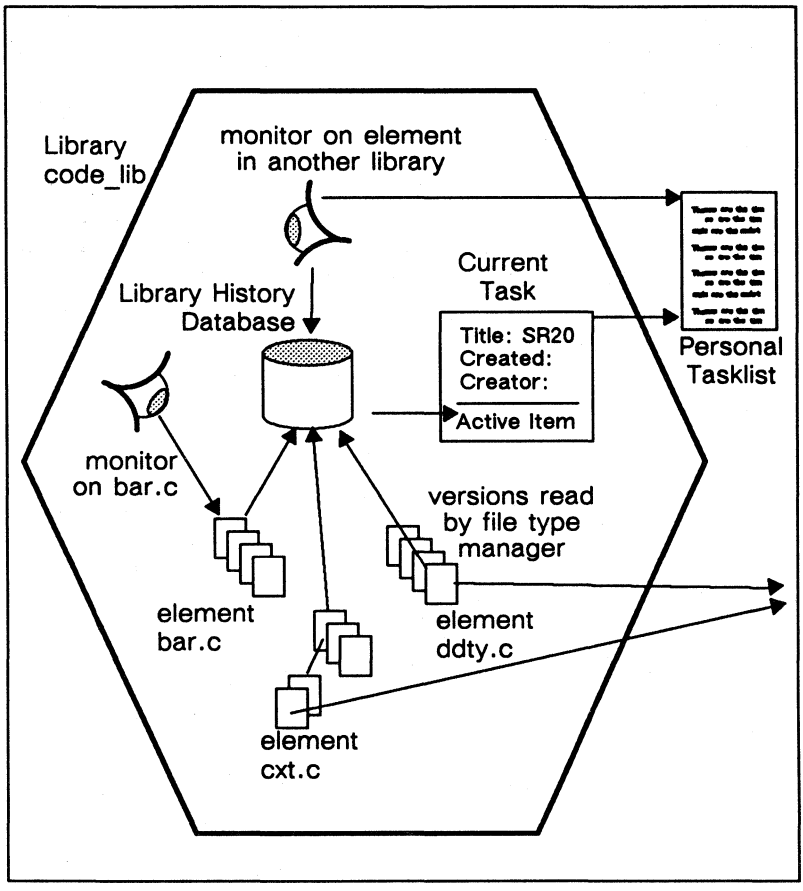


Figure 1–7a. Interaction of DSEE Components

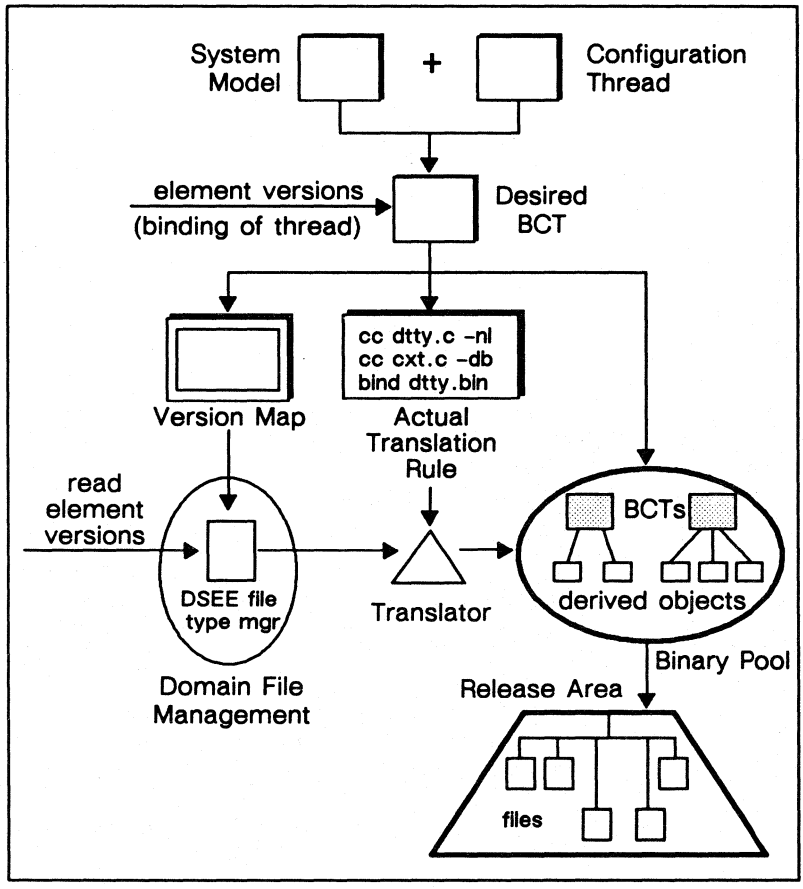


Figure 1–7b. Interaction of DSEE Components

Figure 1–7b depicts components of the configuration manager and the release manager. When you build a system, the configuration manager uses the system model and the configuration thread to create the desired BCT. Element version specifications in the configuration thread are resolved in the binding process. The configuration manager looks in the binary pool for existing BCTs to satisfy all or part of the build.

From the desired BCT the configuration manager creates a version map and the actual translation rule. The version map communicates with the DSEE type manager, which then reads the appropriate versions of elements and passes them along to the translators invoked by the actual translation rule (note that the translators are not DSEE objects). The BCTs and derived objects go into the binary pool. When you release a build, the release manager creates a release area and places BCTs and derived objects in it as files.

Security and Protection

The DSEE environment uses a protection mechanism based on Domain system access control lists (ACLs) to ensure that DSEE objects (for example, libraries and tasklists) are secure and are accessible only by users with the appropriate access rights. The DSEE protection mechanism defines four classes of users.

- **Non-users**, who are denied access to DSEE objects
- **Readers**, who can read protected objects and/or their contents but cannot modify them
- **Members**, who can modify protected objects and/or their contents but cannot delete them
- **Administrators**, who have full access rights to all DSEE objects

The DSEE software creates this buffer between you and the Domain ACLs that protect DSEE elements, libraries, and other objects to simplify protection for you. The implementation of a DSEE environment is a much more complex structure than you actually see. Therefore, you would find it very difficult (if not impossible) to protect all of the files and directories that DSEE facilities use consistently and correctly (that is, setting ACLs appropriately for each directory and file in the structure). The DSEE protection mechanism automates this procedure.

To protect a DSEE object (for example, a library), you identify users or groups of users with subject identifiers (SIDs), as you would when creating or changing a Domain system ACL. However, instead of dictating the specific types of access each user has, you associate with each SID a DSEE protection user class (one of the four categories listed above). This causes the DSEE protection facility to protect all the elements in the library consistently.

The DSEE protection mechanism prevents people who don't use DSEE facilities from accessing DSEE objects, and prevents readers from modifying DSEE objects, either intentionally or accidentally. In addition, it prevents project members and administrators from accidentally deleting elements that they did not intend to delete.

Customizing the DSEE Environment

While the DSEE command interface provides you with commands to execute all DSEE facilities, there may be situations in which you want to tailor the interface, or to change it altogether. You can customize your DSEE environment by:

- Writing command files that perform multiple DSEE commands
- Using the programmable interface to the DSEE environment and executing DSEE operations from C or Pascal programs
- Embedding DSEE commands in shell scripts and passing them to a DSEE server for execution

We describe all three methods in the following sections.

Command Files

A **DSEE command file** is a script of DSEE commands that you invoke through input redirection. You use DSEE command files very much as you use shell command scripts, but you invoke DSEE command files at the DSEE prompt rather than the shell prompt. Like shell scripts, DSEE command files can contain parameters for arguments supplied when the scripts are invoked. Command files are particularly useful when you have a series of DSEE commands that you regularly perform, or when you must perform the same DSEE command on many DSEE objects.

Several DSEE features facilitate DSEE command file programming. An option to many history manager commands allows you to format the output, embedding history manager information in strings also containing DSEE commands. This formatted output can be redirected to a command file script. Using this technique, you can, for example, issue one DSEE command that constructs a command file to reserve all the elements in a library.

Another feature that aids command file programming is the ability to specify the severity of error on which you want the command file to abort. Abort severity is inherited by all nested command files by default, but you can also specify different abort severities at each level of nesting.

By default, the DSEE environment looks to an executing command file to get responses to command queries. However, you can, if you want to, write command files that execute interactively and accept responses from standard input.

You can pass DSEE command files parameters from the command line. You can use up to nine substitute parameters in a DSEE command file.

Programmable Interface

The **DSEE programmable interface** allows you to execute DSEE commands, access DSEE library databases, and access and manipulate system model information from Pascal and C programs. Using specialized calls, you can write programs that reserve and replace elements, build systems, and perform many other DSEE functions. Another call allows you to send any DSEE command to the DSEE facility; you simply use the appropriate command syntax and send the command as a string. Other calls let you change input and output streams.

Using calls to library databases, you can write programs that provide you with the kind of historical information that you require from the databases. You can, for example, write programs that show only one user's reservations, or write programs that scan comments associated with various events in an element's history for a particular word or phrase.

Other DSEE procedures allow you to manipulate sets of system components. In this manner, you can obtain such information as the names of all the components that depend upon one particular component and which components are shared dependencies of multiple components.

In short, you can use the programmable interface to:

- Perform many DSEE functions that you perform interactively
- Tailor your access of the history database to meet your specific needs
- Retrieve and manipulate information about system structures

The *Domain Software Engineering Environment (DSEE) Call Reference* provides complete details on using the callable interface. Examples of programs that invoke DSEE routines are included in DSEE software shipments, as well as instructions on how to use them.

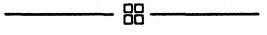
DSEE Server

The **DSEE server** enables you to embed DSEE commands in shell constructs. In this manner, you can perform DSEE operations while taking advantage of the control flow inherent in shell languages. You can execute DSEE commands from within conditional constructs and loops, and pipe command output to shell commands.

There are actually two DSEE server utilities: **dsee_server**, which enables you to combine DSEE commands with any shell supported by Domain/OS (the Aegis, BSD, or SysV shells); and **dsee_server_c**, which is tailored for use with UNIX* shell commands. We ship both **dsee_server** and **dsee_server_c** as examples of programs using the DSEE programmable interface. The source and executable code are included in DSEE software shipments.

DSEE Concepts: Conclusion

In this chapter we've attempted to give you an overview of how DSEE facilities work. In the next three chapters we present examples of how engineers use DSEE facilities to perform their own work.



* UNIX is a registered trademark of AT&T in the USA and other countries.

Chapter 2

Case Study 1: Converting to a DSEE Environment

In this and the following two chapters, we examine how engineers use the DSEE system to facilitate their work. Each chapter focuses on one particular engineering project that is managed with DSEE software.

Each project we examine has some features that make it a unique application of DSEE facilities as well as some that are of interest to all DSEE users. By presenting both the unique and the widely applicable features of these projects, we hope to help you use the DSEE environment in the most effective manner.

The main body of each chapter discusses how the engineers involved in the project work, given those features that make their project a unique DSEE application. We present information of potential interest to all DSEE users in special, shaded sections called "highlights."

In the appendixes, we present scaled-down versions of the system models that the groups use. Appendix A contains a system model related to this chapter's text. Chapters 3 and 4 discuss groups that use the system models presented in Appendixes B and C, respectively.

In this case study we focus on how an ongoing engineering project changed its method of work to employ the DSEE environment. We discuss the process of setting up a DSEE environment. In particular, we examine the writing of a system model to represent an engineering project.

After we introduce the project and discuss its personnel and goals, we present the DSEE objects, such as the libraries, monitors, and system models, that structure the group's DSEE environment. Finally, we observe how the engineers work using DSEE facilities.

Introduction

The project that we observe as it undergoes conversion to the DSEE environment is our company's computer–aided design (CAD) tools group. This group produces a software package that our in–house logic designers use to design printed circuit boards (PCBs).

Briefly, the CAD tools group's product is a set of commands that bridges the gap between data formats used by various vendors of logic design tools and manufacturers of PCBs. Engineers use these commands to translate logic designs in disparate formats into a single format and store the designs in a database. Later, using other commands in the same package, the engineers can convert these stored designs into data formats used by manufacturers of PCBs.

Although the product that the CAD tools group produces is for internal use only, it must be maintained as any revenue–producing product, and the group needs to change and develop aspects of the software in response to user needs.

Converting to a DSEE Environment

The CAD tools project started in 1983 and converted to the DSEE facilities in 1985. Before the project converted to a DSEE environment, the number of commands in the product grew steadily. Eventually, the product became unwieldy.

The CAD tools group decided to use DSEE facilities to manage their product because of the benefits derived from using the history manager as well as the configuration and release managers. (We explain the specific problems the CAD tools group had and the ways that DSEE facilities solved them as we progress through this case study.)

Once the CAD tools engineers determined that they should be working in a DSEE environment, they had to change both their source storage methods and work habits to employ DSEE facilities.

The CAD tools engineers converted to their DSEE environment in two stages. First, they put all of their source code into DSEE libraries. They created all their libraries and elements at one time and immediately started using them so that they could avoid the maintenance problems that are associated with attempting to work with two parallel development structures simultaneously.

Once they'd migrated to DSEE history management for their source code control, the CAD tools engineers developed an appropriate system model. After the engineers wrote and debugged their model, they started using it exclusively for their configuration management.

Project Structure

Much of the effort required to convert to a DSEE environment involves setting up the project structure (that is, establishing libraries, writing models, creating system pools, etc.). Once you have designed the project structure, you can migrate to the DSEE environment and use it with relative ease.

In this section, we address many of the design issues that you might encounter when setting up a project structure. In particular, we examine issues concerning libraries and elements, and we pay close attention to the process of writing a system model.

Highlight: The Role of the DSEE Environment Administrator

Most projects that use DSEE facilities have one or two **environment administrators**—engineers who have a very detailed knowledge of the DSEE environment. These administrators usually take on such responsibilities as setting up the appropriate libraries, protecting DSEE objects, and writing system models. Administrators also frequently serve as the projects' troubleshooters for problems related to DSEE use.

Here are the functions that the administrator might perform:

1. Determine the appropriate library structure for the project's modules, create and protect the libraries, and create the elements
2. Write one or more system models that construct the project's products, and create and protect the necessary system directories and derived object pools
3. Establish naming conventions for branches and releases (and, perhaps, establish a directory to hold release areas)
4. Document the library structure, system use, and naming conventions
5. Write scripts of DSEE commands
6. Act as a resource for coworkers on DSEE information and behavior
7. Monitor the environment to ensure that project members are using the structure and naming conventions consistently and that the structure is appropriate for the project
8. Modify the structure or naming conventions if they prove to be inadequate or inappropriate for current project activities
9. Edit the system model as the system changes

Libraries and Elements

One of the group's chronic project management problems that drove them to convert to a DSEE environment was source code storage structure. During the two years that the engineers worked on the project before converting to DSEE facilities, their source code storage method had evolved into a large, inconsistent directory structure. Although each command had its own directory, these directories weren't at the same level in the storage tree. Finding the right directory involved a frustrating search up and down the structure. Working in such a complex structure was confusing. Engineers rarely knew the full structure of the source area and frequently made mistakes.

Although the CAD tools group could have replicated their old storage technique by placing all the code relevant to one command in one DSEE library, they chose not to. They took the opportunity provided by the conversion to simplify their storage technique. What was once stored in a labyrinth of 70 nested directories now resides in eight DSEE libraries. Table 2–1 lists the libraries and their contents.

The library **//max/cad/database** contains source elements for the commands that create and manipulate the database in which designs are stored. The library **//max/cad/library** contains source elements for the commands that create and manipulate the libraries used by the database. The source modules for other commands are stored in **//max/cad/applications** and **//max/cad/utilities**. The **//max/cad/ins** library holds all the include files used by the modules in the four libraries just described. Scripts of CAD tools commands that are included in each release of the product reside in **//max/cad/scripts**.

The last two libraries listed in the table, **//max/cad/build** and **//max/cad/tests**, contain elements that are not source components of the product. The **//max/cad/build** library contains the system models, DSEE command scripts, and some shell scripts that the engineers use. The **//max/cad/tests** library holds source modules that the testing system uses to test the product.

It's important to note that the CAD tools group didn't lose any of the organizational information provided by their old directory structure by placing the code in only eight libraries. The history manager's library databases provide the same information in a much simpler, more readable form.

Table 2–1. CAD Tools Group Libraries and Their Contents

Library Name	Contents
//max/cad/database	Source files for CAD database management system
//max/cad/library	Source files for CAD library–related commands
//max/cad/applications	Source files for applications–oriented CAD commands
//max/cad/utilities	Source files for CAD utilities routines, design interface routines, and several LIST commands
//max/cad/ins	Include files used by CAD source code in first four libraries
//max/cad/scripts	General–purpose CAD shell scripts
//max/cad/build	System models, DSEE command scripts, and miscellaneous shell scripts
//max/cad/tests	Regression tests for all CAD commands

Highlight: DSEE Performance and Library Structure

You may wonder, as you prepare to create new libraries and elements to hold your source code, what impact your new storage scheme might have on DSEE performance. Is there a maximum number of libraries to which you can refer in your system model before the configuration manager's performance is slowed? Does it matter to the configuration manager where the libraries are stored? Do either the configuration manager or the history manager perform better with one structure for libraries (for example, when all the libraries are at the same level in a directory structure, or when libraries are stored widely scattered in several directory structures)?

In general, you should decide on your library storage scheme based on what's right for your project. However, you should avoid creating many libraries with a few elements in each. There are two reasons to avoid such a scheme. For one thing, it's easier to get all the information you want from the history manager if you use fewer libraries. This is because the context for history manager commands is the current library. If you want information on many elements (for example, if you want to see the names of all elements on which you have reservations), you must reset your current library to each library containing project elements and issue the history manager command. The more libraries you have, the more commands you have to issue to obtain your information.

The second reason to avoid using too many libraries has to do with performance. Because the history manager maintains a separate database for each library holding your system elements, operations that span the system's libraries are subject to a per-library time overhead for the process of opening each database. The more libraries you have, the longer the operation can take.

(continued)

Highlight (continued)

An example of such a situation occurs during configuration thread validation. When the configuration manager validates your configuration thread, any version rules that use wildcards in the rule qualifier and identify the version to use by branch path, branch name, or version name may force the configuration manager to look in the database of every library listed in the system model that contains an element whose name matches the wildcard. The more libraries that must be searched, the longer thread validation can take.

Don't feel, though, that you must have only one library containing all your source elements. Logical grouping of elements is always helpful and should not be avoided. What you should avoid is the use of a great many libraries, each with less than 20 or so elements.

There are several performance considerations that may affect your choice of storage nodes for libraries. In general, though, you can take advantage of large amounts of space available on file servers for libraries containing many elements and enhance DSEE performance. And there is no optimal directory tree pattern for library storage: if storing all your libraries within different directory structures facilitates your work, then you should do so.

Think carefully about your libraries and their contents before you create them so that you can avoid the maintenance overhead involved in moving existing libraries. When you move an existing library, you have to change all references by absolute pathname to it. Also, old system models that refer to the library's former location by absolute pathname won't compile. To rebuild an older configuration of a system, you can't just reuse the older version of the system model used in that configuration. You have to create a branch line of descent of the system model element whose origin is that version and incorporate the new library name into the branched version, or use links to make the old pathname point to the new library name.

(continued)

Highlight (continued)

To avoid this problem, you can refer to libraries in your system model using link-relative pathnames. For example, the CAD tools group might have made their libraries more portable if they'd referred to the libraries in the system model with pathnames like **/cad/database** (instead of **//max/cad/database**), and then had everyone in the group create links in their nodes' root directories called **/cad** that pointed to the appropriate directory. (The next case study contains an example of link-relative pathnames for libraries and pools.)

Placing Existing Source Code in Elements

Once the CAD tools group decided on their library structure, they created the libraries and set about moving their existing source modules into the libraries in the form of new elements. They used the **create element** command to move their source code into the libraries.

The **create element** command's default behavior is to look in your current working directory for a file of the same name as the new element you're creating; if such a file exists, the history manager uses that file as the first version of the new element and deletes it from the working directory. The CAD tools engineers decided that they wanted to retain the old source code files for a while after they created the elements, so they added the **-keep** option to their **create element** commands. They also used the **-from** option so that they could specify the location of the text for the first version of an element without having to change working directories. The **-from** option also allowed the engineers to give an element a different name from the filename that they had used.

In some instances, the engineers had created several different copies of the same source code, each corresponding to a version of the module. This had happened when they needed to make alterations to a module, but they wanted to keep the older text as well. Switching to the DSEE environment, the CAD engineers stored these source modules by placing the different renditions of the same source into different versions of the same element.

Highlight: Automated Library Population

If you have many existing files that you want to place in DSEE libraries, creating the new elements one at a time can be a time-consuming process. Fortunately, you can use one of several techniques to automate this process.

If, for example, all the source code that you want to place in one library resides in your current working directory, you could create a DSEE command file to populate your library by issuing a series of commands like the following:

```
$ ld -lf -c | chpat '%' 'create element ' | chpat '$' @
$_' -keep -com @"Ele created from existing src@"' @
$_ > cre_ele.dsee
```

Then you could execute the DSEE command script **cre_ele.dsee** in the DSEE environment to create your elements, as in this example:

```
DSEE> <cre_ele.dsee
```

If you would like to create a more versatile script that uses shell commands, conditionals and control flow in combination with DSEE commands to populate your library, you can use the **dsee_server** utility. You could also write a C or Pascal program that makes calls to DSEE routines to create elements. (Both **dsee_server** and the programmable interface to the DSEE environment are discussed in the section "Customizing the DSEE Environment" in Chapter 1.)

If your source code resides in an SCCS library (a UNIX source code control system), you can move your modules into a DSEE library using **sccs_convert**. This utility, like **dsee_server**, is shipped as an example of a program using the DSEE programmable interface.

Tasks, Tasklists, and Monitors

The CAD tools engineers create very few tasks. They do, however, set monitors. For example, Serge, one of the group's DSEE environment administrators, has monitors set on all the elements in every one of the group's libraries. Some of these monitors add tasks to his personal tasklist. Others send mail to him and to other members of the group.

Systems and System Models

Writing a system model is the key part of defining a DSEE configuration management environment. It involves describing the structure and translation of your product in terms unique to the DSEE facilities.

If you are converting a working project to a DSEE environment (as were the CAD tools engineers), writing a system model is an evolutionary process. Your existing build scripts are the bases for your translation rules. If you previously used tools that list required include files, you can use these lists as starting points for your system's source dependency declarations.

As you transpose your old working method into system model syntax, you will find that the block structure and scoping rules of the system model allow you to eliminate much of the redundancy that your old working method had. Nested blocks describe interdependencies for you, and default declarations serve to cut down on much of the repetition of information.

In this section we address many of the questions and concerns that you may encounter while developing your model. In particular, we focus on translation rules and dependencies, the two most important parts of model design.

For more information about writing models see the *Domain Software Engineering Environment (DSEE) Command Reference*. The chapters "Writing System Models" and "System Model Language: Declarations" of the Command Reference provide comprehensive information about the basics of system model syntax and composition.

Before we explore system model translation rules and dependencies in this section, we address two key ingredients in system configuration: systems and pools.

Systems

One of the initial questions that the CAD tools engineers asked themselves was whether it would be more appropriate to think of their product as one system or as several systems.

The engineers decided to represent the whole set of commands as one system, with one system model, for a few reasons. For one thing, such a representation is more convenient. The entire set of commands is their largest distribution unit. Having it all in one system allows them to build the entire product with one build command and to share easily those components that are used by more than one part of the product.

Version control is another reason the CAD tools group chose to use one system and one system model to represent the product. They wanted to make certain that each build of any part of the system allowed them full control over all source dependencies. If the engineers had decided to represent a certain portion of their overall product as a system (with its own system model) that was built individually and then imported into the main system, a build of the main system would only allow version control over the elements fully described in that system.

For example, the CAD tools group has a collection of utility programs called the **utilities** library. If they decided to construct **utilities** as a separate system, they could declare a tools dependency in the main system on the outcome of building the **utilities** system. As a result, **utilities** would be treated as an **imported derived object** in the main system. If the CAD tools engineers did this, however, they would have no version control over the imported system's constituent element versions when they built the main system.

In the next chapter we present a case study of a project that uses multiple systems. In this case, although the systems are built with the same source code, they are distributed as separate items. We examine the advantages of such a setup when we describe the product.

Highlight: Pros and Cons of Imported Derived Objects

As we have just pointed out, treating parts of a product as imported derived objects means that you have no version control over the imported component's constituent versions. However, importing derived objects and treating them as tools dependencies can often be a useful thing. It makes your system model smaller, and it makes builds take less time. The more components that depend on the imported derived object, the more build time you save.

You should weigh the trade-offs involved when deciding whether or not to treat part of your product as an imported derived object. How much version control do you require over that piece of the product? The key to the decision is the piece's stability. If it's relatively stable and doesn't change often, you probably don't need to rebuild it frequently, and you can afford to not have version control of its constituent element versions in the main system. You could save yourself build time by treating it as a tools dependency rather than as part of the system.

Ask yourself whether you're going to need to use different versions of the part's constituent elements often. If your product is a compiler, the compiler's source code is integral to your product, and your builds should have version control over its constituent source code. If, however, your system merely uses a compiler, you don't need to control the compiler's constituent source versions. Not all "tools" dependencies are as obvious as this example of a compiler, but you can use the analogy to look at those parts of your system that you might treat as imported derived objects.

In Chapter 4 we discuss imported derived objects at greater length. See the highlight "Importing Derived Objects from Other Systems" for details.

Like libraries, system directories need no special storage considerations. If you choose to use the default system pool to hold any of your derived objects, make sure that the disk on which the system directory is stored has enough storage space to accommodate the greatest number of derived objects that might ever reside in the pool. This space requirement is determined by the default pool's parameters, the number of different components using the pool, and the sizes of the derived objects.

Highlight: The Consistency of DSEE Environments

The discussion of imported derived objects points out one of the attractive advantages of using DSEE environments to manage projects: consistency. Once you understand system model syntax, it becomes easy for you to grasp the structure of projects managed with DSEE facilities by simply reading their system models.

Suppose, for example, your system had a tools dependency on a compiler that was managed in a DSEE environment. If you really needed to find out about the source versions that went into a particular release of a compiler, you could do so with little more initial information than the pathname of the compiler's system model.

Pools

The CAD tools group uses one pool to store all of its derived objects. This pool has an age parameter of one hour and a limit of two versions. The limit is set below the default of four because one full build of the system consumes a very large amount of space.

Highlight: Using Multiple Physical and Logical Pools

The DSEE configuration manager gives you as much control over the way your derived objects are stored as you wish to take. If you want to ignore pools entirely, the configuration manager takes care of all aspects of pool storage for you, creating and configuring a default system pool. If you still want simplicity, but would like more control over where the configuration manager stores the results of your builds, you can do as the CAD tools group has done: create one physical pool, give it whatever parameters you choose, declare one logical pool in your system model that resolves to your physical pool, and use that pool to hold all of your derived objects. Your third option is to take advantage of all the flexibility that the configuration manager provides you for derived object schemes and use several different physical and/or logical pools.

The most effective use of different logical pools results from declaring each to be associated with a different set of physical pools. If all your logical pool declarations resolve to the same physical pool, you haven't gained much more than superficial organization.

When you use multiple logical pools associated with multiple physical pools, you can tailor the parameters of each pool to meet your requirements for different types of built components. For example, you could store your Model block's derived objects in a pool with a high limit parameter, so that you can access older builds for a long time. Subcomponents' derived objects could be stored in pools with smaller limits, in recognition of the infrequency with which you refer to older builds of these components. The case study presented in the next chapter demonstrates the use of different parameters for different pools.

The configuration manager also lets you assign a set of physical pools to one logical pool. It searches all the pools for an existing BCT to match a desired BCT; however, the manager only places new builds in the first physical pool listed (known as the **primary physical pool**). The next chapter's case study includes an example of a logical pool declaration that takes advantage of multiple physical pools and the pool search rules by following the primary physical pool with pools containing "read-only" derived objects.

Translation Rules

Before the CAD tools engineers wrote their system model, they built their system with shell scripts. The group had scripts that built individual components of the system (such as separate commands and utilities), scripts that built groups of components (such as a library of utility programs used by many commands), and a script to build the entire set of commands. (This last script was known as the master builder.) Many of the lowest level scripts performed the same functions: they compiled source code. Higher level scripts invoked these low-level scripts and then bound the results together.

When the engineers wrote their system model, they took advantage of the block structure of the model to streamline their translation rules. All of the scripts that compiled source code were rewritten as generic default translation rules. The engineers rewrote the higher-level scripts as translation rules that bound together the results of lower-level translations.

The **utilities** Aggregate is a good example of how the CAD tools engineers used the block structure when transposing their scripts to translation rules. Here is the text of the old script that built the library of utility procedures that many CAD tools commands use. The symbolic arguments (^1, ^2, and so on) represent options to compilers.

```
#***********UTILITIES BUILD SCRIPT***********
#
# call script to build the argument testing utility
//max/utilities/args_tester.bld ^1 ^2 ^3 ^4 ^5 ^6 ^7 ^8 ^9
# call script to build the banner utility
//max/utilities/banner.bld ^1 ^2 ^3 ^4 ^5 ^6 ^7 ^8 ^9
# call script to build the string equality utility
//max/utilities/equ_string.bld ^1 ^2 ^3 ^4 ^5 ^6 ^7 ^8 ^9
# call script to build the exclusive or utility
//max/utilities/exor.bld ^1 ^2 ^3 ^4 ^5 ^6 ^7 ^8 ^9
# call script to build the left justification utility
//max/utilities/left_just.bld ^1 ^2 ^3 ^4 ^5 ^6 ^7 ^8 ^9
# call script to build the rectangle overlap utility
//max/utilities/overlap_rect.bld ^1 ^2 ^3 ^4 ^5 ^6 ^7 ^8 ^9
# call script to build the quick sort utility
//max/utilities/qsort.bld ^1 ^2 ^3 ^4 ^5 ^6 ^7 ^8 ^9
# call script to build lower- to uppercase converter
//max/utilities/upper_case.bld ^1 ^2 ^3 ^4 ^5 ^6 ^7 ^8 ^9
# call script to build real number conversion utility
//max/utilities/val_real.bld ^1 ^2 ^3 ^4 ^5 ^6 ^7 ^8 ^9
# call script to build string writing utility
//max/utilities/writeshort.bld ^1 ^2 ^3 ^4 ^5 ^6 ^7 ^8 ^9
# call script to build response verfication utility
//max/utilities/verify.bld ^1 ^2 ^3 ^4 ^5 ^6 ^7 ^8 ^9
# call script to build binary tree management utility
//max/utilities/bin_tree.bld ^1 ^2 ^3 ^4 ^5 ^6 ^7 ^8 ^9
```

```
# now build the utilities.lbr file
//max/utilities/utilities.build_lbr
```

All but the last of the low-level scripts that the above script called simply compiled either FORTRAN or Pascal source code. Here, for example, are the contents of the **writeshort.bld** script:

```
pas //max/utilities/writeshort.pas @
 -b //max/utilities/writeshort @
 -l //max/utilities/writeshort ^1 ^2 ^3 ^4 ^5 ^6 ^7 ^8 ^9
```

In the CAD tools group's system model, the engineers eliminated all the separate translation rules for compilation by writing default translation rules:

```
default for ?*.pas =
  depends_tools
    '//max/cad/src/build/pas';
  translate
    //max/cad/src/build/pas %source @
      %cr_opt(-dba) %option(-dbs) %option(-comchk) @
      %option(-subchk) %option(-opt) %option(-nopt) @
      %option(-l, -l %result) -b %result
    %done;
  end of ?*.pas;
```

With most of the work done by the **utilities.bld** script taken over by the default translation rules, all that the **utilities** Aggregate's translation rule has to do is the translation formerly performed by **utilities.build_lbr**. Here is the text of the old script, which uses the Aegis librarian utility, **lbr**. (The librarian, like the Aegis binder, groups binary modules together.) We follow the old script with the new **utilities** Aggregate's translation rule.

The text of **utilities.build_lbr**:

```
IF EXISTF //max/utilities/utilities.lbr @
    THEN dlf //max/utilities/utilities.lbr ENDIF
lbr  -create //max/utilities/utilities.lbr          @
     //max/utilities/args.bin                       @
     //max/utilities/banner.bin                     @
     //max/utilities/equal_string.bin               @
     //max/utilities/exor.bin                       @
     //max/utilities/left_just.bin                  @
     //max/utilities/overlap_rect.bin               @
     //max/utilities/qsort.bin                      @
     //max/utilities/upper_case.bin                 @
     //max/utilities/val_real.bin                   @
     //max/utilities/verify.bin                     @
     //max/utilities/bin_tree.bin                   @
     //max/utilities/writeshort.bin
```

The text of the **utilities** Aggregate's translation rule:

```
translate
    //max/cad/src/build/lbr -create %result.lbr -<<!
    %result_of(?*.pas).bin
    %result_of(?*.ftn).bin
    !
 %done;
```

Note the use of the symbols **%result** and **%result_of** in the translation rule for **utilities** and the default translation rules for components whose names end with the **.pas** extension. These two symbols are at the heart of DSEE derived object management. Using **%result** and **%result_of**, you can refer to translator output without ever having to know its exact location and pathname; you ask the configuration manager to access the appropriate derived object for you. You give the configuration manager control of translator output with the **%result** symbol. Elsewhere in your system model, you access that controlled output with the **%result_of** symbol.

Highlight: Putting Derived Objects in Binary Pools

When the configuration manager executes a translation rule, it reserves space in the appropriate binary pool for the BCT and derived objects of the component. To ensure that derived objects are placed in this reserved space, you use the **%result** symbol in your translation rule. During the creation of the actual translation rule, the configuration manager replaces the **%result** symbol with the actual pathname of the space it reserved. This provides the translator with the information it needs to place the derived object in the binary pool.

(continued)

Highlight (continued)

For example, you might use the following translation rule to compile a Pascal source module and place its binary output in the binary pool:

```
translate
  //max/cad/src/build/pas %source -b %result
%done;
```

In this example, you use the **-b** option to identify the binary output. The **%result** symbol represents the configuration manager's predetermined space for the derived object in the pool.

It's important to understand that the translator places derived objects in the appropriate binary pools only if you use **%result**. For example, if we were to rewrite the above translation rule as follows:

```
translate
  //max/cad/src/build/pas %source
%done;
```

The compiler's binary output would be placed in a file in your working directory.

Most compilers, linkers, formatters, and other processing programs recognize command line options that allow you to specify the name of the file or files to which you want the output written. You use these options to send output to binary pools.

When you are working with processing programs that have no command line options for naming output, add one or more commands to the translation rule to move the output to the pool. Here is an example of this procedure.

```
translate
  /com/scribe %source
  /com/mvf %source({?*}.mss, @1, %leaf).err %result.err
  /com/mvf %source({?*}.mss, @1, %leaf).lpt %result.lpt
```

Assume the above is the translation rule for an Element named **book.mss**. When this component is built, the configuration manager moves two of the output files of the SCRIBE text formatting facility (a software product that runs on Apollo workstations) named **book.lpt** and **book.err** from your working directory into the appropriate pool.

In the default translation rule shown earlier in this section, **%result** takes the place of the pathnames of the binaries and listings that the compiler produces. The compiler automatically adds the **.bin** extension to the name of the binary output and the **.lst** extension to the name of the listing.

The **utilities** Aggregate's translation rule contains examples of a common use of the **%result_of** symbol. This translation rule makes symbolic reference to the results of translating **utilities'** result dependencies. Because the compiler automatically adds the **.bin** extension to the name of its binary output, the Aggregate's translation rule has to refer to the translator's binary output as **%result_of(?*.ftn).bin** and **%result_of(?*.pas).bin**.

The symbols **%result** and **%result_of** serve the same purpose that variables do in any program: they free you from the need to know the true value of an expression. When you use **%result_of** to refer to the result of translating a component, the configuration manager knows what pathname to substitute for the symbol.

Note, for instance, that **utilities'** translation rule doesn't include an Aegis **dlf** command to delete former builds of the Aggregate (as the **utilities.build_lbr** script does). That's because the DSEE configuration manager assigns a unique name to each build of the Aggregate. There's never any ambiguity about which build produced a given derived object, since you let the configuration manager name and access the derived object for you.

You can see the actual pathnames with which the configuration manager replaces your symbols as your translation rules are executed. Include the **-von** option in your **build** command.

Another interesting aspect of the **utilities** Aggregate's translation rule is the use of wildcard expansion in the rule's "here document" (that is, the text between the characters **-<<!** and **!**). In our example, the CAD tools engineers eliminated their original script's list of utility procedures in the new translation rule by taking advantage of the system model compiler's recognition of wildcards. As a result, adding a utility procedure to the library of utilities requires only the appropriate reference to the procedure as a dependency of the Aggregate—the translation rule doesn't have to be edited.

Highlight: How Many Components Do You Need?

In general, you should write your system model so that, once default translation declarations are resolved, there is a one-to-one correspondence between components and translations producing significant output. This allows you to modularize your system and maximize the amount of reuse you can get out of derived objects.

Assume, for example, that you're debugging a module of your system. You edit the module's source code, which is in the DSEE element **design.pas**, try to compile it, re-edit it to correct the compilation errors, and so on until the module compiles. After that, you test your changes by running the program, re-editing and recompiling as necessary.

For expediency, you do your compilations by building the lowest-level component of the system model that depends directly on **design.pas**. The more work that that component's translation rule does, the longer your builds take—and the fewer opportunities exist for parallel building. If, for instance, the lowest-level buildable component depending on **design.pas** is an Aggregate whose translation rule compiles the source code of seven other Pascal modules before recompiling **design.pas**, you have to wait until all seven modules are recompiled before you find out the compilation errors in **design.pas**. The configuration manager can't execute only a segment of the Aggregate's translation rule. Also, you can't declare equivalences for the compilations of the other modules, since they, like **design.pas**, aren't buildable components. Your development work would go much more quickly if the lowest-level system component depending on **design.pas** were an Element with a primary source dependency on **design.pas**.

In short, your system model should be as modular as possible. At its lowest level, most of its components should be Elements and Externals.

Dependencies

Declaring your system's dependencies is a fairly straightforward procedure. If you have been using build procedures that identify include files (as do makefiles, which are recognized by the UNIX system **make** facility), you can use these lists of dependencies when constructing your system model.

If you don't have any such list of dependencies, you can run the **make_model** utility program that we supply with the DSEE software to identify all of your modules' dependencies. This program searches the text of your source code modules for include statements. For each module, it builds a rough version of a system model component declaration, listing include files as source dependencies. The **make_model** utility distinguishes between elements and ordinary files, associating library names with elements and resolving full pathnames for links.

Once you've run **make_model** for all of your modules, you should review the lists of dependencies and make any necessary changes. For example, you might want to eliminate references to an include file depended on by all modules from the individual module declarations and write a default **depends_source** declaration for that particular dependency.

Sandy, one of the CAD tools group engineers responsible for writing the system model, used **make_model** to determine the system's dependencies. Once she'd finished, she edited the resulting text as follows:

- She wrote a default library declaration for all of the include files and deleted the "*@ library_name*" declarations that **make_model** placed after the names of source dependencies that were DSEE elements
- She added square brackets to the names of noncritical dependencies
- She removed the link resolutions that **make_model** had inserted in the names of dependencies that weren't DSEE elements, because she wanted to continue to refer to the files through links

Highlight: Listing a Tool as a Dependency

It's always a good idea to declare your translators to be tools dependencies, even though the configuration manager doesn't require you to do so. Listing a translator in a **depends_tools** declaration causes the configuration manager to record the version stamp of the tool in the BCTs. This information is valuable when you are trying to reconstruct an older build. If, for example, you received a bug report on a module of your system written in FORTRAN, it could be very helpful to know which version of the FORTRAN compiler generated the executable code.

If you don't want the configuration manager to rebuild a component every time its translator changes, you can make the tool a noncritical dependency of the component by enclosing the name of the translator in square brackets in the **depends_tools** declaration.

Using Built Include Files

Several of the source components of the CAD tools group's system model contain include statements that refer to the output of translating other source modules. These references to built (or preprocessed) include files have to be constructed using special syntax, since you can't know the exact pool pathname that the configuration manager will assign to the product of a build.

For example, the system model contains an Element named **cad.sch**. One of the derived objects of **cad.sch** is a preprocessed include file that describes the design database. This preprocessed include file is a dependency of the Element **design_database_declare.ins.pas**.

Below is a portion of the system model's declaration for **cad.sch**. The example contains a **make_visible** declaration, which tells the configuration manager to make the results of translating this element temporarily visible to other source code during the build. The example also contains the portion of the translation rule that produces the include file describing the design database.

```
element cad.sch  @ database_lib =
  make_visible;
    .
    .
  translate
    .
    .
    /com/chpat -o <%source >%result'des_define.base' @
      "%% *RECORD *= *{[-.]*}.?*" @
      '@1 @: Define db_$Record_type_identifier;'
    /com/chpat -o <%source >>%result'des_define.base' @
      "%% *FIELD *= *{[-.]*}.?*" @
      '@1 @: Define db_$Field_Type_identifier;'
    /com/chpat -o <%source >>%result'des_define.base' @
      "%% *SET *= *{[-.]*}.?*" @
      '@1 @: Define db_$SET_Type_identifier;'
    .
    .
    %done;
```

Later in the system model is the declaration of the Element whose primary source contains an include statement referencing the preprocessed include file:

```
element design_database_define.ins.pas =
      .
      .
    promote_depends;
    depends_result
      cad.sch;
end of design_database_define.ins.pas;
```

Naturally, the source code of **design_database_define.ins.pas** has to contain an include directive referencing the preprocessed include file so that the compiler can read the preprocessed file in the compilation. This include directive uses a special syntax, as shown below, to stand in for the actual name of the preprocessed file. This syntax is parallel to the syntax you use when referencing the result of building a component within your system model: the **%result_of** symbol, followed by any extension that the translator appended to the output file.

```
%INCLUDE '$(cad.sch)des_define.base';
```

The **make_visible** declaration tells the configuration manager to assign the pathname **%result_of(cad.sch)** to an environment variable named **cad.sch**. The include directive contains the appropriate syntax for dereferencing the environment variable, thus resolving the pathname.

Highlight: Nested Include Files

The declaration of **design_database_define.ins.pas** that we discuss above is a good example of how you handle nested include files in your system model. The Element **design_database_define.ins.pas** is, itself, an include file on which a number of system components depend. To ensure that the configuration manager recognizes that the include files referred to by **design_database_define.ins.pas** are direct dependencies of any system components that have include dependencies on **design_database_define.ins.pas**, the CAD tools group engineers have included a **promote_depends** statement in the Element's declaration. This forces the configuration manager to rebuild all the dependent system components whenever **cad.sch** changes.

Undeclared Include Dependencies

If you neglect to declare an include dependency in the system model, the configuration manager still builds your system. However, the system's BCT contains no information on the version or time stamp of the source dependency, making it difficult to trace an error in the dependency back to the original source code. Also, the configuration manager won't rebuild components that depend on the unlisted source dependency when someone changes that dependency. Finally, omitting a source dependency declaration for a DSEE element from your system model means that you will never be able to use any but the most recent version on the main line of descent of the element in your builds. Version specifications in the configuration thread apply only to elements declared as dependencies in the system model.

The configuration manager will identify your undeclared dependencies on DSEE elements for you. It generates warning messages if it encounters undeclared dependencies on DSEE elements while building your system. Later, you declare these dependencies in your system model. (Note that this only happens for elements in libraries listed in your **library** declaration.)

Working in the DSEE Environment

Because the focus of this chapter is the process of establishing a DSEE environment, we won't spend much time discussing how the CAD tools group uses the environment they have created. Their use of DSEE facilities is fairly straightforward.

Therefore, we devote this section to one aspect of the CAD tools group's use of their DSEE environment: how they create and manipulate releases.

Releasing the Product

The ability to use the DSEE release manager was one of the CAD tools group's principal reasons for converting to a DSEE environment. Before they started using DSEE facilities to control their builds and releases, the engineers spent a substantial amount of time backtracking errors in their own production system. The group occasionally produced releases built with incorrect versions of source files and had trouble tracing bugs back to the constituent source code.

The configuration and release managers eliminated many of the errors involved with the CAD tools group's production scheme. The configuration manager provides a degree of control over build specifications that the group never had before. Creating a system build for release became a matter of writing the correct thread.

How the CAD Tools Group Creates a Product

The CAD tools group maintains a file that contains the names of all the buildable components that make up a product release. When the engineers are ready to generate a build that they want to distribute, they update this file and name it **cad_release_files_***release*, where *release* is the name of the release.

The updated list of buildable components becomes the argument to the **-export** clause of the **create release** command. For example, when the CAD tools group generated Revision 5.1, they edited **cad_release_files_REV5.1** to refer to all the buildable components of their system and then created a release with the following command:

```
DSEE> cre rel ~/src/rel_REV5.1 -from cad!16-Dec-1985.13:24:38
     -export */cad/src/build/cad_release_files_REV5.1
```

(Of course, the entire command must actually be entered on one line.)

The release manager creates a separate directory in the release area for each built component listed in **cad_release_files_REV5.1**. However, this isn't the structure that the CAD tools group wants to use for its distribution. Therefore, the engineers execute a script called **create_cad_release.ash** that copies the contents of the new release area into a directory tree with the appropriate structure for distribution. Below is the text of **create_cad_release.ash**.

```
eon
IF EQS ^2 THEN
# Get the name of the DSEE release directory
# and the name of the directory to create

readln -prompt ' Enter release directory name: ' release_dir
readln -prompt ' Enter directory for CAD release: ' cad_dir
ELSE
  release_dir := ^1
  cad_dir := ^2
ENDIF
/com/args "Creating ^cad_dir"
/com/crd ^cad_dir
#
# Get copies of object files, shell scripts and help files
# from this DSEE release directory (files will be copied
# from the EXPORTS directory into COM and HELP directories)
#
```

```
/com/crd ^cad_dir/com
/com/args "Creating ^cad_dir/com"
/com/crd ^cad_dir/help
/com/args "Creating ^cad_dir/help"
/com/ld ^release_dir/exports -ld -c -nwarn -nhd  | @
WHILE READLN command DO
ARGS "working with ^command"
  IF EXISTF  ^release_dir/exports/^command/^command THEN
      #
      # get the CAD objects
      #
      /com/args "  OBJ file"
      /com/cpf ^release_dir/exports/^command/^command ^cad_dir/
com -lf

ELSE
     #
     # get CAD HELP files
     #
     IF EXISTF ^release_dir/exports/^command/?*.hlp THEN
         /com/args "  .HLP file"
         /com/cpf ^release_dir/exports/^command/?*.hlp
^cad_dir/help -lf

       ELSE
       #
       # get CAD shell scripts
       #
       script := ' '
       /com/args "cad script or no file"
       /com/ld -c -nwarn -nhd ^release_dir/exports/^command @
          | /com/fpat [.] -x | READLN script
       /com/cpf ^release_dir/exports/^command/^script @
          ^cad_dir/com -lf >?/dev/null
    ENDIF  # .hlp file or shell script
  ENDIF  # object file
ENDDO # all files exported
```

Highlight: Should I Store Configuration Threads as Elements?

Configuration threads, like any text files, can be stored as DSEE elements. However, if you rely principally on dynamic configuration thread rules to build your system, you probably won't want to reuse older versions of configuration threads very often—in particular, you can't re-create older builds with them.

(continued)

Highlight (continued)

Suppose, for example, the CAD tools engineers used the following thread when they generated Revision 5.1:

```
.../REV5.1 -when_exists
[]
```

Later, they need to create another release of their product to correct several bugs in this earlier distribution. They want the new build to contain the most recent versions of elements with branches with the leaf name **REV5.1_fixes**. For all other elements, they want the new build to replicate the Revision 5.1 build.

Editing the old configuration thread by adding a new rule to refer to the bug fix branch is highly unlikely to produce a configuration thread that builds the desired configuration. As you can see, the old thread contains two dynamic rules referencing the most recent versions on two lines of descent of elements. Unless no one has created a new version on either line of descent since the Revision 5.1 build, a new build with the edited old thread won't re-create the older configuration.

The thread that does build the desired configuration is one that makes specific reference to the BCT of the earlier build using a build-ID-based rule:

```
.../REV5.1_fixes -when_exists
cad!//max/cad/src/rel_REV5.1 -versions -options -exact
```

In summary, configuration threads containing dynamic rules refer to the current state of the DSEE environment. When you change the state of the environment (as you might by creating a new version of an element), you may alter the outcome of a build, even though you've changed neither your system model nor your configuration thread.

Once the engineers have created their distributable directory structure, they perform a couple of steps to "tidy up." First, they give all the versions of elements used in the release the name of the release; for example, after releasing Revision 5.1, the engineers executed the following command:

```
DSEE> name version cad!//max/cad/src/rel_REV5.1 REV5.1
```

Not only does this command assign the name **[REV5.1]** to all element versions used in the system build; it also assigns the name to all versions of system model elements (the root model and any model fragments) used to construct the build.

To signal the completion of one release and the beginning of work on the next release, the engineers reserve, edit, and replace the element **banner.ins.pas**, which resides in the include files library. The text of this element changes so that the banner identifying the release of the product (which users of the product can access by executing the **banner** command) displays the name of the next release (in our example, Revision 5.2).

Highlight: Relating a Released Product Back to Its Constituent Source Versions

The CAD tools group's **banner** command demonstrates an easy, effective way to associate a released product with the element versions that constitute the product. This association is particularly useful when users report problems with the product. Because the released product and its BCT are stored in a DSEE release area, maintaining engineers can pinpoint the constituent source versions of a problematical release quickly by looking up the DSEE build maps of the released build.

As we mention above, the engineers place the name of the release that they're working on in the text of the include file for the banner. Every subsequent build of the system has the release name incorporated into it. This same release name is used to name the release area containing the build that the group distributes to users. As a result, an engineer supporting releases of the system has only to ask a customer reporting a bug to type the **banner** command to find out what version of the product is causing problems.

In fact, the CAD tools engineers take another measure to ensure easy tracing of a bug to the source code that caused it: they name all element versions used in the build for the release. This is a useful supplement to the technique of naming the release area after the release reported by the **banner** command.

Chapter 3

Case Study 2: Developing a Multi-Targeted Operating System

The subject of this chapter is Apollo's operating system (OS) development project. This project illustrates an application of the DSEE product that uses a single source file to represent several DSEE system models and systems to produce software products that run on several different hardware configurations. However, in the course of their work, the project's personnel encounter difficulties that are common to many development projects.

As we did with our previous case study, we will present the group and its goals, the DSEE objects they use (for example, their libraries, systems, and system models), and the way they work in the DSEE environment. Our discussion will focus on how the OS engineers use DSEE facilities to implement their multi-target development. For example, we will examine the branches they create, when and why they create releases, and how they coordinate simultaneous subprojects.

As you read this chapter, you may want to refer to Appendix B. This appendix contains the text of an abbreviated version of the OS group's system models.

Introduction

The OS project is a large-scale engineering project. The OS group's work focuses specifically on the operating system. Other groups handle local area and heterogeneous networking, integrating layered products, and other operating system-related activities.

Separate Products that Share Source Code

The OS group uses one set of sources to produce different executable operating systems for different hardware configurations. When they started to use the DSEE configuration manager, the engineers had to find an effective representation of a structure in which common source code produces multiple products. The engineers considered possible representations of their system carefully. Nat, the principal DSEE environment administrator for the group, examined the alternatives and devised a satisfactory configuration.

Representing the Software with Multiple Systems

Nat opted to represent the OS software with several separate systems. This configuration had several distinct advantages that appealed to the group.

The first advantage lay in the organization of the software as it is distributed. A DSEE release area is associated with the system from which its contents are released. The OS group wanted to tie together various releases of one product. Therefore, it was natural to have a separate system for each target machine. Using this organization, an engineer could see all the releases of one specific product by issuing the **show releases** command after setting the system to the one associated with the particular product. (Figure 3–1 illustrates this point.)

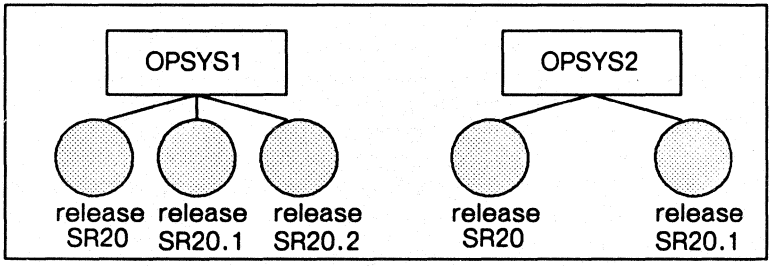


Figure 3–1. Releases Associated with Different Products

The second advantage of using multiple systems involves the performance of system model and configuration thread validation. The configuration manager caches a limited number of validated system models and configuration threads for each system. Whenever a user sets the current system model or current configuration thread, the configuration manager checks to see whether the new setting is one whose validated version is available. If so, the configuration manager doesn't need to validate the model or thread, and the configuration manager sets the model or thread relatively quickly.

For many software systems, the configuration manager's cache of validated system models and configuration threads is large enough to be very useful. However, the OS engineers knew that they would be using many different model and thread settings. The engineers knew they ran the risk of wanting more validated models and configuration threads than one system could hold. Each target machine would require at least one different model setting; realistically, one system couldn't be expected to hold any significant variety of the model and thread settings that all the OS engineers would require. Having separate systems meant that they could have separate caches for their validated model and threads.

Because each product is built with a unique configuration, however, the OS engineers have to consider all the affected systems. They have to ensure that their changes are tested for all systems, and they want to reduce the rebuild impact for others. Suppose, for example, that an OS engineer changes an include file that's used in builds of the machine-specific aspects of two systems. When building one of the affected systems, the engineer rebuilds all the affected components and declares equivalences for those components that don't need the newer version of the include file. Since the other affected system's BCT might be quite different from the first's, though, the configuration manager might not be able to reuse any of the builds or equivalences that the engineer created when building the first system.

Building the Software with One System Model

Once Nat decided to create one system directory for each product, he had to decide which it was better to have:

- A separate system model source file for each product
- One system model source file shared by all the products, with embedded conditional directives differentiating between products

Using multiple system model source files was not attractive to Nat because of the maintenance problems such a configuration would cause. Much of the operating system is identical for all supported hardware configurations. Each has the same elements and dependencies—only their translation rules differ. Therefore, a large portion of each of the different system model source files would have to be identical. Altering any of the modules the systems shared (for example, listing a new include file as a source dependency) would require identical edits to every copy of the module declaration.

Therefore, Nat wrote one source file to provide a system model for all of the group's systems. Nat used conditional variables to control the model declarations and the resolution of **alias** declarations. Using aliases in this manner lets an engineer use different pools, systems, translators, and other machine-dependent objects without having to specify these different objects each time they're used in the model. Thus, many differences between target models are isolated in one area of the text, which makes it easy to maintain them.

To set the current model, an OS engineer sets the current model thread to one that includes a **-target** rule that appropriately establishes the variable setting for the target operating system. For example, the model thread might have the following text:

```
-target OPSYS1
-reserved
[]
```

A subsequent **set model** command causes the configuration manager to validate the system model. The argument to the **-target** rule establishes variables that are referred to by conditional directives in the model. For OS engineers, the variable argument to the **-target** option determines which set of **alias** declarations will be expanded in the model. The results are target-specific declarations for the system the user is building and the storage pools for the machine-dependent derived objects, and target-specific parameters to pass to the translation script **make_build_time**.

Below are some excerpts from a simplified version of the OS group's system model source file, one that produces only two different products. The excerpts illustrate the use of conditional directives in an **alias** declaration. Notice that, in some instances, aliases are expanded *within* the declarations of other aliases. Aliases enable you to isolate parts of a system model that might vary, and nested aliases provide even further isolation of variable parts.

```
%var OPSYS1 OPSYS2

%if not (OPSYS1 or OPSYS2) %then
  %error 'You must give a system name after @
    -TARGET on the SET MODEL command'
  %exit
%endif

%if OPSYS1 %then
  model operating_system1 =
  alias
    os = '1';
    asmname = 'asm';
    asmoptnuc = '-ndb -config os';
    asmoptker = '-ndb -config os apollo_%exp(os)';
    nucbin = 'bin';
      .
      .
```

```
%elseif OPSYS2 %then
  model operating_system2 =
  alias
    os = '2';
    asmname = 'asm';
    asmoptnuc = '-ndb -config os';
    asmoptker = '-ndb -config os apollo_%exp(os)';
    kerbin = 'bin%exp(os)';
      .
      .

%endif
title
  'Operating system %exp(os)';
system
  '//opera/op_sys/op_sys%exp(os)';

pool
  opsys_pool =
    '//opera/op_sys/pools/sr20.bl002/opsys%exp(os);
  nuc_pool =
    '//opera/op_sys/pools/sr20.bl002/bin',
    '//opera/op_sys/pools/sr20.bl001/bin';
  ker_pool =
    '//opera/op_sys/pools/sr20.bl002/bin%exp(os)',
    '//opera/op_sys/pools/sr20.bl001/bin%exp(os)';
      .
      .
```

One advantage of the conditional structuring of the system model **alias** declarations for the OS group is that it is parallel to the way that much of the group's machine-dependent software is written. Most of the machine-dependent modules are used to construct every product. Embedded in each one of these modules are conditional directives that control what aspects of the module are translated for each particular machine. A logical development of the structure used in the code led to making the compilation of the system model source file conditional on the same variables used in the source code.

Working in a Multi-Target Environment

The engineers' activities fall into three general categories: incorporating major enhancements into the operating system; adding support for new kinds of workstations; and performing maintenance work, such as responding to Apollo Product Reports (APRs).

The OS engineers work on project teams. Each major enhancement or new workstation is an OS group project. All engineers do maintenance work. An APR is generally assigned to the person or persons responsible for the related aspect of the operating system.

Presently, there are many projects going on in the OS group. Some involve adding support for new hardware configurations. Others involve upgrades for new communications and graphics hardware and software, enhancements for better support of layered products, important fixes to bugs in the released operating system, and similar activities.

Some projects, like the additional support for new hardware configurations, require changes to only one system. Others, like upgrades for new hardware, may involve changes to several, but not all, of the systems. A few projects (for example, enhancements in support of layered software) require modifications that affect all systems.

Project coordination is critical to the OS group. Each project must isolate itself from simultaneous development work. However, each engineer working on a project needs to be able to see the work of other engineers on the same project as it evolves. As work on a project progresses, its engineers need to integrate and test their work in each system. Finally, all isolated projects must integrate their work with all other projects when development and testing are complete.

Achieving this level of coordination requires careful use of DSEE tools. The OS engineers must consider carefully the configuration threads they use as well as the equivalences they declare and share. They must agree to work protocols, such as developing naming conventions for branches and determining the individual's degree of responsibility to the group.

Much of this chapter focuses on how the OS developers work on projects. In particular, we examine such aspects of the engineers' work as:

- How their working directory setting impacts the building process, and what techniques they use to minimize some of the associated problems
- How they declare equivalences within the context of multiple systems
- What DSEE command scripts they use to facilitate their work
- How they use releases to create an environment for use by engineers who are not using the DSEE configuration manager

Project Structure

This section presents the DSEE objects that the OS group uses to do their work. Here we show a static picture of the OS group's environment, explaining why they chose certain aspects of the environment. In the following section, we illustrate how they use the DSEE objects presented here to perform their work.

Libraries and Elements

The OS group maintains five DSEE libraries. Table 3–1 contains the library names and summaries of their contents.

All machine-independent source files are stored in the library **//opera/op_sys/nuc**. These files and their associated derived objects are known collectively as the "nucleus" of the operating systems.

The OS group maintains all of its machine-dependent source code in one library (**//opera/op_sys/ker**). Each system requires the same machine-dependent source files (collectively referred to as the "kernel"). However, the machine-dependent code's compilation or assembly differs with the machine type. Conditional compilation directives determine which sections of each machine-dependent module are compiled in a particular build.

Two other libraries hold the include files used by the source modules. The machine-independent modules' include files are stored in **//opera/op_sys/ins**, and the machine-dependent modules' include files are stored in **//opera/op_sys/kins**.

Table 3-1. OS Group Libraries and Their Contents

Library Name	Contents
//opera/op_sys/nuc	Source files for machine-independent operating system modules
//opera/op_sys/ker	Source files for machine-dependent operating system modules
//opera/op_sys/ins	Include files for machine-independent operating system modules
//opera/op_sys/kins	Include files for machine-dependent operating system modules
//opera/op_sys/scripts	Scripts and the operating system's DSEE system model (root and fragments)

The OS group's fifth library, **//opera/op_sys/scripts**, contains the project's element that represents the system models and various DSEE scripts written by members of the OS group to facilitate their work. We discuss the elements containing the system models in "Systems and System Models," below. We discuss the scripts that the OS group uses in the section on "Working in the DSEE Environment," later in this chapter.

> *Highlight: Where to Store Machine-Dependent Source Code*
>
> The OS group found it convenient to store all their machine-dependent code in one source library, since sections of the sources, rather than entire sources, were machine-dependent. (These sections contained embedded conditional constructs that determined which text would be compiled during a build.) However, if each operating system used some source files that none of the other operating systems used, it might have been more appropriate to store the source files relevant to each operating system in a separate library.
>
> Compiler development is an example of a multi-target engineering project that might use a different library configuration. Compiler development frequently involves sharing source code for a code generator among many compilers. Each compiler has a set of source files distinct from those used by other compilers. In this case, it makes more sense to store the source code unique to each compiler in a separate library, rather than pooling it in one library, as the OS group has done.
>
> Neither storage scheme has a significant effect on build performance.

Tasks, Tasklists, and Monitors

The group's use of tasks, tasklists, and monitors is minimal. At present, they have set only one monitor. This monitor was set by a project team leader working on a project known as Phantom. The project team leader set a monitor on the element containing the system models. The monitor sends him mail whenever someone creates a new version of the system model. The mail helps the project leader keep abreast of the evolution of the system model element.

Systems and System Models

As mentioned earlier, the OS group uses one system model source file containing conditional directives to build all of the operating systems. In our scaled-down version of the operating system's situation, there are two OS systems, both of which are subdirectories of **//opera/op_sys: op_sys1** and **op_sys2**.

Each system model is flat; that is, it contains no Aggregate blocks. Each model consists of many Elements that are translated individually and then bound together by the Model block's translation rule.

A representation of the system model source file appears in Appendix B. Here we note a few significant aspects of its text.

Highlight: Why Should a System Model Be Stored as a DSEE Element?

While it's possible to store system models as regular files rather than as DSEE elements, there are several good reasons not to.

If your system model uses the **%include** system model language directive to incorporate model fragments, storing both the root model and the fragments as DSEE elements gives you all the same advantages as storing your source code modules as elements. Your model thread can identify particular versions of the root model and model fragments, on both the main line of descent and branches. If you store your model and fragments as ordinary files, you can't take advantage of the version specification available with model threads.

Related to this benefit is being able to use older versions of the system model to replicate past builds. Unless you store the copy of all the system model files used for a particular build (both the root model and model fragments), it's impossible to re-create an old build if the system model changes substantially.

(continued)

Highlight (continued)

Also, a system model stored as a regular file has no history associated with it. If you store a system model as a DSEE element, you get the benefit of history management for the system model. In a complex development environment like that of the OS project, the system model changes frequently. New dependencies and new modules are added often. Being able to monitor the evolution of the system model is as valuable as being able to monitor the evolution of source code.

Another advantage of storing the system model in a DSEE library is the ability to create alternate lines of descent for the system model. Working with a branch version of the system model allows engineers on a subproject to modify parts of the model to their advantage and develop their own system without disrupting anyone else's work.

For example, the engineers working on one subproject needed to make modifications to a few OS include files. These include files are listed in the system model as non-critical dependencies, because the OS developers don't want to have to rebuild for most changes to the files (for example, such changes as adding more procedures). However, the engineers on this subproject were changing the procedures in the include files, and they wanted the configuration manager to rebuild because of the changes.

The subproject's engineers created a branch off of the system model's main line of descent. In the first version on the branch, the engineers made the include files critical dependencies. This change allowed the engineers to test and debug the changes they were making to the include files. After they finished debugging the include files, the engineers declared the system model branch obsolete.

Pools

Regardless of the system being built, its model has three logical pools: **opsys_pool**, **nuc_pool**, and **ker_pool**. **Nuc_pool** and **ker_pool** are used for the machine-independent and machine-dependent derived objects, respectively. The remaining pool, **opsys_pool**, is used to hold the derived object of building the Model block.

The following fragment of the system model element shows the pool definitions.

```
pool
  opsys_pool =
    '//opera/op_sys/pools/sr20.bl002/opsys%exp(os)';
  nuc_pool =
    '//opera/op_sys/pools/sr20.bl002/bin',
    '//opera/op_sys/pools/sr20.bl001/bin';

  ker_pool =
    '//opera/op_sys/pools/sr20.bl002/bin%exp(os)',
    '//opera/op_sys/pools/sr20.bl001/bin%exp(os)';
```

The OS group's pool declarations allow every system to share the derived objects stored in **nuc_pool**. This sharing avoids unnecessary rebuilding, because the derived objects this pool holds are built from the same translation rules and are not specific to one hardware configuration.

Because the translation rules for each system differ, the machine-dependent derived objects stored in **ker_pool** cannot be shared among systems. Therefore, **ker_pool**'s declaration differs for each system (the expansion of the **os** alias points the configuration manager toward a different pool for each target). The derived objects unique to one system do not have to compete for space in the pool with another system's derived objects.

Highlight: Reusing Parts of Builds

When you issue the **build** command, the DSEE configuration manager searches the physical pools listed in your system model's logical pool declaration for a BCT that matches the desired BCT. If the configuration manager finds a match, it reuses the derived objects associated with the existing BCT rather than rebuilding the component.

The OS group's system model source file takes full advantage of the configuration manager's pool search method. The engineers reuse the derived objects of an earlier base level simply by referring to a pool containing the older derived objects.

The declarations of both **nuc_pool** and **ker_pool** employ this technique. Here is the declaration of **nuc_pool**.

```
nuc_pool =
  "//opera/op_sys/pools/sr20.bl002/bin",
  "//opera/op_sys/pools/sr20.bl001/bin";
```

The second physical pool, **sr20.bl001/bin**, was the sole physical pool listed in the declaration of **nuc_pool** in the version of the system model that produced Software Release 20, Base Level 001. Then one of the OS engineers created a new version of the system model element, inserting the name of physical pool **sr20.bl002/bin** as the first pool in **nuc_pool**'s declaration.

When the configuration manager builds the operating system with this version of the model, it searches both pools, sequentially, for a reusable BCT for each machine-independent component. If none exists, the configuration manager builds the component and places it in the first pool listed, **sr20.bl002/bin**. The second pool, **sr20.bl001/bin**, is searched during a build, but new builds are never inserted into it.

(continued)

Highlight (continued)

This technique is most helpful for engineering groups that, like the OS group, perform much more development work than maintenance work. Groups that more evenly divide their work between development and maintenance can keep all of their derived objects in one physical pool, because the configuration manager employs a least-recently-used (LRU) algorithm when removing derived objects from the pool. Such a group's maintenance work ensures frequent reuse of the derived objects from the build being maintained, keeping them in the pool. Because the OS group does more development than maintenance work, it is likely that the derived objects from Base Level 001 would be eliminated from the pool if they had to compete for space in a pool with derived objects of the same components produced for development work.

Note that including the Base Level 001 pool in the logical pool declaration for **nuc_pool** is only useful to the OS group as long as the builds for Base Level 001 are reusable for Base Level 002 builds. When development on Base Level 002 progresses to the point where Base Level 001 builds aren't reusable, the OS group engineers might want to remove the Base Level 001 pool from **nuc_pool**'s declaration. While it causes no confusion to the configuration manager to have the Base Level 001 pool in the declaration, it does needlessly increase the length of time that the configuration manager spends searching for reusable builds.

The **opsys_pool** is used to hold the Model block's derived objects. The logical pool declaration of **opsys_pool** contains an expansion of an **alias**, which causes the configuration manager to use a different pool for each target. Although the Model block builds for each target could coexist in the same pool without forcing each other out (because each target operating system's Model block has a different name, dependent on the **-target** rule in the model thread used), the OS group chose to create a different pool for each target's Model level build. This enables the engineers to set different age and limit parameters on each of these physical pools, acknowledging the varying usefulness in rebuilds of older builds for each target.

Each pool's parameters have been set according to the pool's use. All pools appearing as the first of two physical pools in declarations have limit parameters of five versions and age parameters of 24 hours. The limit of five takes into account the number of concurrent development activities that are likely to take place. The engineers set the age parameter to 24 hours because they do not want the configuration manager to purge builds less than one day old from the pool.

Pools appearing as the second of two physical pools in logical pool declarations have limit parameters of one version and age parameters of one hour. These parameters recognize that builds never add derived objects to these pools.

Each **opsys_pool** (that is, the physical pools pointed to by the expansions of **sr20.bl002/opsys%exp(os)**) has a different set of parameters, depending on the requirements for the particular target operating system. The physical pool that holds the builds for operating system 1, **sr20.bl002/opsys1**, has a limit parameter of 10 versions and an age parameter of 24 hours. The OS engineers have set the limit parameter this high because they do much experimentation and testing on OPSYS1 and they know from experience that an older build might form a good basis forfuture experimental work. If they write a thread that refers to the older build in a build-ID-based rule, the configuration manager won't have to rebuild the older configuration unless it has been purged from the pool. The high limit parameter ensures that the older build they want to use is likely to still be in the pool, despite the use of the pool by all engineers.

The physical pool that holds the builds for operating system 2, on the other hand, has a much lower limit parameter: only two builds over 24 hours old are permitted to live in the pool. This lower limit recognizes the small likelihood that an older build of this operating system will be reusable for a later build.

Highlight: Pool Storage and Access

To look at the OS system model text, you would assume that the binary pools, like the libraries, all reside on the node **//opera**. This is not really the case. The OS group's pools are very large, as are their libraries. If they coexisted on one node, builds would fail because the pools would exceed the available disk space. Moreover, the input/output load would be too high on a single node containing all of the group's libraries and pools.

In reality, all the physical pool pathnames in the system model are links to real pool locations on other nodes. By referring to the pools through links, the engineers aren't obliged to change the system model whenever it becomes necessary to move pools to larger disks. The configuration manager keeps track of link resolutions and notifies you that the system model must be revalidated whenever the link points to a new location.

Translation Rules

The OS group uses several scripts in their translation rules. For example, the Model block's translation rule calls the script **make_build_time**, passing it some parameters (including a value determined by the system being built). This script places a time stamp in the derived objects produced by building the model.

The time stamp is useful for tracing problems detected in released software back to the constituent source code. When a customer detects a problem with the operating system, he or she relays the time stamp of the operating system (determined by issuing a **/com/bldt** shell command) to Customer Service along with the description of the problem. The OS engineers can then match the time stamp of the customer's software against the time stamps of the releases of the operating system maintained in release areas. When they locate the released system with the same time stamp as the one the customer is using, they issue an **examine build** command to discover which versions of the source elements were used in that build.

Here is a portion of the Model block's translation rule.

```
translate
    von; eon; abtsev -p

    //opera/op_sys/scripts/make_build_time %exp(os) @
                                %result.bldt.ins.pas @
                                %result.bldt.asm @
                                %result.pbldt.bin @
                                %result.bldt.bin @

%done;
```

Here is the text of that script.

```
voff
# usage: make_build_time <mach id>
#       <print_build_time ins pathname>
#       <build_time source pathname>
#       <build_time obj pathname>
#       <print_build_time obj pathname>
#       [-build_string <Build string>]
#

# This script is invoked by the DSEE OS system model to
# assign a build time to the operating system build.

#
# Make "bldt.ins.pas" -- contains the "bldt"
# string.
#
# The "-build_string" option can be used to
# override the header string.
#

bstring := "OPSYS 20.0+"

/com/date | /com/chpat "{?*} (?[SD]T)" @
  "CONST build_time = ''^bstring, @1.%.';" >^2

#
# Compile "print_build_time.pas".  Note that
# we create a link temporarily because of the
# way the insert file is referred to.
#

/com/crl -r build_time^1.ins.pas ^2
//opera/op_sys/pas //opera/op_sys/ker/print_build_time @
  -b ^4 -ndb -idir //opera/op_sys -config os apollo_^1
/com/dll build_time^1.ins.pas
```

```
#
# Make "bldt.asm" -- contains the
# definition of "build_$time".
#


/com/catf >^3 <<!
 module  build_time,WIRED_PROC,WIRED_DATA
 entry.d build_$time
!
/com/ld -u -nhd ^2 | @
 /com/chpat "%{?????????}.?* ?*" "build_$time @
   dc.l    $@1" >>^3
   /com/catf >>^3 <<!
     end
     !


#
# Assemble "build_time.asm".
#


//opera/op_sys/asm ^3 -b ^5 -nl
```

The OS group uses calls to scripts like **make_build_time** rather than calls to actual shell commands in order to customize a build. For example, one engineer might copy the text of **make_build_time** into a file named **make_build_time** in his or her working directory and then edit it. To use the edited copy of the script in the next build, the engineer would add the following rule to the beginning of the current configuration thread:

```
-for make_build_time -from .
```

Note that this is possible because **make_build_time**, like all the OS translation scripts, is declared to be a tools dependency in the system models.

One disadvantage of using such scripts is the potential loss of control over programs called from the script. While the script itself is declared in the model to be a tools dependency of the system, the utilities that it calls are not. Thus, DSEE cannot record the identities of the tools called by the script in the BCT. Thus, version stamp information is lost.

To partially remedy this situation, the OS group added the full pathname to the reference of each critical tool in a script. As you can see from **make_build_time**, the Pascal compiler (**/pas**) called by the script is the one residing in the **//opera/op_sys** directory, which is a protected directory monitored by the OS group. This does not record the time stamp in the build, but it does ensure that builds use a consistent set of programs used by the script.

Another possible remedy would be to modify the system model to include all of the important utilities called by the scripts as tools dependencies.

Working in the DSEE Environment

This section examines the tasks that OS engineers need to perform to accomplish their work, given the DSEE environment that they created. Several of the explanations given earlier for certain choices that the engineers made in determining an appropriate configuration for their system illustrate some of the behavior and habits of the OS engineers. This section focuses on the ways the engineers:

- Perform development work as individuals
- Work with each other and with other engineering projects
- Create and store builds of their products for distribution

Working as Individuals

Even on large projects, such as the subject of this chapter, engineers work, for the most part, on their own. They develop and enhance code on their own, and they debug and unit test their own code.

This section discusses the ways in which OS engineers work on their own and the concerns they have.

Using Working Directories to Organize Jobs

Individual OS engineers generally use working directories to organize their work. Usually, an engineer works on each different job in a different working directory.

When the engineer is working on a job that involves modifications to one system, working on the job in one working directory is straightforward. The engineer simply reserves the lines of descent required for the job, performs the necessary editing, building, debugging, and rebuilding, and replaces the line of descent. The configuration manager promotes all the appropriate derived objects in the reserved pool to the appropriate pools.

When a job that requires changes to multiple systems dictates modifications to nucleus elements, using one working directory still presents no problems. The translation rules for nucleus Elements are the same for all systems. A build produces the same derived object, regardless of the system that is built. When the engineer replaces the nucleus element, the configuration manager promotes this one derived object to the **nuc_pool**, where it is accessible for all systems.

The situation is more complicated when an engineer is working on a job that requires changes affecting more than one system and that involves the kernel elements. The translation rules for kernel Elements differ from system to system. Even if the engineer's focus is on how his or her modifications affect one particular system, he or she is responsible for test building the modified Element for each system to make sure that the build succeeds in all instances. Each build of one Element for a different system produces a different derived object. These derived objects compete with one another for space in the reserved pool.

To alleviate contention in the reserved pool, OS engineers working under such circumstances reconfigure their reserved pools. For example, Alice, an OS engineer who is working on kernel changes that will affect both OPSYS1 and OPSYS2, reconfigures the reserved pool in **//lop/alice/grm** to accommodate derived objects for two systems at once with the following command:

```
DSEE> configure pool //lop/alice/grm -limit 2
```

Promoting the derived objects from the reserved pool, however, presents another problem. As we mentioned earlier in the chapter, the configuration manager automatically promotes only those derived objects associated with the current system when a line of descent is replaced. If Alice builds usable derived objects for both OPSYS1 and OPSYS2 in her working directory she promotes the derived objects for one system with the **promote** command.

The sequence of events that OS engineers follow in these cases is as follows:

1. Replace the elements, thereby automatically promoting derived objects for one system.
2. Set the current system to each other system one at a time.
3. Ensure that the current model setting is identical to the one that produced the derived object being promoted. (The **show builds** command provides this information.)
4. Issue the **promote** command.

Highlight: Scripts of DSEE Commands

The DSEE environment administrators among the OS engineers simplified everyone's work by creating scripts of DSEE commands. These scripts, which are stored in the **//opera/opsys/scripts** library along with the system model element and translation scripts, allow the engineers to perform a standard set of DSEE functions with one command.

One of the group's scripts, **buildk**, facilitates the process of test building for other systems (as discussed above). When the engineers execute the **buildk** command script, they supply the root name of the Element that needs to be built for each system. The **buildk** script, in turn, calls another script called **buildk_one** for each supported operating system.

```
</opera/op_sys/scripts/buildk_one ^1 1
</opera/op_sys/scripts/buildk_one ^1 2
```

(continued)

Highlight (continued)

The **buildk_one** script calls yet another script, called **set_env**, which establishes the appropriate system and model thread settings. Once these settings are established, **buildk_one** builds the Element. Here is the text of **buildk_one**:

```
</opera/op_sys/scripts/set_env ^2
von
build ^1
```

Finally, here is the text of **set_env**. Note that it issues several shell commands to copy a standard model thread into a file in the working directory, edit the thread to include the necessary **-target** rule, and then delete the file once the model thread setting has been established.

```
von
set system //opera/op_sys/opsys^1 -default
sh /com/catf //opera/op_sys/scripts/model_thread @
   > model_thread.^1
sh /com/args "-target opsys^1" >> model_thread.^1
set thread -model -from model_thread.^1
sh /com/dlf model_thread.^1
set model //opera/op_sys/scripts/os.sml
```

The **set_env** script uses the **-default** option to **set system** because some of the OS engineers save and reuse multiple working contexts for each system. By using the **-default** option, they ensure that the saved contexts aren't overwritten.

To execute a nested DSEE command script, you type the name of the script, preceded by an input redirection symbol (left angle bracket(<)). Follow the script's name with any arguments, separated by spaces. For example, to execute **buildk** to build **dtty.pas**, you would issue the command

```
DSEE> </opera/op_sys/scripts/buildk dtty.pas
```

(The DSEE input redirection facility assumes that the script you specify resides in the working directory unless you tell it otherwise. We did this in the example by specifying the **scripts** library in the script file name.)

Accessing Derived Objects

A primary concern for the OS group is accessing listing files when testing and debugging modules. Because the DSEE configuration manager insulates derived objects, the OS engineers cannot access the expanded listings their translators produce directly. They need a method of viewing the files outside of the DSEE environment.

Before the DSEE configuration manager was available, the engineers built configurations of the operating system with shell scripts. One of these scripts produced executable code and listings for the most recent versions of all main lines of descent of elements. Each new build produced with this script deleted the products of the previous build.

One function of this build script was to generate expanded listings for the operating system in a central directory. This technique gave the engineers easy access to all the listing files produced by the system build: all they had to do was to press <READ> and type in the name of the listing file. At this stage, their builds dealt only with the main lines of descent of elements and didn't use different versions of elements. Therefore, the engineers were satisfied with the limited access (that is, to the most recent build of the main lines of descent) that their technique provided.

When they began using the configuration manager, the developers replicated their old style of accessing listing files by writing scripts that exported links to the expanded listings using the DSEE **export** command and its **-link** option. These links were maintained in a central directory, much as the old technique maintained the expanded listings in a central directory. As with the previous method, this scheme provided listings for builds of the main lines of descent of elements only. Also, each new link overwrote older links, giving users access to listings produced by the most recent build only.

As the OS group became more familiar with the capabilities of the DSEE configuration manager, they started using branch lines of descent more. Moreover, engineers began building systems that had more complex sets of composite element versions. Soon their simple script became inadequate. The script couldn't accommodate the flexibility of complex configurations because it had been written to replicate a situation where builds were almost always composed of the most recent versions of elements on the main lines of descent.

Highlight: Finding Out Why a Module Needs to Be Rebuilt

At times, you may find that the DSEE configuration manager detects a need to build a module that you don't think needs to be rebuilt. You may believe that an appropriate derived object already exists in the pools (as in the case discussed in the previous chapter).

To find out why the configuration manager believes that the module needs to be rebuilt, issue the **compare builds -previous** command. This command, in effect, builds the module with the **-bct_only** option, compares this build with your previous build of the component in the current working context, and tells you how they differ.

Here is an example. You issue the **show builds** command and, based on the results, you expect that your previous build of **dtty.pas** should satisfy the desired BCT when you rebuild **dtty.pas** now. You haven't reset your model thread, your system model, or your configuration thread. However, when you issue the **build** command, the configuration manager indicates that **dtty.pas** needs to be rebuilt. To find out why the configuration manager doesn't consider the older build to be reusable, issue the command

```
DSEE> compare builds dtty.pas -previous
```

The resulting list tells you all the differences that the configuration manager detects between the older build of **dtty.pas** and the build you would produce by issuing a **build** command now. This list includes those differences between the two builds related to noncritical dependencies and options as well as any other differences. Noncritical dependencies and options may not cause the configuration manager to rebuild the component, but the configuration manager considers them to be differences between the builds. When looking for the reason that the configuration manager doesn't consider the older build to be reusable, ignore the differences related to noncritical dependencies and options.

The developers created many scripts, one geared toward each major line of descent. Each script used a private directory for its links. This technique solved the problem of accessing derived objects for different lines of descent, but it did nothing to handle access to more than one build for a line of descent. The outdated assumption that all builds would be produced using the most recent version on a line of descent was maintained by this scheme.

The OS engineers eventually determined that a far better method of accessing their derived objects was not to export the listings but to read the listings as they were required, and they use this new method now. To do this, they issue the **export** command on the build they're interested in, using the **-select** option to identify the subcomponent whose listing they wish to view and the **-read** option to indicate that they only want to read the derived object. This method allows the engineers to identify the precise build whose expanded listings they want to access. Also, it doesn't tie the engineers to builds using the most recent versions on main lines of descent. Moreover, they needn't be concerned with overwriting and storing link names—they simply read the listings as they need them.

Working with Others

The DSEE software provides many tools that help people on a large, complex project like the operating system work together without confusion. To work in harmony, engineers can work on branch lines of descent of elements, declare equivalences, set monitors on dependencies, and write configuration thread rules that refer to BCTs in release areas. However, to produce a work environment that's effective for everyone, it's necessary to establish which tools all the engineers use, and when and how they use them.

Deciding Who Takes Responsibility

There are two ways that you, as part of a large and complicated engineering effort, can view your work in relationship to that of other engineers:

- You can take the view that you are responsible for only your own work, protecting yourself from other people's work and assuming that other people protect themselves from your work

- You can take the view that you are responsible for your work in the context of the entire group's work, and that you work in a way that doesn't hinder (and, if possible, helps) other group members

The DSEE environment supports both styles of work.

To make development go smoothly, every engineer on the project should take the same approach to their work. Otherwise, people who assume that they are protected from other work find their work hampered by engineers with a different attitude toward group members' responsibilities.

The OS engineers take the second of the two views toward their work. They view all enhancements, modifications, and fixes to the software in the context of the entire group's work.

Creating Different System Configurations

In general, each OS engineer creates a different configuration of one or more systems for each job undertaken. Different configurations are the result of building with different configuration threads.

When many engineers must work in harmony, it's vital that each configuration of the system be built with care. Therefore, each engineer must use configuration threads that make careful references to the element versions and build options needed, and do not accidentally result in incorrect builds.

To provide the necessary level of organization, configuration threads have to take advantage of the organizational capabilities that the DSEE software provides. For example, many precise threads refer to alternate lines of descent. The branches referred to in these threads have to exist, however, for the thread to be effective.

Using Lines of Descent to Protect Other Group Members

Establishing and following a protocol for working on various lines of descent is essential to avoiding confusion, errors, and unnecessary rebuilding on a large project like the operating system.

The OS group adopts a straightforward protocol for determining which subprojects should work on main lines of descent and which should work on branch lines of descent. Development that affects the entire product line proceeds on the main lines of descent of the constituent elements. Maintenance work and development that affect a subset of the product line proceed on branch lines of descent.

In Figure 3–2, which shows an abbreviation of the element **ast.pas**'s lines of descent, you can see that engineers working on two projects, Phantom and Amazon, are performing modifications to **ast.pas** on branches. Phantom involves supporting a new hardware configuration. Amazon involves modifications to the graphics hardware of several of the target machines. The OS engineers use the third branch shown, **sr20.1**, for maintenance work on Software Release 20.

Note the version name given to the version of **ast.pas** from which the **phantom** branch leaves the main line of descent. The Phantom team engineers used the **[phantom_base]** version name to mark each operating system element version that they wanted to use as a basis for their development work. When they were ready to start doing development work, they simply created the initial branch version of each element to be modified from the version named **[phantom_base]**.

The Phantom team isolates its system builds by using the following configuration thread:

```
-reserved
.../phantom -when_active
[phantom_base]
```

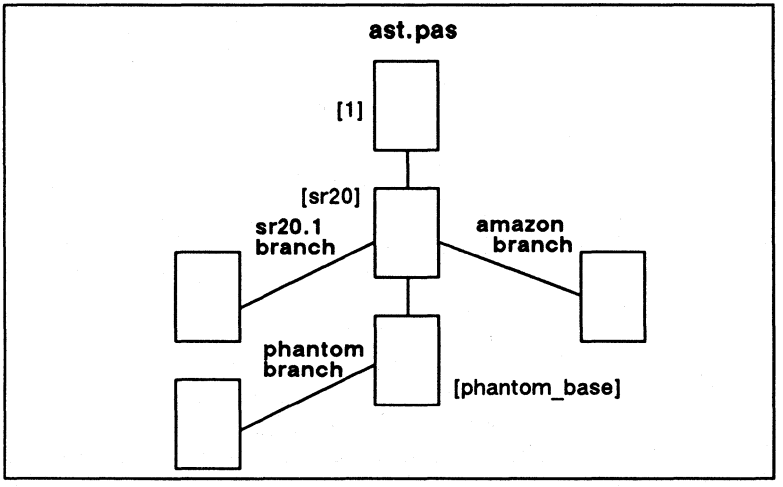


Figure 3–2. Lines of Descent of ast.pas

As you can see, the Phantom team's thread relies on a version named **[phantom_base]** for each element. While this is often a useful way to build a configuration, you should bear in mind that version names can be changed easily. This can cause difficulties, particularly when many people are working on a project and all have DSEE administrator protection access to libraries. In fact, the Phantom team encountered problems when someone outside of the team misunderstood the purpose of the version name marker. The other engineer used the **name version** command to give the **[phantom_base]** name to another version of an element.

If the Phantom team had replaced the last line of their thread with a rule that referred to a previously released build, they wouldn't have encountered difficulties when the version name was changed.

Highlight: Using Releases for Development

Frequently, it is useful to create a release solely for the purpose of ensuring a dependable base for development. This technique would have been valuable to the Phantom team, as we discussed above. Their base build could have been composed of all the element versions that they named **[phantom_base]**. Once this build was released, its BCT would be safe from pool purging and accessible through a configuration thread rule that referred to the release.

Since only a build's BCT, and not its derived objects, are referred to by a build-ID-based configuration thread rule, it's not necessary to perform a full-scale build to create a developmental release. You can issue the **build** command with the **-bct_only** option. Once the configuration manager has produced the BCT, you can use the **create release** command's **-bct_only** option to save it.

If, for example, the Phantom team had generated a **-bct_only** build from the versions named **[phantom_base]** and stored it in **//opera/releases/phantom_base**, a release area, they could use the following thread:

```
-reserved
.../phantom -when_active
opsys2!//opera/releases/phantom_base -versions -options
```

Note that, while working on branches shields people working on other lines of descent from your changes, people who work on the same branch as you are affected by your changes.

For example, Amazon team members' changes don't affect Phantom team members, but they do affect other members of their own group. Because the work for Amazon requires changes to multiple systems, kernel builds automatically promoted from one engineer's reserved pool won't be of use to another Amazon team member working on another system.

Amazon team members handle this situation exactly as if they were responsible for all the work going on for the subproject. First, they test build the kernel element(s) for each system while they still have the element(s) reserved, using the **buildk** script we presented earlier. Next, they replace the kernel element(s) and systematically set their current system to each one in turn, promoting the derived objects appropriate for each system.

Highlight: An Update on Version Naming and Development Builds

The OS group has grown considerably over time. As a result of the group's growth, the engineers have made some changes to their use of version names and development builds along the way. In the above discussions of these topics we describe the way the OS engineers used to handle version names and development builds. These methods are still worthwhile for many projects, so we haven't modified them considerably. However, the effect of OS group growth on its use of version names and development builds holds some important lessons for engineering groups that are growing rapidly.

With the great increase of group members, the OS engineers found that element version names were valuable only if judiciously used. For example, they would now never name versions **[phantom_base]**; they only name versions used in shipped releases. Over time, version names like **[phantom_base]** proliferated in OS libraries and became unhelpful. With a policy of only naming versions used in shipped releases, the OS engineers have eliminated the problems associated with many engineers and too many version names.

(continued)

Highlight (continued)

The OS engineers also now rely even more heavily on development builds (that is, builds done incrementally). They found that their increased numbers caused problems, particularly when many people were working on the same line of descent. When one engineer replaced a line of descent on an element used throughout the system, the subsequent **build** commands of all of the other engineers working on the same line of descent would require rebuilding of all the components dependent on that element—an often unnecessary cost, since the changes may not be vital to the other engineers' work.

Now, every night, one engineer issues **build** commands for each system and creates release areas for their BCTs (using the **-bct_only** option to the **create release** command). In the morning, the pools are populated with stable derived objects that all the engineers can use. They simply use configuration threads that refer to the BCT placed in the release area the night before. Because the derived objects for these builds still reside in the binary pools, they don't need to be rebuilt. The engineers' builds contain very recent material, without unnecessary rebuilding.

Here's an example configuration thread that an OS engineer might use. Note the second rule; such rules are used to obtain elements that the engineers know they have replaced since the previous night's build.

```
-reserved
-for argsv.pas .../phantom
opsys2!//opera/releases/opsys2_mar.5 -versions -options
```

Using Lines of Descent to Protect and Isolate Yourself

One point that we want to stress here is that, not only do you protect other people when you decide to do development work on a branch, but you also protect yourself. Creating a branch for your own use, or for the use of a small group of people, gives you a more dependable environment in which to perform builds.

On a large engineering project, using a configuration thread like the default thread is inconvenient. It means that your work must be constantly integrated with the work of everyone else who uses the main line of descent. If, for example, both the Amazon and Phantom teams worked on the main line of descent, every increment of each element version would require the integration of both teams' work as well as that of the people working on changes that affect the entire product line.

By working on branches, the Phantom and Amazon teams can integrate with each other and the main line of descent at discrete points in their development. Development on each branch proceeds without the complication of constant integration. Whenever an interim integration of work is required, the engineers merge a version from one line of descent into another line of descent.

In certain situations, you might want to create a private branch line of descent for your own use. This provides you with a greater degree of isolation. In isolation, you can do things that might otherwise adversely affect other team members. If, for example, you want to replace an element before you fully test it, you might create a private branch and then include a reference to it in your configuration thread.

If one of the Phantom team members wants more isolation, this example thread might be useful:

```
-reserved
.../phantom/my_branch -when_active
.../phantom -when_active
[phantom_base]
```

Coordinating with Projects Outside of the OS Group

Because the operating system software underlies and is tied to other software, non-operating system development projects need to access OS group source code. However, it's important that the other groups access the right versions of the OS elements. If other groups used modules under development, for example, the results could be quite unexpected.

Highlight: Cleaning Up Older Branches

If you use branches to isolate yourself or your project, it's a good idea to delete older branches once they've outlived their usefulness. In general, the history manager's compaction of versions makes it unnecessary to reclaim space by deleting older branches. However, deleting older branches helps people who might be browsing through your libraries to follow the development of your code without being misled by "dead ends."

The OS group, with its many engineers and subprojects, finds it very valuable to occasionally clean up the lines of descent of their elements. However, they do this with great care: they don't want to accidentally delete branches that might be useful. Therefore, they make lines of descent with no activity obsolete (using the **obsolete** command) for some time before deleting them. If, for some reason, the line of descent later needs to be reactivated, someone can issue a **cancel obsolete** command.

After a certain length of time, and after checking with all interested parties, a DSEE environment administrator deletes obsolete lines of descent.

When another project that is managed by DSEE facilities needs to access OS elements, the OS group identifies the appropriate versions of elements to use by marking them with a version name. By issuing the command

```
DSEE> name ver os2!//opera/os2_rels/sr20 os2_sr20
        -library //opera/op_sys/ins
```

for example, an OS engineer can name all the element versions in the **ins** library used in the build of **os2** contained in the release directory **//opera/os2_rels/sr20 [os2_sr20]**. (Of course, the command would have to be issued all on one line.)

The other group of engineers requiring OS elements then uses a configuration thread that contains a rule indicating that the version named **[os2_sr20]** be used for the elements they require from the operating system's **ins** library. For example, this other group might use the following thread:

```
-reserved
-for "?*.ins.pas @ ins" [os2_sr20] -when_exists
[]
```

This thread ensures that the other group's builds use the right versions of all Pascal include files required from the OS group's `ins` library.

However, some projects that need to use OS elements do not manage their builds with the DSEE configuration manager. When the engineers involved with these projects need to use the appropriate versions of OS elements, they and the OS engineers have three options:

- As above, the OS group can name versions from a build, and then either they or the other group can set an environment for the other group's builds from the version names; for example,

  ```
  DSEE> set env ?*.ins.pas[os2_sr20]
  ```

- Either the OS group or the other group can create an environment from a build for the other group's work; for example

  ```
  DSEE> create env os2!//opera/os2_rels/sr20
  ```

- The OS group can name versions from a build, as above, and then simply communicate the name to the other groups. These other groups can then use extended naming to access the appropriate versions from outside the DSEE environment. For example, once another group knew the appropriate version name, they could read the right version of a particular insert file by pressing the READ key and typing

  ```
  Read file://opera/op_sys/ins/grp.ins.pas/[os2_sr20]
  ```

Using any one of these techniques, the other group's references to OS elements are resolved to the appropriate versions.

Releasing an Operating System for Distribution

When the OS engineers are ready to release a version of one of their products for general distribution, they want to generate a build for the product's system that has

- No reserved versions of elements
- Only one version of an element used in all contexts (known as a "flat build")
- A controlled set of options (both critical and noncritical) used in all contexts
- A controlled set of versions for all elements, whether or not they are critical dependencies

Generating this type of build is largely a matter of using the right thread and specifying the right options to the **build** command.

To make certain that their build contains no reserved versions of elements, the engineers exclude the **-reserved** version rule from their configuration thread. They also eliminate any rules that restrict the application of a version rule or a translation option rule to one particular context (rules with **-under** clauses). All options desired in the build, critical and noncritical, are specified in the thread. If they decide that they specifically do not want certain noncritical options in their released build, the engineers include the **-exact** specification in the thread's translation option rules.

Issuing the **build** command with the **-noequivalences** option seems to make sense for the OS engineers' released build, since they do not want to use equivalent builds that might have been built with different versions of elements than they want in this build. However, **-noequivalences** does not ensure a flat build, because the OS group's system model source file declares some dependencies to be noncritical. The **-noequivalences** switch does not ensure that the same version of a noncritical dependency is used throughout a system.

To guarantee that the system and all of its components are rebuilt, regardless of whether they are noncritical, the engineers use the **-force_all** option. Because **-force_all** tells the configuration manager to rebuild everything, it makes **-noequivalences** redundant.

Once they generate this build, the OS engineers double-check to make sure that it contains only one version of each element. They do this by issuing the **examine build** command with the **-check** option, which examines the build for multiple versions of elements, files, and tools. Then, they create a release area to hold the build's derived objects, BCT, and dependencies that are not DSEE elements.

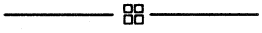
Highlight: An Alternative to -force_all

One disadvantage of using the **-force_all** switch is that it causes the configuration manager to rebuild every buildable system component, even when existing builds could be reused. This can considerably lengthen the amount of time it takes to perform the build.

The OS engineers have devised a way to force a rebuild of the system that reuses builds when possible. They created two dummy source dependencies, the elements **force_build.pas** and **force_build.asm**. The first they made a default source dependency for all the buildable Pascal components of the system; the latter they made a default source dependency for all the buildable assembly language components of the system. (In each case, the dummy file is a critical dependency.)

When the OS engineers want to make sure that all the Pascal modules get rebuilt, they create a new version of **force_build.pas**; when they want to make sure that all the assembly language modules get rebuilt, they create a new version of **force_build.asm**. Their configuration threads request the most recent versions of these elements, so creating new versions automatically forces the configuration manager to rebuild dependent components.

While this method may not be appropriate for a release since it does not ensure appropriate contents, it is useful for other situations. Forcing builds this way rather than using **-force_all** cuts in half the number of builds that will be rebuilt. Only Pascal modules will be rebuilt forcibly in one case, and only assembly language modules will be rebuilt in the other.



Chapter 4

Case Study 3: Maintaining Released Software in a DSEE Environment

In the previous chapter we examined the way that DSEE facilities help engineers control a large-scale development project. In this chapter, we focus on using DSEE software to control simultaneous maintenance and development. We investigate the ways that you can use DSEE facilities to maintain multiple releases of software while continuing to develop the product. We discuss the kinds of version and branch naming conventions that you should establish and how you can use them.

The subject of this chapter's case study is the DSEE software project itself. The engineers on this project use the DSEE environment to develop new capabilities as well as to support multiple released versions of the software. This group's goals are similar to those of many on-going engineering project teams. The engineers must create and maintain a few different releases of the same product while going forward with development of the next generation of the software.

In this chapter we present the DSEE facilities that enable these engineers to do simultaneous development and maintenance. We also look at the protocols the DSEE engineers have adopted that facilitate their work.

One word about this case study and the DSEE software: Our presentation of the DSEE project will discuss the development of Version 2.1 of DSEE software. However, the methods that we present the DSEE engineers as using will be those of current DSEE software capabilities. In effect, our discussion presents what the DSEE engineers would have done when developing Version 2.1 if they'd had the capabilities of the current DSEE software.

Abbreviated versions of the DSEE group's system models appear in Appendix C.

Introduction

Two major software programs provide the DSEE environment's capabilities: the system model compiler and a binding of the executable commands. Each of these two programs is represented by a separate system model. The command program consists of several parts: all the configuration management facilities; all the non-configuration management facilities; and the database management facilities underlying the history manager. The command program's system model also builds several other programs: the **make_model** utility; the builder; the module that creates a shell in response to the **create environment** command; and a group of executable modules shared by the system model compiler. Of these separate programs built by the command system, only the first, **make_model**, is user-callable. The command system itself makes calls to the other three programs.

The DSEE project has been managed with DSEE facilities ever since its inception in 1983. The DSEE project is a small-scale engineering effort, with all of its engineers working on both maintenance and development.

Simultaneous Maintenance and Development

Like many active software projects, the DSEE project is concerned with both the maintenance of several released versions of the product as well as new development for future release. DSEE software has been in use by our customers for several years. Inevitably, widespread use of the product turns up bugs that the engineers have to fix. Moreover, DSEE capabilities are still evolving. Version 2.0 contains significant increases in capability over Version 1.0. Plans

for DSEE Version 3.0 contain enhancements to Version 2.0's functions. The DSEE group is implementing Version 3.0 functions in stages called base levels. The group distributes base levels to certain in-house users for testing every few months. In addition to new capabilities, each base level includes all bug fixes performed by the date of the base level release. This ensures that the bug fixes get tested thoroughly in-house before they are released to all customers.

In short, the DSEE team's work has three major aspects:

- Developing new capabilities
- Fixing bugs in released versions of the software
- Releasing new versions of the software

Because both developing new capabilities and fixing bugs ultimately lead to new releases of the software, the DSEE engineers focused on releases when establishing working methods.

The DSEE engineers' first step in controlling their work with DSEE facilities was to define their release requirements. They determined that, at any given time, they had to support one major release of the product in the field and the most recent in-house base level. Maintaining the field release would involve accumulating bug fixes and eventually releasing and distributing an updated version containing bug fixes. Maintaining the in-house base level also involves releases for bug fixes, even though the time between base level releases is short, because some serious bugs need to be fixed quickly, before the next base level is ready.

The DSEE team also determined that, in addition to major releases and base levels, they needed to be able to generate "specials"—releases tailored for one or two customers' needs. Specials would contain some of the bug fixes to the last major release plus some subset of the new capabilities contained in base level releases. In certain cases, a special would also contain capabilities that would never be part of a major release.

Eventually, the engineers knew, they would have to bring together the new development work and the field release bug fixes to form the next major release, Version 3.0. They wanted to make sure that their strategy to isolate development and maintenance work didn't make the eventual merge an arduous, error-filled process.

Once the engineers had determined their requirements, they outlined a management strategy that involved an ordered plan of element version and branch names and configuration threads to refer to the names. Most of this chapter is devoted to the examination of the DSEE team's naming strategy and the way the engineers use it.

Project Structure

We start our examination of the DSEE project team's use of their own product by presenting their

- Libraries and elements
- Tasks, monitors, and tasklists
- Systems and system models

Libraries and Elements

The DSEE engineers store most of their source code in two libraries: **//orange/case/case_1**, which contains the history, task and monitor managers' sources and insert files; and **//orange/case/case_cm**, which contains the source modules and insert files for the configuration and release managers. Source modules for the system model compiler are stored in the library **//blue/smc/smclib**. The library containing tools specific to the DSEE product is called **//orange/case/tools**. Two other libraries, **//opera/ios/mgrs/case** and **//opera/ios/ins**, hold the source modules and the inserts, respectively, for the dsee_history_manager_file type manager.

As we mentioned in the introduction to this chapter, the system model compiler and the DSEE command program both have their own system models. The system model compiler's model is stored in **//blue/smc/smclib**, and the command program's system model is stored in **//orange/case/case_cm**.

Table 4–1. DSEE Group Libraries and Their Contents

Library Name	Contents
//orange/case/case_1	Source modules and insert files for history, task and monitor managers
//orange/case/case_cm	Source modules and insert files for configuration and release managers; system model
//blue/smc/smclib	Source modules and insert files for system model compiler; compiler system model
//orange/case/tools	Tools specific to the DSEE product
//opera/ios/mgrs/case	Source modules for dsee_history_manager_file type manager
//opera/ios/ins	Insert files for type managers
//red/case/design	Design notes
//red/case/dsee_tutorial	Online graphic overview
//red/case/dsee_cmd_ref	Command reference text and system model
//red/case/dsee_call_ref	Call reference text and system model
//red/case/help_files	Online help files

The DSEE team and the technical writer associated with the project also use DSEE libraries to store their documentation. The engineers keep their design notes in a library called **//red/case/design**. The writer uses four libraries: , **//red/case/dsee_tutorial**, which holds the online tutorial; **//red/case/dsee_cmd_ref**, which holds the source modules for the manual, *Domain Software Engineering Environment (DSEE) Command Reference* as well as the system model that formats the book; **//red_case/dsee_call_ref**, which holds the source modules and system model for the manual, *Domain Software Engineering Environment (DSEE) Call Reference*; and **//red/case/help_files**, which holds the text for the DSEE help files.

Table 4–1 summarizes the libraries that the engineers and writers use.

Version and Branch Name Strategy

Key to the DSEE group's management of their various types of releases are their version and branch naming conventions. The engineers have established guidelines for determining the version names that they employ for every element they use.

The engineers use lines of descent to distinguish among various types of work and version names to mark the versions that constitute a particular release. Development of new capabilities for Version 3.0 is done on the main line of descent, and bug fixes are done on branch lines of descent. Development for special releases, when necessary, is done on branch lines of descent.

The DSEE group uses the same strategy to name both versions and branches. Major distributions are known as "v*n*," where *n* is the release number. For example, the versions of source elements that constituted DSEE Version 2.0 are named **[v2]**, and the versions used in DSEE Version 2.1 are named **[v2.1]**. Base level distributions are known as "v*n*.bl*m*"; *m* is the base level number, and, again, *n* is the major version number. For example, the source versions of elements used in the first base level distribution after DSEE Version 2.0 are named **[v2.bl1]**.

Highlight: Ensuring Consistency of Branch Names

Adherence to branch naming conventions provides engineers with consistency and a dependable structure for their work. Consistent branch naming makes it possible for project team members to share configuration threads for builds of branch versions. Also, when you're doing branch work, your job is much easier when there's only one branch name you need to know for all your **reserve**, **replace**, and **merge** commands. Moreover, consistent branch naming makes it possible to check the status of work on branch projects with the **-having**, **-missing**, **-having -merge**, and **-missing -merge** options to the **show elements** command. In short, consistent branch naming is an important factor in standardized element evolution, which we discuss in a highlight of that name later in this chapter.

In order for any naming convention to be useful, every member of a project must use it consistently. However, there's always the possibility that someone will make a typographic error, or forget whether an underscore or a period is used between parts of a name.

To minimize the chances of incorrect branch names, Bob, the DSEE project engineer, writes scripts for other engineers to use when they create branches. For example, Bob's script (called **bl2_bugfix_branch.dsee**) for creating a branch for bug fixes to Version 2, base level 2 creates a branch named **v2.bl2_bugfix** for a given element. Whenever the DSEE engineers need to create a branch for fixes to Version 2, base level 2 they execute this script. Here is the text of the script (which, in reality, all appears on one line).

```
create branch v2.bl2_bugfix ^1[v2.bl2]
-comment "Branch for base level 2 bugfixes"
```

This script does more than ensure that the branch is correctly named. It also ensures that the branch originates from the appropriate version of the element, and that the commentary on branch creation included the information that the project engineer wants it to.

Bug fix branch lines of descent originate from the versions used in the distribution that contained the bugs. These lines of descent are named for the distribution and the purpose of the branch. For example, fixes to bugs reported in base level 2 are done on branch **v2.bl2_bugfix**.

As we discussed in the introduction to this chapter, base level releases contain all of the bug fixes performed by release time. Combining the bug fixes with the base levels ensures thorough testing of the fixes. Incorporating these bug fixes into the development work requires periodically merging the bug fix branch into the main line of descent.

Periodic merging of the bug fix branch not only increases the amount of testing the bug fixes get, it also eases the difficulties inherent in doing simultaneous work on code along several separate paths. The longer a branch lives without being merged into the main line of descent, the more likely it is to differ significantly from code on the main line. These differences make merging the branch into the main line of descent more difficult. The DSEE engineers' interim mergers of their branch work into the main line of descent keep the number of differences between the merged versions down to a tolerable level.

The engineers assign version names in large groups, effectively tagging all of the versions used in a particular build of the system by identifying the build ID in the **name version** command. Because not every element has every branch (for instance, an element that didn't have a bug in base level 2 won't have a **v2.bl2_bugfix** branch), engineers create branches (and, consequently, assign them names) only as needed.

Figure 4–1 illustrates the evolution of the DSEE element **cm_utl.pas**.

Highlight: Naming Versions from Build IDs

In each of the case studies we've examined so far in this book, we've mentioned naming element versions from build IDs. We want to stress the ease and benefits of this technique again.

Naming versions from build IDs allows you to use DSEE facilities to identify every element version that went into a particular system build. You don't have to try to compose the list of constituent versions yourself; all you have to do is issue the **name version** command, identifying the particular build whose BCT you want the history manager to use when identifying versions to bear a particular name. For example, the command

```
DSEE> name version dsee!21-feb-1986.21:32:36 v2.2
```

names all the element versions used in the build of the DSEE command facility **[v2.2]**. The versions of the system model elements (the element that contains the root model and all elements containing model fragments) that were used in the build are also named **[v2.2]**.

You can use the name of a released build rather than a build ID in a **name version** command; for example

```
DSEE> name version !//black/case/releases/v2.2 v2.2
```

In addition to illustrating naming conventions we have discussed, Figure 4–1 presents an example of the engineers' version naming rules for "respins"—redistributions of base levels, augmented with bug fixes for the base level code. As you can see from the figure, the group distributed a respin of the first base level which contained all of the bug fixes done on the **v2_bugfix** branch up to the time of the respin plus some fixes to the code in the first base level. The engineers named all the versions used in the respin **[v2.bl1.1]**. The respin of the second base level was composed of the element versions named **[v2.bl2.1]**.

Note that the respin of the first base level was created after the **v2.bl1_bugfix** and **v2_bugfix** branches were merged into the main line of descent. However, the respin of the second base level was

created on a branch. The merger was desirable for the first base level's respin because it allowed the engineers to incorporate bug fixes, which had been merged into the main line just before the respin, into the first base level's respin. Merging into the main line of descent wasn't desirable for the second respin because the engineers wanted to avoid picking up Version 3.0 development work in the respin.

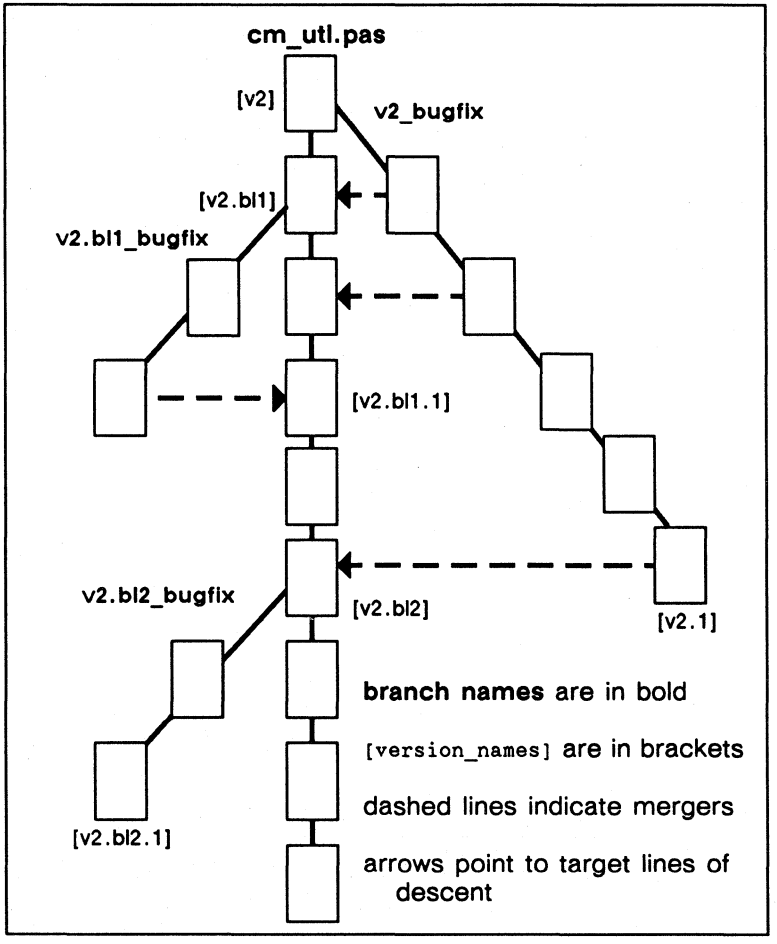


Figure 4–1. Derivation of Element cm_utl.pas

Highlight: Standardized Element Evolution

It's worth noting that Figure 4–1 could represent the evolution of almost any element that the DSEE group uses in its system, not simply the history of **cm_utl.pas**. As the words imply, a naming convention like the DSEE group's applies to all the elements used in the system.

The result of consistent use of a naming convention is standardized element evolution. You need only learn the conventions to be able to figure out what branches are for which purpose, and which element versions constituted a particular release. While not all elements may have every branch (for example, only elements that need to be modified for bug fixes have bug fix branches), each required branch's purpose conforms to the naming convention. The same is true for version names: an element created since Version 1 won't have a version named **[v1]**, but all its other version names will conform to the convention.

This standardized evolution is possible because the naming convention doesn't rely on the characteristics of a particular element's development. Version numbers aren't important. For example, the version of **cm_utl.pas** named **[v2.bl1.1]** could be version number **[12]**; however, version number **[25]** of the element **bld_previous.pas** could be named **[v2.bl1.1]**. Moreover, it might be the case that, in element **bld_previous.pas**'s evolution, version number **[25]** is also named **[v2.bl2]**. Version names tie together all related versions with one logical mnemonic name—an extremely useful abstraction.

Tasks, Tasklists, and Monitors

The DSEE engineers take advantage of the facilities of both the task and monitor managers to coordinate their work. First we examine the group's use of tasks and tasklists, and then we look at some of the monitors they employ.

Tasks and Tasklists

The DSEE engineers share all of their maintenance and development work. Each engineer is responsible for some aspect of new development as well as some bug fixes. Work for special releases is also performed by all members of the group.

Reflecting this distribution of responsibility, the engineers make little use of personal tasklists and use, instead, shared tasklists. For example, all of the engineers use the tasklist **//red/case/v3_tasks**. This tasklist contains several tasks itemizing the work that the group needs to do to produce DSEE Version 3.0, such as:

- Schedule Version 3.0 work
- Fix bugs in Version 2.0
- Enhance **merge** command

For the most part, these tasks have relatively few active items. They serve as records of the work that is performed for a certain task; as such, their task transcripts are their most important components. For example, the task "Fix bugs in Version 2.0" contains no active items. (Open bugs are recorded elsewhere.) Its transcript, however, contains many entries like this one:

```
✓ Replaced bld_pass4.pas[1]
  As part of the task entitled:
  DSEE V2 bugfixes
  ----------------
      Fix "build -bct_only -von" to not cause a
      system error when the component being built has
      NIL_TRANSLATION.
  Completed: 12-Aug-1985 16:09
  Completor: Robert Eastwood at //FIT (bob.none.r_d.4E)
```

When Bob is fixing a bug in Version 2.0, he sets the "Fix bugs in Version 2.0" task as his current task. Every time he replaces an element, the history manager records the event in the task transcript. This provides the engineers with a list of all versions created for Version 2.0 bug fixes.

Monitors

The DSEE engineers use many monitors. Several of these monitors demonstrate some interesting applications of the monitor manager, such as:

- Watching other engineering projects' elements for changes that might affect the DSEE system
- Issuing a warning to DSEE engineers when they change an element
- Notifying technical writers of changes to the user interface or the product design

Monitors on Other Projects' Libraries

As we mention in Chapter 1, the DSEE facilities use the Domain/OS store–and–forward facility. Use of this facility makes the DSEE software dependent on that facility's current capabilities. To automate tracking this dependency, the DSEE engineers set a monitor that watches the interface description of the store and forward facility.

Warning Monitors

The DSEE engineers outline the design, direction, and implementation of the DSEE software in a series of design notes. One of these, titled "Notes on the Implementation of the Configuration Manager's Builder," describes how the system builder is implemented. It is required reading for all engineers modifying the source code of the system builder.

To make sure that no one modifies the system builder's source code without reading this design note, one of the DSEE engineers set a monitor on all the pertinent elements. This monitor has an empty activation list (that is, it creates no tasks and executes no shell commands). Its purpose is to remind engineers who activate the monitor that they should read the relevant design note.

When an engineer reserves a monitored element, the monitor manager informs the engineer that the element is monitored and prints the purpose of the monitor. This purpose is the text written when the monitor was created. In this particular instance, the engineer who created the warning monitor inserted the following text in the edit pad for describing the monitor's purpose:

```
Have you read the design note titled "Notes on the Implementa-
tion of the Configuration Manager's Builder"?
```

Monitors that Keep Writers Abreast of Product Changes

As we note in the previous section, the DSEE engineers describe the goals and course of their work in design notes. These design notes are stored as elements in the library **//red/case/design**.

The technical writer responsible for DSEE documentation has set a monitor on all existing and new elements in the design notes library. This monitor makes her aware of the presence of new design notes and changes to older design notes. The technical writer's monitor adds a task to her personal tasklist, so the historical information associated with the monitor–triggering event becomes part of the new entry on her tasklist. Thus, she knows the details of the design note change.

Reading design notes keeps the technical writer up–to–date on the direction of the product's evolution. However, design notes occasionally don't contain specific details on how a change in product direction affects product use. Therefore, the technical writer has a second monitor set on the source code for the DSEE command line parser. Whenever the engineers change the parser to recognize new command syntax, this monitor lets her know of the event. Again, because this monitor adds a task to her personal tasklist, the writer can see the historical information associated with the triggering event. This data generally includes details of the new command syntax and gives the writer a basis for discussion with the engineers about new syntax.

Systems and System Models

Although users see the DSEE software as one package, it has several parts. The command facility is the largest piece of the package; other pieces include the **make_model** utility, the system builder, the facility that executes **create environment** commands, and the system model compiler. All the parts but the last are represented by components of the system model stored as the element **//orange/case/case_cm/dsee.sml**. The system model compiler is larger than all parts of the DSEE software other than the command facility. For convenience, it is represented by a separate system model: **//blue/smc/smclib/smc.sml**.

The system built by **dsee.sml** calls the compiler when you execute the **set model** command, passing it the pathname of the system model file that needs to be compiled. After a call, the compiler returns to the other DSEE system the binary result of compiling the system model. This interface from the compiler to the DSEE system is very stable, depending on a binary format that doesn't change often.

The two models are largely independent of one another. Changes in one program are unlikely to affect the other. Only when the DSEE engineers are doing a build for a release do they build both systems at the same time, to ensure that the release has the most up–to–date built versions of both systems.

There is a small subset of the DSEE system's software that the system model compiler system also requires. This subset consists of several utilities the systems have in common (for example, error reporting functions). It also includes the description of the binary format that both systems require (discussed above). This package is collected in an include file called **dsee_common.ins.sml**. Both **dsee.sml** and **smc.sml** contain **%include** directives that fold the shared utilities into both of the models.

The DSEE engineers would like the two systems to be able to reuse one another's builds of the components in **dsee_common.ins.sml** whenever possible. Doing this requires some careful coordination between the two models. In order for systems to share builds, their models must have four things in common:

- The dependency structures of the shared components must be identical
- The translation rules for the shared components must be identical (with the exception of any noncritical options)
- The logical, as well as physical, names of the libraries that contain the shared source modules must be identical
- The logical, as well as physical, names of the pools used to hold the shared derived objects must be identical

Model fragments can help the engineers meet all four requirements. The DSEE engineers take care of the first requirement by isolating the shared components in the include file **dsee_common.ins.sml**, which both of the models use. The engineers handle the requirement of identical translation rules by isolating the translation rules into yet another include file called **dsee_default_trans.ins.sml**. This include file is included by both system models prior to the inclusion of **dsee_common.ins.sml**.

Currently, both system models contain identical logical pool and logical library declarations for **dsee_common.ins.sml** components. The physical pool and library pathnames contain references to links to ensure long-term accuracy if the engineers need to move objects later. They could create a hedge against future difficulties by placing some commentary in the beginning of the **dsee_common.ins.sml** fragment that warns all users of the fragment that certain pool and library declarations are required. If they wanted to eliminate all possibility of disparate pool and library declarations, the engineers could also use a model fragment to isolate these declarations, as they do with the translation rules.

Highlight: Importing Derived Objects from Other Systems

The DSEE engineers use model fragments shared by two system models to share the built components in **dsee_common.ins.sml** because they require version control for constituent elements when building both systems. This was not always the case. In the past, they shared the binaries by declaring the components of the binaries in only one of the system models (**dsee.sml**). These components were grouped together in an Aggregate whose translation rule bound the binaries together and copied the bound binary into a directory. The system model compiler system model then imported the binary from this directory. This technique was useful for their purposes, although it meant that building the system model compiler was done with no version control over the constituent components of the shared binaries and made it difficult to trace bugs in the system model compiler back to source components.

If you don't need tight version control over the constituent elements of a shared binary, you might find this approach useful. By assuming that whatever build of the binary is in the directory is good enough for one of the two systems requiring the code, you eliminate any possibility of a rebuild of the code for one of the two systems. (As we mentioned in Chapter 2's highlight, "The Pros and Cons of Imported Derived Objects," the stability of the shared binary is a vital factor in this decision.)

If you choose this approach, you may want builds of the system model that imports the shared binary to contain some information about the binary. This would help you track down the source versions of the binary and fix bugs.

(continued)

Highlight (continued)

To incorporate version information of imported binaries into builds, edit the importing system model and declare the imported binary to be a tools dependency of the model. This action will ensure that the version stamp of the file is recorded in the BCT of each build. Then, if you find a bug in the importing system, you can examine the BCT of the problematical build (with the **examine build** command if the build is still stored in the binary pools, or by looking at the readable build map file if the build is now in a release area). Compare the version stamp you find in the system's BCT for the imported binary against version stamps of released builds of the exporting system's build of the component by issuing the **show version** command with its **-from** option; for example:

```
DSEE> show version
-from //node_1/releases/v32/exports/shared/shared
```

(Of course, the entire command would have to appear on one line to be executed.)

When you find a version stamp for a released build of the exported component that matches the version stamp of the imported binary in the system with the bug, you can determine the composite element versions by examining the release of the exported component build.

To avoid requiring a rebuild of your importing system model whenever the exporting system overwrites the exported binary with a new copy, declare the binary to be a noncritical dependency of the importing system.

If you share binaries in this manner, you must remember the importing system's automatic dependency on the last build of the binary when you rebuild the component—particularly when you use older versions of constituent elements. A configuration of the exported component built with older versions may not perform well with the importing system. You must make sure that incompatible configurations aren't inadvertently made available to the importing system.

Working in the DSEE Environment

In this section we examine how the DSEE engineers use the conventions for branch and version naming they established to develop and maintain their product. Our examination focuses on several aspects of the engineers' work procedures, including:

- How they create a special distribution
- Why and how they do interim merges of bug fix branches into the main line of descent, and how their naming conventions facilitate the merge process
- How they produce a bug fix release of an existing shipped product

Throughout these discussions we concentrate our attention on the building and releasing of the software represented by the system model **dsee.sml** only. The system model compiler isn't undergoing new development, so all bug fix modifications are occurring on the main lines of descent of compiler elements. Therefore, builds and releases of the system model compiler are uncomplicated by the variety of work involving the other DSEE software.

Creating a Special Distribution

As we mention in the introduction to this chapter, the DSEE engineers occasionally have to generate special versions of their system for one or two customers. These "specials" include subsets of the bug fixes being performed and the new capabilities being developed. Some specials might contain functions that never become part of the official product.

To illustrate the way that the DSEE engineers handle a special release, we will follow them through the development and release of one special distribution. This special version of DSEE was created for a company called INCO. The general outline for this release's structure can be summed up as follows:

- DSEE Version 2.0 served as the basis for the INCO version
- Several new capabilities were added
- All available bug fixes were incorporated into the INCO version

To produce the special version, the DSEE engineers created branches off of the main lines of descent of all elements in the system that need to be altered for the special release. This branch started at the versions named **[v2]** and is named **inco**.

The engineers then created new versions on the **inco** branches as they added new capabilities to various elements. Then they were ready to add all available bug fixes.

Once they'd done all the INCO-related work on the elements, the DSEE engineers merged the line of descent with bug fixes to Version 2.0 into the INCO special branch using the **merge -reserve** command. Their subsequent **replace** command used the text of the merge file to create a new version on the branch.

Once the developers had done all the necessary work to elements with **inco** branches, they were ready to build the special version of DSEE software for INCO.

First, the engineers set a model thread that would use all of the appropriate versions of **dsee.sml**, **dsee_default_trans.ins.sml**, and **dsee_common.insl.sml**:

```
.../inco -when_exists
.../v2_bugfix -when_exists
dsee!//orange/case/releases/v2
-target dsee!//orange/case/releases/v2
```

This model thread used the most recent model fragment version on **/inco** branches if they existed. If a fragment did not have an **/inco** branch, the model thread used the most recent version on the **/v2_bugfix** branch for that fragment, if there was such a branch. Had a fragment neither of these branches, the thread used the same version employed in building **dsee!//orange/case/releases/v2**. In all instances, model validation used the same **-target** specifications as used in **dsee!//orange/case/releases/v2**.

The engineers used a configuration thread similar to the model thread to construct the special version.

```
.../inco -when_active
.../v2_bugfix -when_active
dsee!//orange/case/releases/v2 -versions -options -exact
```

Note that the last configuration thread rule refers to the BCT of the released build for Version 2.0 rather than referring to the versions named **[v2]**. This ensures that exactly the versions and options used in the released product are incorporated in the special release, a particularly helpful safeguard when there's any possibility that the released build might have been constructed with different options or element versions in different contexts.

Another noteworthy aspect of this configuration thread is the second rule, which tells the configuration manager to use the most recent version on the bug fix line of descent should an element not have an active **inco** branch. Not all the elements used in the system were modified for the INCO release; however, the unmodified elements may well have been fixed for bugs. The second rule in the configuration thread ensures that the bug fixes to these elements are incorporated in the build.

The similarity between the model thread and configuration thread points out the flexibility available when you use model threads. You can use all the history manager functions that allow you to do simultaneous maintenance and development in code work in model development as well.

When the engineers completed the build for INCO, they decided that they wanted to incorporate the new capabilities they'd produced for INCO in regular releases of the product. They decided to add the new capabilities to the main line development.

Again, the engineers used the **merge -reserve** command to merge the most recent versions of the **inco** branches into the main lines of descent. Here is the command that produced a merged working directory copy of element **cst.pas**.

```
DSEE> merge cst.pas/inco[] -with cst.pas -reserve
```

Merging the most recent versions of the branches into the main lines of descent meant that the most recent bug fixes were also merged into the main line, since these fixes had been incorporated into the **inco** branch. This side effect presented no difficulties because, as we've mentioned, the engineers periodically merged the bug fixes into the main line of descent anyway. In fact, the next formalized round of interim merges became simpler as a result of merging the **inco** branch into the main line: many of the bug fixes performed on the bug fix branch had already been incorporated into the main line of descent. To determine which bug fixes still needed to be merged into the main line at that point, the engineers executed the following command:

```
DSEE> show elements -missing /inco
```

The engineers then checked which of these elements had bug fix branches and then merged the bug fix branches of those elements directly into the main lines of descent.

Figure 4–2 depicts the **inco** branch off of the element **bldcom.pas**.

Merging Bug Fixes into the Main Line

As we mentioned in our discussion of branch and version naming conventions, the DSEE engineers do interim merges of bug fix branches into the main line. These merges provide a means of testing bug fixes, and they help avoid the difficulties of merging two very disparate lines of descent later in their evolution.

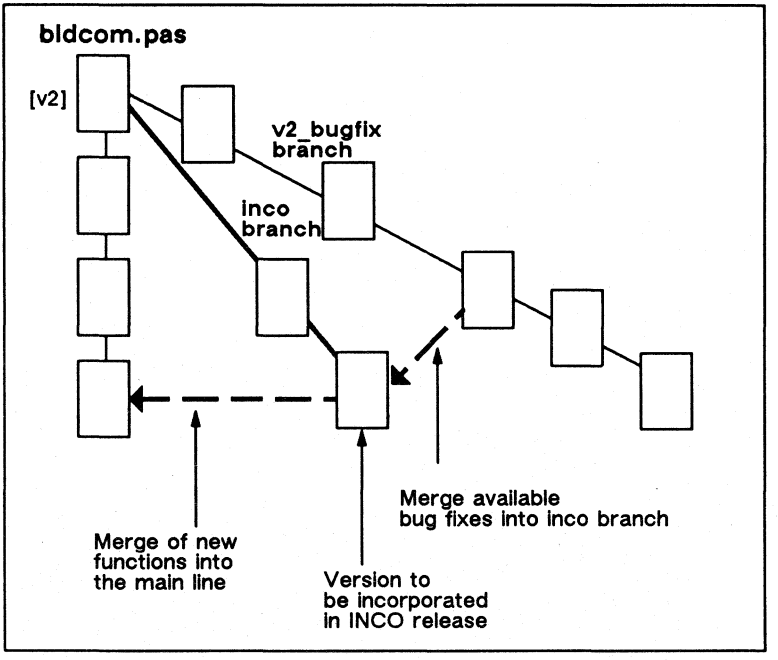


Figure 4–2. Evolution of bldcom.pas's inco Branch

The engineers merge in bug fixes just before they create a base level release. Because of the many elements involved, the process of merging in each bug fix branch for the base level can become confusing. In order to find out which elements have bug fix branches that haven't been merged into the main line of descent, the engineers use the **show elements** command with the **–missing** and **–merge** options. For example, when Erica is doing an interim merge of the Version 2.0 bug fix branch, she can issue the command

```
DSEE> show elements -missing -merge /v2_bugfix
```

to determine on which elements she still needs to perform **merge** commands.

Producing a Bug Fix Release

In effect, every release of the DSEE software is a release of bug fixes. As we've seen, each base level release is also a bug fix release, since all available bug fixes are merged into the main line of descent before each base level release. Special releases, also, include bug fixes, as our discussion of the INCO release illustrates. But there are two types of releases devoted to bug fixes: respins and bug fix releases. Respins contain bug fixes to a base level. A bug fix release contains fixes to bugs found in a version of DSEE software in general use.

Since we've just discussed how the DSEE engineers do interim merges of the bug fix branch, it would be interesting to see how they produce a bug fix release. Such a release draws its constituent versions from the bug fix branch.

We will look at a release of bug fixes to Version 2.0 (named Version 2.1) using DSEE facilities.

First, we'll locate the release area containing Version 2.1. A **show releases** command issued for the DSEE system produces the following list of release areas:

```
DSEE> show releases
The following release areas currently exist:
   //black/case/releases/inco
   //black/case/releases/v2
   //black/case/releases/v2_plus_inco
   //black/case/releases/v2.1
   //black/case/releases/v2.bl1
   //black/case/releases/v2.bl1.1
   //black/case/releases/v2.bl1.2
```

The release that we're interested in is **//black/case/releases/v2.1**. We issue a **examine release** command to find out more about this release:

```
DSEE> examine release //black/case/releases/v2.1
Release CREATED on 23-Feb-1986 13:32
  by Robert P. Eastwood at //RED (bob.none.rd.4E)
  Command was: cre rel  -/irel/v2.1
  -from  dsee!21-feb-1986.21:32:36 -exp *-/dc/release_list
-----------------------------------
Version 2.1 - Bug Fix Release for Version 2.0.
Thread was:
.../v2_bugfix -when_active
!//black/case/releases/v2 -ver -options -exact -when_exists
[]
**********************************
This release contains the following exported components:
                              current  timestamp
                               length
Component dsee
(v2.1/exports/dsee/...)
   dsee                       1165592  21-Feb-1986 21:30:25
   dsee.bct                     40544
   dsee.bld                    275317


Component create_env_shell
(v2.1/exports/create_env_shell/...)
   create_env_shell             5742  21-Feb-1986 18:39:02
   create_env_shell.bct         1584
   create_env_shell.bld         2440
Component dsee_builder
(v2.1/exports/dsee_builder/...)
   dsee_builder                 8378  21-Feb-1986 15:44:21
   dsee_builder.bct             1772
   dsee_builder.bld             3200
Component library_database.ddl
(v2.1/exports/library_database.ddl/...)
   library_database.sch        11032
   library_database.sub        22248
   library_database.uwa.pas     9751
   library_database.bct          794
   library_database.bld         1065
Component make_model
(v2.1/exports/make_model/...)
   make_model                   6892  21-Feb-1986 15:45:45
   make_model.bct               1228
   make_model.bld               1386
```

In the comment area, the DSEE engineers included the text of the configuration thread that they used to build the release.

Highlight: Threads and Obsolete Branches

The DSEE engineers use the **-when_active** option to branch version rules rather than the **-when_exists** option because they want to avoid building with branches that may exist, but that have been declared obsolete. This eliminates the possibility of their builds including work that is not yet ready for incorporation into the software.

Occasionally, a DSEE engineer will create a branch for an element, start working on the branch, and then, for some reason (for example, when an emergency situation arises which requires them to turn their attention elsewhere), leave the work incomplete. When this happens, the engineer replaces the line of descent (so as not to lose track of the work in the working directory) and declares the line of descent obsolete. Because all the engineers use **-when_active** rather than **-when_exists**, coworkers won't accidentally pick up the incomplete work in their builds.

Another situation in which you might use **-when_active** is when you want to build a configuration when some, but not all, of the branches of a given name have been merged into their elements' main lines of descent. In this case, you would have to declare the branches to be obsolete once you'd merged them. Then you would use a configuration thread that built with the branch only when it was active, and with the main line of descent otherwise.

Deciding when to use **-when_exists** and **-when_active** in a configuration thread branch version rule depends on what declaring a branch to be obsolete means to you. To the DSEE engineers, an obsolete line of descent is one on which work is temporarily halted. They will use the **cancel obsolete** command to reactivate the line of descent when they need to continue development on the branch. Other engineers—for example, people declaring merged branches obsolete—may think of an obsolete line of descent as one on which work has been completed. They can still use versions on the obsolete branch by employing the **-when_exists** option.

There are a couple of interesting aspects of the bug fix release that the **examine release** command doesn't reveal. One is that the DSEE engineers issued the **examine build** command with the **-check** option before they created the release area to make sure that the release contained only one version of every constituent element. (We discuss this in more detail in Chapter 3.)

The DSEE system model's logical pool declarations are delimited by conditional constructs, as you can see in the following fragment from the system model's text.

```
%if bl_pool %then
pool
   cm_pool = '//black/case/bl_pool/cmbin';
   hm_pool = '//black/case/bl_pool/hmbin';
%elseif bugfix_pool %then
pool
   cm_pool = '//black/case/bugfix_pool/cmbin';
   hm_pool = '//black/case/bugfix_pool/hmbin';
%else
pool
   cm_pool = '//black/case/cmbin';
   hm_pool = '//black/case/hmbin';
%endif
```

When the DSEE engineers were preparing to build Version 2.1, they set their model thread to one like this:

```
-target bugfix_pool
/v2_bugfix -when_exists
dsee!//orange/case/releases/v2
```

The first rule in this thread ensured that all the builds produced by their **build** command with the **-force_all** option wouldn't contend for space in the pools that other group members might be using for base level releases or other work and possibly cause the removal of the other group members' builds from the pools.

Appendix A

CAD Tools Project System Model

Below is an abbreviated version of the system model used by our company's CAD tools group. Chapter 2 discusses the CAD tools project's use of the DSEE environment in detail.

```
model  CAD =

  title    'CAD SOURCE BUILD';
  system   '//MAX/CAD/SRC/SYSTEM';
  shell    '/com/sh';


 {declare all libraries}
  library
          help_lib          =  '//max/cad/doc/help';
          user_lib          =  '//max/cad/doc/user';
          ins_lib           =  '//max/cad/src/ins';
          build_lib         =  '//max/cad/src/build';
          tests_lib         =  '//max/cad/src/tests';
          scripts_lib       =  '//max/cad/src/scripts';
          library_lib       =  '//max/cad/src/library';
          database_lib      =  '//max/cad/src/database';
          utilities_lib     =  '//max/cad/src/utilities';
          applications_lib  =  '//max/cad/src/applications';

  pool cad_pool = '//max/cad/src/cad_pool';
```

```
{set  all DEFAULTS}
{-------------------------------------------------------}
 default for ?* =
    use_pool cad_pool;
    end of ?*;

 default for ?*.hlp_src =
    @ help_lib;
    depends_tools
        '//max/cad/src/build/fmt';
    depends_source
        setup_help_macros.fmt @ help_lib;
    translate
      # This translation rule creates links to formatted
      # help files in a directory called help.  If the
      # directory doesn't exist, it generates an error
      # message.
      #
      //max/cad/src/build/fmt %SOURCE -out %RESULT.HLP
      eon
      IF existf help THEN @
        /com/crl help/%source({?*}.hlp_src, @1.hlp, %leaf) @
          %result.hlp -r
      ELSE wd  >?/dev/null | readln where
           args "WARNING: ^where/help does not exist - link "
           args "%source({?*}.hlp_src, @1.hlp, %leaf) not "
           args "created"
      ENDIF
      %done;
  end of ?*.hlp_src;

 default for ?*.ins.pas =
    @ ins_lib;
   end of ?*.ins.pas;

 default for ?*.ins.ftn =
    @ ins_lib;
   end of ?*.ins.ftn;


default FOR ?*.pas =
   depends_tools
     '//max/cad/src/build/pas';
   translate
     //max/cad/src/build/pas %source @
       %cr_opt(-dba) %option(-dbs) %option(-comchk) @
       %option(-subchk) %option(-opt) %option(-nopt) @
       %option(-l, -l %result) -b %result
     %done;
   end of ?*.pas;
```

```
default for ?*.ftn =
  depends_tools
    '//max/cad/src/build/ftn';
  translate
    //max/cad/src/build/ftn %source @
      %cr_opt(-dba) %option(-dbs) %cr_opt(-i*2) @
      %option(-i*4)  %cr_opt(zero) %option(-subchk) @
      %option(-opt) %option(-nopt) @
     %option(-l, -l %result) -b %result
    %done;
  end of ?*.ftn;

default for ?*.ash =
  @ scripts_lib;
  depends_tools
    ['/com/cpf'];
  translate
    # This translation rule copies Shell scripts into the
    # binary pool and then creates links to them in the com
    # directory.
    #
    /com/cpf %source %result
    eon
    IF existf com THEN /com/crl @
       com/%source({?*}.ash, %leaf) %result -r
    ELSE wd  >?/dev/null | readln where
         args "WARNING: ^where/com does not exist - link "
         args "%source({?*}.ash, %leaf) not created"
    ENDIF
  %done;
end of ?*.ash;
{-------------------------------------------------------------}

  use_pool cad_pool;
  nil_translation;
{
    NIL_TRANSLATION for the top-level component ensures
    that there's no single top-level derived object.
}
  depends_result

{-------------------------------------------------------------}
{ cad shell scripts }
    element cad_csr.ash;
    element create_design_database.ash;
    element expand_des.ash;
    element inlib_rcom.ash;


{-------------------------------------------------------------}
{help files}
    element create_design_database.hlp_src;
    element inlib_rcom.hlp_src;
    element cad_csr.hlp_src;
    element commands.hlp_src;
    element expand_des.hlp_src;
```

```
{-------------------------------------------------------------}
{utilities}
    element writeshort.pas @ utilities_lib =
      depends_source
        writeshort.ins.pas;
    end of writeshort.pas;


    element args_tester.pas @ utilities_lib =
      depends_source
        ['/sys/ins/base.ins.pas'];
        ['/sys/ins/pgm.ins.pas'];
        ['/sys/ins/error.ins.pas'];
        ['/us/ins/cl.ins.pas'];
        writeshort.ins.pas;
        args.ins.pas;
    end of args_tester.pas;

    element exor.ftn @ utilities_lib;

    element equ_string.pas @ utilities_lib =
      depends_source
        ['/sys/ins/base.ins.pas'];
        constants.ins.pas;
        equal_string.ins.pas;
    end of equ_string.pas;

    element upper_case.pas @ utilities_lib =
      depends_source
        ['/sys/ins/base.ins.pas'];
        upper_case.ins.pas;
    end of upper_case.pas;

    element banner.pas @ utilities_lib =
      depends_source
        ['/sys/ins/error.ins.pas'];
        ['/sys/ins/cal.ins.pas'];
        ['/sys/ins/vfmt.ins.pas'];
        ['/sys/ins/type_uids.ins.pas'];
        ['/us/ins/ms.ins.pas'];
        ['/us/ins/mst.ins.pas'];
        ['/us/ins/objmod.ins.pas'];
        ['/us/ins/name.ins.pas'];
        ['/us/ins/ubase.ins.pas'];
        banner.ins.pas ;
    end of banner.pas;

  element left_just.pas @ utilities_lib =
      depends_source
        left_just.ins.pas;
  end of left_just.pas;


  element qsort.pas @ utilities_lib =
      depends_source
        qsort.ins.pas;
  end of qsort.pas;
```

```
element overlap_rect.pas @ utilities_lib =
    depends_source
      ['/sys/ins/base.ins.pas'];
      constants.ins.pas;
      overlap_rectangles.ins.pas;
end of overlap_rect.pas;

element val_real.pas @ utilities_lib =
    depends_source
      ['/sys/ins/base.ins.pas'];
      constants.ins.pas;
      val_real.ins.pas;
end of val_real.pas;

element verify.pas @ utilities_lib =
    depends_source
      ['/sys/ins/base.ins.pas'];
      ['/sys/ins/pgm.ins.pas'];
      ['/sys/ins/error.ins.pas'];
      ['/sys/ins/vfmt.ins.pas'];
      ['/us/ins/cl.ins.pas'];
      verify.ins.pas;
end of verify.pas;

element bin_tree.pas @ utilities_lib =
   depends_source
     ['/us/ins/ubase.ins.pas'];
     ['/us/ins/rws.ins.pas'];
     ['/us/ins/strl.ins.pas'];
     ['/us/ins/bin_tree.ins.pas'];
end of bin_tree.pas;

element net.pas @ utilities_lib =
   declare_only;
   depends_source
     ['/sys/ins/base.ins.pas'];
     ['/sys/ins/vfmt.ins.pas'];
     db_design.ins.pas;
     constants.ins.pas;  {Common defs}
     utilities.ins.pas;  {Utility procedures}
     design_interface.ins.pas;
     net.ins.pas; {PROCEDURE defs (this module) }
end of net.pas;
```

```
element component.pas @ utilities_lib =
   declare_only;
   depends_source
     ['/sys/ins/base.ins.pas'];
     db_design_and_library.ins.pas;
     constants.ins.pas;          {Misc. constants}
     utilities.ins.pas;          {Utility procedures}
     design_interface.ins.pas;   {Defs for design interface}
     pin.ins.pas;                {Pin utility procedures}
     net.ins.pas;                {Net utility procedures}
     library_general.ins.pas;
     component.ins.pas;          {PROCEDURE defs (this mod) }
end of component.pas;


element pin.pas @ utilities_lib =
   declare_only;
   depends_source
     ['/sys/ins/base.ins.pas'];
     db_design_and_library.ins.pas;
     constants.ins.pas;          {Misc. constants}
     design_interface.ins.pas;   {Defs for design interface}
     net.ins.pas;                {Net utility procedures}
     pin.ins.pas;                {PROCEDURE defs (this mod)}
     utilities.ins.pas;          {Utility procedures}
     library_general.ins.pas;    {Defs for lib interface}
end of pin.pas;

element gate.pas @ utilities_lib =
   declare_only;
   depends_source
     ['/sys/ins/base.ins.pas'];
     ['/sys/ins/vfmt.ins.pas'];
     ['/sys/ins/cal.ins.pas'];
     ['/sys/ins/time.ins.pas'];
     bin_tree.ins.pas;
     db_design_and_library.ins.pas;
     constants.ins.pas;          {Misc. constants}
     utilities.ins.pas;          {Utility procedures}
     design_interface.ins.pas;   {Defs for design interface}
     pin.ins.pas;                {Pin utility procedures}
     net.ins.pas;                {Net utility procedures}
     library_general.ins.pas;
     component.ins.pas;          {component design interface }
     design_general.ins.pas;
     gate.ins.pas;
     gate_intern.ins.pas;
end of gate.pas;
```

```
element default_pwr_gnd_names.pas @ utilities_lib =
   declare_only;
   depends_source
     ['/sys/ins/base.ins.pas'];
     db_design_and_library.ins.pas;
     constants.ins.pas;
     utilities.ins.pas;
     design_interface.ins.pas;
     library_general.ins.pas;
     pin.ins.pas;
     default_pwr_gnd_names.ins.pas;
end of default_pwr_gnd_names.pas;

element design_general.pas @ utilities_lib =
   declare_only;
   depends_source
     ['/sys/ins/base.ins.pas'];
     ['/sys/ins/name.ins.pas'];
     ['/sys/ins/pm.ins.pas'];
     db_design.ins.pas;
     constants.ins.pas;          {Misc. constants}
     utilities.ins.pas;          {Utility procedures}
     design_interface.ins.pas; {Defs for design interface}
     component.ins.pas;          {component design interface }
     design_general.ins.pas;
end of design_general.pas;


aggregate utilities =
   { This aggregate is a combination of utilities required
     by several other components.  The individual
     utilities themselves are declared above.
   }
   declare_only;
   translate
      //max/cad/src/build/lbr -create %result.lbr -<<!
      %result_of(?*.pas).bin
      %result_of(?*.ftn).bin
      !
   %done;
   depends_result
      args_tester.pas;
      banner.pas;
      bin_tree.pas;
      exor.ftn;
      left_just.pas;
      equ_string.pas;
      overlap_rect.pas;
      qsort.pas;
      upper_case.pas;
      val_real.pas;
      verify.pas;
      writeshort.pas;
end of utilities;
```

```
aggregate design_interface =
   declare_only;
   translate
      //max/cad/src/build/lbr -create %result.lbr -<<!
      %result_of(?*.pas).bin
      !
   %done;
   depends_result
      component.pas;
      pin.pas;
      net.pas;
      gate.pas;
      design_general.pas;
end of design_interface;


aggregate list_nets =
   translate
      //max/cad/src/build/bind -b %result -<<!
      %result_of(?*.pas).bin
      %result_of(utilities).lbr
      !
    eon
    IF existf com THEN /com/crl com/list_nets  %result -r
    ELSE wd  >?/dev/null | readln where
         args "WARNING: ^where/com does not exist - "
         args "link list_nets not created"
    ENDIF
   %done;


   depends_result
     element list_nets.pas @ utilities_lib =
         depends_source
           ['/sys/ins/base.ins.pas'];
           ['/sys/ins/pgm.ins.pas'];
           ['/sys/ins/error.ins.pas'];
           ['/sys/ins/ms.ins.pas'];
           ['/sys/ins/name.ins.pas'];
           ['/sys/ins/streams.ins.pas'];
           db_design.ins.pas;
           constants.ins.pas;
           design_interface.ins.pas;
           banner.ins.pas;
           writeshort.ins.pas;
           equal_string.ins.pas;
     end of list_nets.pas;
     db_interface.pas;
     init_database.pas;
     utilities;
end of list_nets;
```

```
{-------------------------------------------------------}
{database}
  element db_interface.pas @ database_lib =
    depends_source
      ['/us/ins/ubase.ins.pas'];
      ['/sys/ins/streams.ins.pas'];
      ['/sys/ins/error.ins.pas'];
      ['/sys/ins/pgm.ins.pas'];
      ['/sys/ins/vec.ins.pas'];
      ['/us/ins/file.ins.pas'];
      ['/us/ins/name.ins.pas'];
      ['/us/ins/fu.ins.pas'];
      ['/us/ins/ms.ins.pas'];
      ['/us/ins/mst.ins.pas'];
      ['/us/ins/asknode.ins.pas'];
      upcase.ins.pas;
      writeshort.ins.pas;
      db_interface.ins.pas;
  end of db_interface.pas;

  element init_database.pas @ database_lib =
    depends_source
      ['/sys/ins/base.ins.pas'];
      db_interface.ins.pas;
      design_database_define.ins.pas;
  end of init_database.pas;

  element init_library_database.pas @ database_lib =
    depends_source
      ['/sys/ins/base.ins.pas'];
      db_interface.ins.pas;
      library_database_define.ins.pas;
  end of init_library_database.pas;


  aggregate db_$create =
    translate
      //max/cad/src/build/bind <<!
      %result_of(%.pas).bin
      %result_of(utilities).lbr
      -b %result
      !
      eon
      IF existf com THEN /com/crl com/db_$create %result -r
      ELSE wd  >?/dev/null | readln where
         args "WARNING: ^where/com does not exist - "
         args "link db_$create not created"
      ENDIF
    %done;
```

```
depends_result
  element db_$create.pas @ database_lib =
     depends_source
       ['/sys/ins/base.ins.pas'];
       ['/sys/ins/streams.ins.pas'];
       ['/sys/ins/ms.ins.pas'];
       ['/sys/ins/name.ins.pas'];
       ['/sys/ins/pgm.ins.pas'];
       ['/sys/ins/error.ins.pas'];
       db_interface.ins.pas;
       banner.ins.pas;
       writeshort.ins.pas;
       upcase.ins.pas;
    end of db_$create.pas;
    db_interface.pas;
    utilities;
  end of db_$create;

  aggregate initialize_design_database =
    translate
      //max/cad/src/build/bind <<!
      %result_of(%.pas).bin
      %result_of(utilities).lbr
      -b %result
      !
      eon
      IF existf com THEN /com/crl @
         com/initialize_design_database   %result -r
      ELSE wd  >?/dev/null | readln where
           args "WARNING: ^where/com does not exist - "
           args "link initialize_design_database not created"
      ENDIF
    %done;


depends_result
   element initialize_design_database.pas @ database_lib =
     depends_source
       ['/sys/ins/base.ins.pas'];
       ['/sys/ins/pgm.ins.pas'];
       db_design.ins.pas;
       banner.ins.pas;
    end of initialize_design_database.pas;
    utilities;
    db_interface.pas;
    init_database.pas;
  end of initialize_design_database;


  aggregate db_$gen_type_ins =
    { this aggregate defines the tool db_$gen_type_ins }
    declare_only;
    translate
      //max/cad/src/build/bind <<!
      %result_of(%.pas).bin
      -b %result
      !
    %done;
```

```
 depends_result
    element db_$gen_type_ins.pas @ database_lib =
      make_visible;
      depends_source
        ['/sys/ins/base.ins.pas'];
        ['/sys/ins/name.ins.pas'];
        ['/sys/ins/pgm.ins.pas'];
        ['/sys/ins/error.ins.pas'];
        banner.ins.pas;
    end of db_$gen_type_ins.pas;
    banner.pas;
  end of db_$gen_type_ins;

element cad.sch  @ database_lib =
  make_visible;
  { This Element is temporarily visible outside of the
    configuration manager so that Elements like
    design_database_define.ins.pas can reference it in their
    code.
  }
  depends_tools
     ['/com/chpat'];
     ['/com/cpf'];
  translate
     { This translation rule uses the binary produced by
       translating the Aggregate db_$gen_type_ins to translate
       the schema file cad.sch.  Then it edits with ed and
       chpat and creates a link.
      }
     %result_of(db_$gen_type_ins) %source @
         %result'des_type.ins.pas'


  #
  # Change the name of the following because of DSEE
  # limitation of 16 characters for the user extension
  # (Code ref. follows)
  # design_type.ins.pas  to  des_type.ins.pas
  #   '$(cad.sch)des_type.ins.pas'
  # design_database_define.base  to des_define.base
  #   '$(cad.sch)des_define.base'
  # design_database_extern.base    to des_extern.base
  #   '$(cad.sch)des_extern.base'
  # init_database.ins.pas    to  init_des.ins.pas
  #   '$(cad.sch)init_des.ins.pas'
  #
```

```
    ed -n %result'des_type.ins.pas'  <<!
    /*)/
    .+1 i
    %%IFDEF NOT design_db_types_are_defined %%THEN
    %%VAR design_db_types_are_defined;
    %%ENDIF
    .
    w
    q
    !
     /com/cpf %source %result.sch
     # design_database_define.base
     /com/chpat -o <%source >%result'des_define.base' @
       "%% *RECORD *= *{[~.]*}.?*" @
       '@1 @: Define db_$Record_type_identifier;'
     /com/chpat -o <%source >>%result'des_define.base' @
       "%% *FIELD *= *{[~.]*}.?*" @
       '@1 @: Define db_$Field_Type_identifier;'
     /com/chpat -o <%source >>%result'des_define.base' @
       "%% *SET *= *{[~.]*}.?*" @
       '@1 @: Define db_$SET_Type_identifier;'
     # design_database_extern.base
     /com/chpat <%result'des_define.base' @
       >%result'des_extern.base' 'Define' 'Extern'
     # init_database.ins.pas
     /com/chpat -o <%source >%result'init_des.ins.pas' @
       "%% *RECORD *= *{[~.]*}.?*" @
       'db_$get_record_type_id (database_id, @'@1@',@1);'
     /com/chpat -o <%source >>%result'init_des.ins.pas' @
        "%% *FIELD *= *{[~.]*}.?*" @
        'db_$get_field_type_id (database_id, @'@1@',@1);'
     /com/chpat -o <%source >>%result'init_des.ins.pas' @
        "%% *SET *= *{[~.]*}.?*" @
        'db_$get_set_type_id (database_id, @'@1@',@1);'
      eon
      IF existf com THEN /com/crl com/cad.sch %result.sch -r
      ELSE wd  >?/dev/null | readln where
           args "WARNING: ^where/com does not exist - "
           args "link cad.sch not created"
      ENDIF
  %done;
  depends_result
      db_$gen_type_ins;
end of cad.sch;

element library.sch  @ database_lib =
  make_visible;
  depends_tools
     ['/com/chpat'];
     ['/com/cpf'];
  translate
   %result_of(db_$gen_type_ins) %source @
          %result'lib_type.ins.pas'
```

```
#
# Change the name of the following because of DSEE
# limitation of 16 characters for the user extension
# library_type.ins.pas  to  lib_type.ins.pas
#   '$(library.sch)lib_type.ins.pas'
# library_database_define.base  to lib_define.base
#   '$(library.sch)lib_define.base'
# library_database_extern.base    to lib_extern.base
#   '$(library.sch)lib_extern.base'
# init_library_database.ins.pas    to  init_lib.ins.pas
#   '$(library.sch)init_lib.ins.pas'
#
  ed -n %result'lib_type.ins.pas'  <<!
  /*)/
  .+1 i
   %%IFDEF NOT library_db_types_are_defined %%THEN
   %%VAR library_db_types_are_defined;
   %%ENDIF
  .
  w
  q
  !
   /com/cpf %source %result.sch
   # library_database_define.base
   /com/chpat -o <%source >%result'lib_define.base' @
      "%% *RECORD *= *{[~.]*}.?*" @
      '@1 @: Define db_$Record_type_identifier;'
   /com/chpat -o <%source >>%result'lib_define.base' @
      "%% *FIELD *= *{[~.]*}.?*" @
      '@1 @: Define db_$Field_Type_identifier;'
   /com/chpat -o <%source >>%result'lib_define.base' @
      "%% *SET *= *{[~.]*}.?*" @
      '@1 @: Define db_$SET_Type_identifier;'
   # library_database_extern.base
   /com/chpat <%result'lib_define.base' @
      >%result'lib_extern.base' 'Define' 'Extern'
   # init_library_database.ins.pas
   /com/chpat -o <%source >%result'init_lib.ins.pas' @
      "%% *RECORD *= *{[~.]*}.?*" 'db_$get_record_type_id @
      (library_database_id, @'@1@',@1);'
   /com/chpat -o <%source >>%result'init_lib.ins.pas' @
      "%% *FIELD *= *{[~.]*}.?*" 'db_$get_field_type_id @
      (library_database_id, @'@1@',@1);'
   /com/chpat -o <%source >>%result'init_lib.ins.pas' @
      "%% *SET *= *{[~.]*}.?*" 'db_$get_set_type_id @
      (library_database_id, @'@1@',@1);'
    eon
```

```
      IF existf com THEN /com/crl com/library.sch @
           %result.sch -r
      ELSE wd  >?/dev/null | readln where
           args "WARNING: ^where/com does not exist - "
           args "link library.sch not created"
      ENDIF
  %done;
  depends_result
      db_$gen_type_ins;
end of library.sch;

{-------------------------------------------------------------}
{library}
  element library_general.pas @library_lib =
     depends_source
        ['/sys/ins/base.ins.pas'];
        db_library.ins.pas;
        constants.ins.pas;
        upper_case.ins.pas;
        library_general.ins.pas;
  end of library_general.pas;

  element join_library.pas @ library_lib =
     depends_source
        ['/sys/ins/base.ins.pas'];
        db_design_and_library.ins.pas;
        design_interface.ins.pas;
        library_general.ins.pas;
        component.ins.pas;
        join_library.ins.pas;
  end of join_library.pas;

{-------------------------------------------------------------}
{inserts}
  element db_design.ins.pas =
    { This is one of several insert files used by other
      components.  Note that it is only declared here so that
      it can be widely used, and that its dependencies are
      promoted to be direct dependencies of all referencing
      components.
}
    declare_only;
    promote_depends;
    depends_source
      design_database.ins.pas;
      db_interface.ins.pas;
  end of db_design.ins.pas;

  element design_database.ins.pas =
    declare_only;
    promote_depends;
    depends_source
      design_database_declare.ins.pas;
    depends_result
       cad.sch;
  end of design_database.ins.pas;
```

```
element design_database_define.ins.pas =
  declare_only;
  promote_depends;
  depends_source
    design_database_declare.ins.pas;
  depends_result
    cad.sch;
end of design_database_define.ins.pas;

element db_library.ins.pas =
  declare_only;
  promote_depends;
  depends_source
    library_database.ins.pas;
    db_interface.ins.pas;
end of db_library.ins.pas;

element library_database.ins.pas =
  declare_only;
  promote_depends;
  depends_source
    library_database_declare.ins.pas;
  depends_result
     library.sch;
end of library_database.ins.pas;

element library_database_define.ins.pas =
  declare_only;
  promote_depends;
  depends_source
    library_database_declare.ins.pas;
  depends_result
    library.sch;
end of library_database_define.ins.pas;

element db_design_and_library.ins.pas =
  declare_only;
  promote_depends;
  depends_source
    db_interface.ins.pas;
    design_database.ins.pas;
    library_database.ins.pas;
end of db_design_and_library.ins.pas;

element utilities.ins.pas =
   declare_only;
   promote_depends;
   depends_source
      args.ins.pas;
      banner.ins.pas;
      upcase.ins.pas;
      upper_case.ins.pas;
      writeshort.ins.pas;
      equal_string.ins.pas;
end of utilities.ins.pas;
```

```
{-------------------------------------------------------}
{applications}
  element asi_conv.pas @ applications_lib =
     depends_source
       ['/sys/ins/base.ins.pas'];
       ['/sys/ins/vfmt.ins.pas'];
       db_design.ins.pas;
  end of asi_conv.pas;

  aggregate load_cad =
    translate
      //max/cad/src/build/bind <<!
      %result_of(?*.pas).bin
      %result_of(utilities).lbr
      %result_of(design_interface).lbr
      -b %result
      !
      eon


      IF existf com THEN /com/crl com/load_cad   %result -r
      ELSE wd  >?/dev/null | readln where
           args "WARNING: ^where/com does not exist - "
           args "link load_cad not created"
      ENDIF
    %done;
    depends_result
    element load_cad.pas @ applications_lib =
      depends_source
        ['/sys/ins/base.ins.pas'];
        ['/sys/ins/pgm.ins.pas'];
        ['/sys/ins/error.ins.pas'];
        ['/sys/ins/vfmt.ins.pas'];
        db_design_and_library.ins.pas;
        banner.ins.pas;
        writeshort.ins.pas;
        join_library.ins.pas;
        design_interface.ins.pas;
        library_general.ins.pas;
        component.ins.pas;
        args.ins.pas;
        pin.ins.pas;
        gate.ins.pas;
        net.ins.pas;
        asi_conv.ins.pas;
        design_general.ins.pas;
    end of load_cad.pas;
    utilities;
    db_interface.pas;
    init_database.pas;
    init_library_database.pas;
    join_library.pas;
    library_general.pas;
    asi_conv.pas;
    design_interface;
  end of load_cad;
```

```
aggregate rcom_cad =
   translate
      eon
      /com/crl /arc ~/arc -r
      args "/ARC link now points to ~/ARC
      /com/ld //max/cad/arc -ld -c -nwarn >?/dev/null | @
      WHILE READLN arc_rev DO
        /com/crl ~/arc /cad/arc/^arc_rev -r
        /com/ld -ll -nhd -lt -en ~/arc
        //max/cad/src/build/bind -b %result.^arc_rev @
          -nound -<<!
            ~/arc/sys/lib/dfi.bin
            %result_of(rcom_to_cad.pas).^arc_rev.bin
            %result_of(db_interface.pas).bin
            %result_of(init_database.pas).bin
            %result_of(init_library_database.pas).bin
            %result_of(join_library.pas).bin
            %result_of(library_general.pas).bin
            %result_of(utilities).lbr
            %result_of(design_interface).lbr
            %result_of(default_pwr_gnd_names.pas).bin
      !
      eon


      IF existf com THEN /com/crl @
         com/rcom_cad.^arc_rev   %result.^arc_rev -r
      ELSE wd  >?/dev/null | readln where
         args "WARNING: ^where/com does not exist - "
         args "link rcom_cad.^arc_rev not created"
      ENDIF
      args "bind with arclib (test only) and "
      args "check for undefined globals"
      args "'Attempt to respecify start addr' message is ok"
      //max/cad/src/build/bind %result.^arc_rev @
         ~/arc/arclib
      ENDDO
      RETURN -T
   %done;
   depends_result
      element rcom_to_cad.pas @ applications_lib =
        depends_tools
          '//max/cad/src/build/pas';
```

```
        translate
          eon
          /com/crl /arc ~/arc -r
          args "/ARC link now points to ~/ARC"
          /com/ld //max/cad/arc -ld -c >?/dev/null | @
          WHILE READLN arc_rev DO
            /com/crl ~/arc //max/cad/arc/^arc_rev -r
              /com/ld -nhd -ll -lt -en ~/arc
              //max/cad/src/build/pas %source @
              %cr_opt(-dba) %option(-dbs) @
              %option(-comchk) %option(-subchk) @
              %option(-opt) %option(-nopt) @
             %option(-l, -l %result.^arc_rev) @
              -b %result.^arc_rev
          ENDDO
          RETURN -T
        %done;


        depends_source
           ['/sys/ins/base.ins.pas'];
           ['/sys/ins/vfmt.ins.pas'];
           ['/sys/ins/pfm.ins.pas'];
           ['/sys/ins/pgm.ins.pas'];
           ['/sys/ins/error.ins.pas'];
           db_design_and_library.ins.pas;
           constants.ins.pas;
           design_interface.ins.pas;
           library_general.ins.pas;
           utilities.ins.pas;
           pin.ins.pas;
           net.ins.pas;
           gate.ins.pas;
           join_library.ins.pas;
           default_pwr_gnd_names.ins.pas;
           design_general.ins.pas;
      end of rcom_to_cad.pas;


      db_interface.pas;
      init_database.pas;
      init_library_database.pas;
      join_library.pas;
      library_general.pas;
      design_interface;
      utilities;
      default_pwr_gnd_names.pas;

  end of rcom_cad;

end of cad;
```

Appendix B

OS Project System Model

This appendix presents an abbreviated version of the system model used by the OS engineers at Apollo. For more information on the OS project, see Chapter 3.

```
%var OPSYS1 OPSYS2
%if not (OPSYS1 or OPSYS2) %then
   %error 'You must have a -TARGET rule in your @
       model thread'
    %exit
%endif

%if OPSYS1 %then
   model operating_system1 =
   alias
     os = '1';
     asmname = 'asm';
     asmoptnuc = '-ndb -config os';
     asmoptker = '-ndb -config os apollo_%exp(os)';
     kerbin = 'bin%exp(os)';
     nucbin = 'bin';
     pasname = 'pas';
     pasoptnuc = '-cpu any -talign -info 3 -config os';
```

```
%elseif OPSYS2 %then
   model operating_system2 =
   alias
     os = '2';
     asmname = 'asm';
     asmoptnuc = '-ndb -config os';
     asmoptker = '-ndb -config os apollo_%exp(os)';
     kerbin = 'bin%exp(os)';
     nucbin = 'bin';
     pasname = 'pas';
     pasoptnuc = '-cpu any -talign -info 3 -config os';
%endif

{ common aliases }

     asm = '//opera/tools/%exp(asmname)';
     pas = '//opera/tools/%exp(pasname)'
title
    'Operating System %exp(os)';
system
    '//opera/op_sys/opsys%exp(os)';


pool
    opsys_pool =
        '//opera/op_sys/pools/sr20.bl002/opsys%exp(os)';
    nuc_pool =
        { Both of the listed physical pools are searched for
          usable derived objects when the configuration
          manager builds components for nuc_pool.  If no
          suitable substitute exists, however, the
          configuration manager puts the new derived objects
          in the physical pool listed first.
        }
        '//opera/op_sys/pools/sr20.bl002/bin',
        '//opera/op_sys/pools/sr20.bl001/bin';
     ker_pool =
         '//opera/op_sys/pools/sr20.bl002/bin%exp(os)';
         '//opera/op_sys/pools/sr20.bl001/bin%exp(os)';

library
    nuc             = '//opera/op_sys/nuc';
    ker             = '//opera/op_sys/ker';
    ins             = '//opera/op_sys/ins';
    kins            = '//opera/op_sys/kins';
    com             = '//opera/op_sys/scripts';

shell
    '/com/sh';
```

```
{ Defaults for all Pascal source modules }
default for %.pas =
   depends_source
        base.ins.pas @ ins;
        force_build.pas @ nuc;
     depends_tools
        [´%exp(pas)´];
        [crll @ com];
end;

{ Defaults for all ASM modules }
default for %.asm =
     depends_source
        base.ins.asm @ ins;
        force_build.asm @ nuc;
     depends_tools
        [´%exp(asm)´];
        [crll @ com];
end;

use_pool
     opsys_pool;


depends_tools
     [´/com/bind´];
     [´//opera/op_sys/rfc´];
     [´//opera/op_sys/rm´];
     [make_build_time @ com];
     rfc_image @ com;

{ Source dependencies for top level build ("make_build_time")
}
depends_source
     kernel.ins.pas @ kins;
     base.ins.pas @ ins;
     print_build_time.pas @ ker;
     get_build_time.ins.pas @ ins;
     md_if.ins.pas @ kins;
     build_string @ com;

translate
     # This translation rule calls a script to give the
     # operating system a time stamp.  Then it binds all the
     # system components.
     #
     von; eon; abtsev -p

     //opera/op_sys/scripts/make_build_time %exp(os) @
                                %result.bldt.ins.pas @
                                %result.bldt.asm @
                                %result.pbldt.bin @
                                %result.bldt.bin @
```

```
#
##### bind
#
bind -sys -b %result.bin -map >%result.bmap * <<!
{
 All result dependencies are mentioned by name in
 the following list instead of being referenced
 by a wildcard (e.g. %RESULT_OF(?*.bin)) in order to
 optimize the bind sequence and achieve better
 performance.
}

%result_of(crash_record.asm).bin
%result.bldt.bin
%result_of(addr.asm).bin
%result_of(disk_buffers.asm).bin
%result_of(ast.pas).bin
%result_of(chksum.pas).bin
%result_of(crash_system.asm).bin
%result_of(iodefs.asm).bin
%result_of(io_tbls.asm).bin
%result_of(io_wired.pas).bin
%result_of(net_io.pas).bin
%result_of(netbuf.pas).bin
%result_of(netlog_asm.asm).bin
%result_of(network.pas).bin
%result_of(gpu_asm.asm).bin
%result_of(stacks.asm).bin
%result_of(uid_$hash.asm).bin
%result_of(uid_list.asm).bin
%result_of(ioinit.pas).bin
%result.pbldt.bin
%result_of(svc_catcher_cm.asm).bin
%result_of(uid.pas).bin
%result_of(acl.pas).bin
%result_of(ringlog.pas).bin
%result_of(io.pas).bin
%result_of(dtty.pas).bin
%result_of(dtty_asm.asm).bin
%result_of(color.pas).bin
%result_of(gpu.pas).bin
%result_of(lpr.pas).bin
%result_of(cbuf.pas).bin
%result_of(netlog_start_addr.asm).bin
%result_of(netlog.pas).bin
%result_of(netlog_end_addr.asm).bin
%result_of(buffer_pages.asm).bin
-und
-end
!
```

```
if eqs %cr_opt(-xref) then
    #
    # The rfc script, used below, creates a special
    # object file format for the operating system.
    #
    //opera/op_sys/tools/rfc %result.bin %result.rfc @
        >%result.rmap  <<!
-d absolute     0
-d trap_page    0
-d prot         0
-d ptt          $700000
-d crash_record $E00000
-d cold         $101400 $101400
-d dump2        *page

%if OPSYS1 %then
-d dump         $E00400 $102000
%else
-d dump         $e00400 $101c00
%endif

-d os_proc      *page
-d wired_proc   *
-d wired_data   *
-d net_port_table *
-d rem_file_$data *
-d page_init    *page
-d os_init_proc *page
-d os_init_data *
-d procedure$   *segment
-d proc2_create_proc *
-d proc2_delete_proc *
-d os_proc_end  *
-d os_data      *page
-d data$        *
-d proc2_create_data *
-d proc2_delete_data *
-d iic_aqwrd_proc *page
-d iic_aqwrd_data *
-d rtwired_code *page
-d rtwired_data *
-d mt_page      *page
-d pbu_page     *page
-d pbu_wired_proc *
-d pbu_wired_data *
-d reloc        *
```

```
%if OPSYS2 %then
    -reloc          * * *
%endif

    -d acl_$data    *
    -d file_$lot_data *
    -d ringlog_$data *
    -d proc2_$data  *
    -d os_data_end  *
    -d os_page      *segment
    -d page         *
    -d os_page_end  *
    -d unwired_stacks *page
    -d disk_buffers *page
    -d pbu_tables   *page
    -d os_buffers_end *page
    -d netpool      *page
    -d vm_tables    *page
    -d iodefs       $FA0000     { must match mst_seg_high!! }
    -m
    -u
    -end
    !


        # The rm sript, used below, builds a map of the
        # system.
        #
        args ^dsee_full_build_name @
          | edstr "s/{?*}@n/Build ID: @1/" >%result.map
        //opera/op_sys/tools/rm %result.rfc %result.bmap @
          %result.rmap >>%result.map
    endif


    #
    # These files can be deleted once used.
    #
    dlf %result.bldt.ins.pas @
        %result.bldt.asm @
        %result.pbldt.bin @
        %result.bldt.bin
%done;

depends_result

(   {************************************************
     ****************** N U C ***********************
     ************************************************ }

    default for ?* =
        @ nuc;
        use_pool
            nuc_pool;
    end;
```

```
default for %.pas =
    translate
             %exp(pas) %source -b %result  @
            -l %result.lst %exp(pasoptnuc) @
            -abs -xrs -ndb -comchk -opt -align -exp @
            -idir //opera/op_sys %option(-map) @
            %cr_opt(-peb) %option(-dbs) %cr_opt(-dba) @
            %option(-extra) %cr_opt(-cpu)  @
            %cr_opt(-cond) %cr_opt(-subchk) @
            %option(-warn)
    %done;
end;

default for %.asm =
    translate
             %exp(asm) %source -b %result @
            -l %result.lst -idir //opera/op_sys @
            %exp(asmoptnuc)
    %done;
end;


element acl.pas =
    depends_source
        uid.ins.pas @ ins;
        file.ins.pas @ ins;
        mst.ins.pas @ ins;
        rem_file.ins.pas @ ins;
        ml.ins.pas @ ins;
        proc1.ins.pas @ ins;
        network.ins.pas @ ins;
        acl.ins.pas @ ins;
        dbuf.ins.pas @ ins;
        ml.ins.pas @ ins;
        route.ins.pas @ ins;
        iic.ins.pas @ ins;
end;

element addr.asm =
    depends_source
        os_or_sau.ins.asm @ ins;
end;


element crash_record.asm;

element disk_buffers.asm;

element dtty_asm.asm =
    depends_source
        os_or_sau.ins.asm @ ins;
end;


element io_wired.pas =
    depends_source
        io.ins.pas @ ins;
end;
```

```
    element netlog.pas =
        depends_source
            proc1.ins.pas @ ins;
            mmap.ins.pas @ ins;
            mst.ins.pas @ ins;
            time.ins.pas @ ins;
            io.ins.pas @ ins;
            netbuf.ins.pas @ ins;
            pkt.ins.pas @ ins;
            ring.ins.pas @ ins;
            netlog.ins.pas @ ins;
            net_io.ins.pas @ ins;
    end;

    element netlog_asm.asm =
        depends_source
            os_or_sau.ins.asm @ ins;
    end;

    element netlog_end_addr.asm;

    element netlog_start_addr.asm;

    element ringlog.pas =
        depends_source
            io.ins.pas @ ins;
            wp.ins.pas @ ins;
            network.ins.pas @ ins;
            ring.ins.pas @ ins;
            ringlog.ins.pas @ ins;
            pkt.ins.pas @ ins;
    end;

    element uid.pas =
        depends_source
            time.ins.pas @ ins;
            time.pvt.pas @ ins;
            uid.ins.pas @ ins;
    end;

    element uid_$hash.asm =
        depends_source
            os_or_sau.ins.asm @ ins;
    end;

    element uid_list.asm =
        depends_source
            os_or_sau.ins.asm @ ins;
    end;
);
```

```
(   {************************************************
     ****************** K E R ***********************
     ************************************************ }
    default for ?* =
        @ ker;
        use_pool
            ker_pool;
    end;

    default for %.pas =
        depends_source
            kernel.ins.pas @ kins;
        translate
                %exp(pas) %source  -b %result @
                -l %result.lst -xrs -ndb -comchk -opt -nalign @
                -cpu any -exp -idir //opera/op_sys @
                -config os apollo%exp(os) %option(-map)  @
                %cr_opt(-peb)  %option(-dbs) %cr_opt(-dba) @
                %option(-extra) %cr_opt(-cpu)  @
                %cr_opt(-cond) %cr_opt(-subchk) @
                %option(-warn)
        %done;
    end;

    default for %.asm =
        depends_source
            kernel.ins.asm @ kins;
        translate
                %exp(asm) %source -b %result @
                -l %result.lst -idir //opera/op_sys @
                %exp(asmoptker)
        %done;
    end;

    element ast.pas =
        depends_source
            mmu.ins.pas @ ins;
            mmu.kins.pas @ kins;
            mmap.ins.pas @ ins;
            vol.ins.pas @ ins;
            mst.pvt.pas @ kins;
            ast.ins.pas @ ins;
            ast.pvt.pas @ ins;
            network_page.ins.pas @ ins;
    end;

    element buffer_pages.asm =
        depends_source
            cpu.ins.asm @ kins;
    end;
```

```
element cbuf.pas =
    depends_source
        ec.ins.pas @ ins;
        fim.ins.pas @ ins;
        proc1.ins.pas @ ins;
        cbuf.ins.pas @ ins;
end;

element chksum.pas =
    depends_source
        os_or_sau.ins.pas @ ins;
        vm.ins.pas @ kins;
        mmap.ins.pas @ ins;
        mmap.pvt.pas @ kins;
        mmu.ins.pas @ ins;
        mmu.kins.pas @ kins;
end;

element color.pas =
    depends_source
        proc1.ins.pas @ ins;
        mst.ins.pas @ ins;
        wp.ins.pas @ ins;
        vm.ins.pas @ kins;
        mmu.ins.pas @ ins;
        mmu.kins.pas @ kins;
        io.ins.pas @ ins;
        smdu.ins.pas @ ins;
        iomap.ins.pas @ kins;
        vector.ins.pas @ kins;
        ec.ins.pas @ ins;
        time.ins.pas @ ins;
        testpage.ins.pas @ ins;
        color.ins.pas @ ins;
        color_nuc.ins.pas @ kins;
        color_ops.ins.pas @ kins;
        vme.ins.pas @ kins;
end;

element crash_system.asm =
    depends_source
        vm.ins.asm @ kins;
        iodefs_a.ins.asm @ kins;
        cregs.ins.asm @ kins;
        md_if.ins.asm @ kins;
        color4_regs.ins.asm @ ins;
        cpu.ins.asm @ kins;
        smd.ins.asm @ kins;
end;
```

```
element dtty.pas =
    depends_source
        proc1.ins.pas @ ins;
        vfmt.ins.pas @ ins;
        ec.ins.pas @ ins;
        fim.ins.pas @ ins;
        term.ins.pas @ ins;
        term.pvt.pas @ kins;
        smd.ins.pas @ ins;
        color.ins.pas @ ins;
        color2.ins.pas @ ins;
        color_nuc.ins.pas @ kins;
        smdu.ins.pas @ ins;
        color_ops.ins.pas @ kins;
        dtty.ins.pas @ ins;
end;

element gpu_asm.asm;

element gpu.pas =
    depends_source
        vol.ins.pas @ ins;
        file.ins.pas @ ins;
        ml.ins.pas @ ins;
        mmap.ins.pas @ ins;
        mmu.ins.pas @ ins;
        mst.ins.pas @ ins;
        proc1.ins.pas @ ins;
        proc2.ins.pas @ ins;
        time.ins.pas @ ins;
        wp.ins.pas @ ins;
        fim.ins.pas @ ins;
        vm.ins.pas @ kins;
end;

element io.pas =
    depends_source
        kernel.ins.pas @ kins;
        name.ins.pas @ ins;
        proc1.ins.pas @ ins;
        pbu.ins.pas_at_os_ins;
        network.ins.pas @ ins;
        cal.ins.pas @ ins;
        route.ins.pas @ ins;
        mt.ins.pas @ ins;
        io.ins.pas @ ins;
end;

element iodefs.asm =
    depends_source
        iodefs_a.ins.asm @ kins;
end;
```

```
element ioinit.pas =
    depends_source
        iomap.ins.pas @ kins;
        vector.ins.pas @ kins;
        dma.ins.pas @ kins;
        mmu.ins.pas @ ins;
        io.ins.pas @ ins;
        ct.ins.pas @ ins;
        ml.ins.pas @ ins;
        dbuf.ins.pas @ ins;
end;

element io_tbls.asm =
    depends_source
        vector.ins.asm @ kins;
        disktape.pvt.asm @ kins;
        ring.pvt.asm @ kins;
        pbu.ins.asm @ kins;
        disk.pvt.asm @ ins;
        iodefs_a.ins.asm @ kins;
end;

element lpr.pas =
    depends_source
        name.ins.pas @ ins;
        pbu.ins.pas_at_os_ins;
        pbu.pvt.pas @ kins;
        cbuf.ins.pas @ ins;
        mst.ins.pas @ ins;
        wp.ins.pas @ ins;
        time.ins.pas @ ins;
        proc1.ins.pas @ ins;
        proc2.ins.pas @ ins;
        testpage.ins.pas @ ins;
        lpr.ins.pas @ ins;
end;

element netbuf.pas =
    depends_source
        vm.ins.pas @ kins;
        mmu.ins.pas @ ins;
        mmu.kins.pas @ kins;
        mmap.ins.pas @ ins;
        wp.ins.pas @ ins;
        vol.ins.pas @ ins;
        network.ins.pas @ ins;
        netbuf.ins.pas @ ins;
end;
```

```
element network.pas =
    depends_source
        vm.ins.pas @ kins;
        os_or_sau.ins.pas @ ins;
        proc1.ins.pas @ ins;
        vol.ins.pas @ ins;
        file.ins.pas @ ins;
        ast.ins.pas @ ins;
        win.ins.pas @ins;
        io.ins.pas @ ins;
        wp.ins.pas @ ins;
        mmap.ins.pas @ ins;
        mmap.pvt.pas @ kins;
        pmap.ins.pas @ ins;
        time.ins.pas @ ins;
        ec.ins.pas @ ins;
        ml.ins.pas @ ins;
        vfmt.ins.pas @ ins;
        network.ins.pas @ ins;
        network_page.ins.pas @ ins;
        cal.ins.pas @ ins;
        rem_file.ins.pas @ ins;
        netbuf.ins.pas @ ins;
        ring.ins.pas @ ins;
        pkt.ins.pas @ ins;
        net_io.ins.pas @ ins;
end;

element net_io.pas =
    depends_source
        io.ins.pas @ ins;
        iomap.ins.pas @ kins;
        iic.ins.pas @ ins;
        ml.ins.pas @ ins;
        mmap.ins.pas @ ins;
        mmap.pvt.pas @ kins;
        netbuf.ins.pas @ ins;
        netlog.ins.pas @ ins;
        network.ins.pas @ ins;
        pkt.ins.pas @ ins;
        proc1.ins.pas @ ins;
        ring.ins.pas @ ins;
        ringlog.ins.pas @ ins;
        route.ins.pas @ ins;
        sock.ins.pas @ ins;
        time.ins.pas @ ins;
        net_io.ins.pas @ ins;
end;

element stacks.asm;
```

```
    element svc_entries_cm @ kins =
        make_visible;
        depends_tools
            ['//opera/op_sys/tools/preproc'];
            ['//opera/op_sys/tools/svc_prep'];
        depends_source
            kernel.ins.asm @ kins;


        translate
            eon
            //opera/op_sys/tools/preproc @
                -config os apollo_%exp(os) <%source @
                | /opera/op_sys/tools/svc_prep @
                >%result.codes.ins.asm @
                -svc %result.lib.ins.asm
        %done;
    end;

    element svc_catcher_cm.asm =
        depends_source
            vm.ins.asm @ kins;
            fault.ins.asm_at_os_ins;
        depends_result
            svc_entries_cm;
    end;
);


    {************************************************
     ****************** I N S ***********************
     ************************************************ }

 %include '//opera/op_sys/os_ins.ins.sml';


    {************************************************
     ****************** K I N S *********************
     ************************************************ }
%include '//opera/op_sys/os_kins.ins.sml';

end {of opsysN};
```

Appendix C

DSEE Project System Models

The DSEE project uses two system models: one to build the DSEE command facility, and one to build the system model compiler. Abbreviated versions of both models are presented here. For details on the DSEE engineering project and how these two system models are related to one another, read Chapter 4.

Both models use **%include** directives to incorporate model fragments in them. These two fragments, **dsee_default_trans.ins.sml** and **dsee_common.ins.sml**, are presented following the text of the system model root fragments.

DSEE Command Facility System Model

```
%var bl1_pool bugfix_pool
model dsee =
system "//orange/case/dsee";
title "Domain Software Engineering Environment System Model";

library
   case_1  = '//orange/case/case_1';
   case_cm = '//orange/case/case_cm';
```

```
{ We use different pool declarations for forced builds, so
  that developmental builds don't get bumped out of the pool
}
%if bl1_pool %then
  pool
    cm_pool = '//black/case/bl1_pool/cmbin';
    hm_pool = '//black/case/bl1_pool/hmbin';
%elseif bugfix_pool %then
  pool
    cm_pool = '//black/case/bugfix_pool/cmbin';
    hm_pool = '//black/case/bugfix_pool/hmbin';
%else
  pool
    cm_pool = '//black/case/cmbin';
    hm_pool = '//black/case/hmbin';
%endif


shell "/com/sh";

default for ?* =
   use_pool
      cm_pool;
end;

default for ?*.pas =
    @ case_cm;
end;

{ Note how we make default dependencies be noncritical so that
  major rebuilds are not required when the dependency changes.
  If a change makes a global rebuild necessary, use
  BUILD -FORCE_ALL.
}

%include '//orange/case/case_cm/dsee_default_trans.ins.sml';
default for %.pas =
   depends_tools
      ['/com/pas'];
   depends_source
      [khronos_global_types.ins.pas @ case_1];
      ['/sys/ins/base.ins.pas'];


   depends_result
      [dpm.msg];
end of %.pas;

default for ?*.msg =
    @ case_cm;
end;


default for ?*.asm =
    @ case_cm;
end;
```

```
use_pool
   cm_pool;

{ Here's the translation rule for DSEE Model block. It creates
  a version number for the resultant code, binds it and the
  other parts of the software, and creates a link to the
  software in the working directory.
}
translate
   //orange/case/tools/dsee_version_info.sh
   cpf //orange/case/precomp/dsee_version_info.pas @
      %result.version.pas
   /com/pas %result.version.pas -b %result.version
   bind -b %result -nomes - <<!
      %result_of(%.pas).bin
      %result_of(%.msg).bin
      %result_of(%.asm).bin
      %result_of(outside_bins)
      %result_of(khr_db)
      %result_of(not_cm)
      %result.version.bin
      //orange/case/bin/current_dsee_license.bin
   !
   /com/dlf %result.version.?* -nq
   if eqs %option(-nl) then
      crl xcase %result -r
      args "link xcase created in your working directory"
   endif
%done;

   depends_tools
       {tools for dsee itself}
       dsee_version_info.sh @ tools_lib;
       string_length.sh @ tools_lib;

{ The structure of the DSEE software is as follows.  First,
  there are the DSEE builder, MAKE_MODEL, and the code that
  creates an environment; these are built as part of the
  system, but they aren't bound in with the managers.
  Following these three Aggregates in the system model are the
  elements that constitute the configuration manager and three
  other Aggregates:  outside_bins, which is a bound collection
  of binaries from outside the DSEE project; khr_db, the
  history database; and not_cm, which represents all other
  code (including the model fragment dsee_common.ins.sml,
  which contains Elements that are shared with the
  system model compiler).  The configuration manager elements
  and these three Aggregates are bound together by the Model
  block's translation rule.}
```

```
depends_result
   aggregate dsee_builder =
     default for ?*.pas =
         @ case_cm;
         end of ?*.pas;

     translate
         /com/bind -b %result %result_of(?*.pas).bin
         %done;


     depends_result
         element spab.pas =
             depends_source
                 rock_settings.ins.pas;
                 spab_utl.ins.pas;
                 bldcom_to_spab.ins.pas;
         end of spab.pas;

         element spab_utl.pas =
             depends_source
                 spab_utl.ins.pas;
         end of spab_utl.pas;

         rock_settings.pas;
   end of dsee_builder;

   aggregate make_model =
       translate
           /com/cpf %result_of(make_model.pas).bin %result
       %done;
       depends_result
           element make_model.pas @ tools_lib;
   end of make_model;

   aggregate create_env_shell =
     default for ?*.pas =
         @ case_cm;
         depends_source
             rock_settings.ins.pas;
     end of ?*.pas;

     translate
        /com/bind -b %result - <<!
         %result_of(?*).bin
         !
         %done;


     depends_result
         element create_env_shell.pas =
             depends_source
                 dsee_to_create_env_shell.ins.pas;
         end of create_env_shell.pas;

         rock_settings.pas;
   end of create_env_shell;
```

```
element dem.msg @ case_cm =
   make_visible;

   depends_source
       bld_dem.sh @ tools_lib;

   translate
       //orange/case/tools/bld_dem.sh %result
       %done;
end of dem.msg;

element dpm.msg =
   make_visible;

   depends_source
       bld_dpm.sh @ tools_lib;

   translate
      //orange/case/tools/bld_dpm.sh %result.bin
   %done;
end of dpm.msg;

element khronos_utl.ins.pas @ case_1 =
    promote_depends; {nested include files}

    depends_source
        [hf_sm_utl.ins.pas @ case_1];
        [sort_utl.ins.pas @ case_1];
end of khronos_utl.ins.pas;

element case_hm_$data_move.asm @ case_hm;

aggregate outside_bins =
    depends_tools
        { We make these tools dependencies so that
          their version stamps are recorded in BCTs
        }
        '/usx/lib/remote/remote.bin';
        '/usx/lib/user_info/user_info.bin';
        '//ylang/bind/objio.bin';
        '//orange/case/sf_ipc/sflib_no_user_info';

    translate
        /com/bind -b %result - <<!
         /usx/lib/remote/remote.bin
         /usx/lib/user_info/user_info.bin
         //ylang/bind/objio.bin
         //orange/case/sf_ipc/sflib_no_user_info
         %result_of(case_hm_$data_move.asm).bin
         %result_of(khr_xsm.pas).bin
        !
    %done;
```

```
        depends_result
            element khr_xsm.pas @ case_hm =
                depends_source
                    khr_xsm.ins.pas @ case_hm;
                    case_hm_$hidden.ins.pas @ case_hm;
            end of khr_xsm.pas;

            case_hm_$data_move.asm;
    end of outside_bins;


    element bld.pas =
        depends_source
            khr_fault.ins.pas @ case_1;
            [khronos_global_data.ins.pas @ case_1];
            khronos_cl.ins.pas @ case_1;
            cm_global_types.ins.pas;
            cm_global_data.ins.pas;
            bp.ins.pas;
            bld_global.ins.pas;
            bld_pass1.ins.pas;
            bld_pass2.ins.pas;
            bld_pass3.ins.pas;
            bld_pass4.ins.pas;
            bld_prev_bct_utl.ins.pas;


            bld_utl.ins.pas;
            bld_blessing.ins.pas;
            bld_previous.ins.pas;
            bld.ins.pas;
            [khronos_utl.ins.pas @ case_1];
            lastbld.ins.pas;
        end;

    element bld_previous.pas =
        depends_source
            cm_global_types.ins.pas;
            sm.ins.pas;
            fsm.ins.pas;
            bitvector_utl.ins.pas;
            bld_prev_bct_types.ins.pas;
            bld_utl.ins.pas;
            bld_previous.ins.pas;
        end;
```

```
element brm_text_utl.pas =
   depends_source
      [khronos_global_data.ins.pas @ case_1];
      khr_fault.ins.pas @ case_1;
      cm_global_types.ins.pas;
      fsm.ins.pas;
      bct_hashman.ins.pas;
      bitvector_utl.ins.pas;
      sm.ins.pas;
      cm_utl.ins.pas;
      ct.ins.pas;
      common_bct.ins.pas;
      vbct.ins.pas;
      brm_text.ins.pas;
      brm_text_internal.ins.pas;
      [khronos_utl.ins.pas @ case_1];
   end;

element cm_sys.pas =
   depends_source
      [khronos_global_data.ins.pas @ case_1];
      khronos_cl.ins.pas @ case_1;
      khr_fault.ins.pas @ case_1;
      khr_lib.ins.pas @ case_1;
      ctm_tasklist.ins.pas @ case_1;
      ctm_task.ins.pas @ case_1;
      session_memory.ins.pas @ case_1;
      cm_global_types.ins.pas;
      sm_driver.ins.pas;
      sm_switch.ins.pas;
      cmsc.ins.pas;
      bp.ins.pas;
      cm_sys.ins.pas;
      [khronos_utl.ins.pas @ case_1];
   end;

element lastbld.pas =
   depends_source
      [khronos_global_data.ins.pas @ case_1];
      [khronos_utl.ins.pas @ case_1];
      khr_fault.ins.pas @ case_1;
      cm_global_types.ins.pas;
      sm.ins.pas;
      cm_utl.ins.pas;
      common_bct.ins.pas;
      vbct.ins.pas;
      bct_utl.ins.pas;
      bp.ins.pas;
      lastbld.ins.pas;
   end;
```

```
element create_env.pas =
   depends_source
       [khronos_global_data.ins.pas @ case_1];
       [khronos_utl.ins.pas @ case_1];
       khr_fault.ins.pas @ case_1;
       cm_global_types.ins.pas;
       cm_global_data.ins.pas;
       bitvector_utl.ins.pas;
       cm_utl.ins.pas;
       bp.ins.pas;
       common_bct.ins.pas;
       vbct.ins.pas;
       bct_stringtab.ins.pas;
       bct_hashman.ins.pas;
       create_env.ins.pas;
       rock_settings.ins.pas;
       dsee_to_create_env_shell.ins.pas;
end;

element crc.asm;


element ct.pas =
   depends_source
      cm_global_types.ins.pas;
      cm_global_data.ins.pas;
      bitvector_utl.ins.pas;
      sm.ins.pas;
      [khronos_global_data.ins.pas @ case_1];
      ct.ins.pas;
      mct.ins.pas;
      mct_utl.ins.pas;
      cm_utl.ins.pas;
      khr_environment.ins.pas @ case_1;
      [khronos_utl.ins.pas @ case_1];
   end;

element data_zero.asm;

element fsm.pas =
   depends_source
      [khronos_global_data.ins.pas @ case_1];
      khr_fault.ins.pas @ case_1;
      cm_global_types.ins.pas;
      fsm.ins.pas;
      [khronos_utl.ins.pas @ case_1];
   end;

element rock_settings.pas =
   depends_source
       rock_settings.ins.pas;
   end;
```

```
      element rm.pas =
         depends_source
            [khronos_global_data.ins.pas @ case_1];
            [khronos_utl.ins.pas @ case_1];
            khr_fault.ins.pas @ case_1;
            khronos_cl.ins.pas @ case_1;
            sm.ins.pas;
            bitvector_utl.ins.pas;
            cm_utl.ins.pas;
            common_bct.ins.pas;
            vbct.ins.pas;
            cm_global_types.ins.pas;
            cmsc.ins.pas;
            rm_utl.ins.pas;
            rm.ins.pas;
         end;


{ The following Aggregate builds all the non-configuration
  management managers (e.g. history manager, task manager)
}
      aggregate not_cm =
         default for ?* =
            use_pool
               hm_pool;
         end;

         default for ?*.pas =
            @ case_1;
         end;

         default for ?*.asm =
            @ case_1;
         end;

         use_pool
            hm_pool;

         translate
            /com/bind -b %result -nomes -noundefined - <<!
            %result_of(?*.pas).bin
            %result_of(?*.asm).bin
            !
            %done;

         depends_result
            element ctm_edit_task.pas =
               depends_source
                  [khronos_global_data.ins.pas @ case_1];
                  khronos_cl.ins.pas @ case_1;
                  ctm_edit_util.ins.pas;
                  ctm_edit_task.ins.pas;
                  [khronos_utl.ins.pas @ case_1];
               end;
```

```
element ctm_task.pas =
   depends_source
      [khronos_global_data.ins.pas @ case_1];
      khronos_cl.ins.pas @ case_1;
      ctm_edit_utl.ins.pas;
      khr_fault.ins.pas @ case_1;
      case_messages.ins.pas @ case_1;
      ctm_included_on.ins.pas;
      ctm_tasklist.ins.pas;
      ctm_edit_task.ins.pas;
      ctm_task_msgs.ins.pas;
      khr_db.ins.pas;
      lsc.ins.pas;
      khr_semaphore.ins.pas;
      khr_recovery.ins.pas;
      protect.ins.pas;
      session_memory.ins.pas;
      ctm_task.ins.pas;
      [khronos_utl.ins.pas @ case_1];
   end;

element ctm_tasklist.pas =
   depends_source
      case_messages.ins.pas;
      [khronos_global_data.ins.pas @ case_1];
      khronos_cl.ins.pas @ case_1;
      khr_lib.ins.pas;
      khr_fault.ins.pas @ case_1;
      ctm_edit_utl.ins.pas;
      ctm_included_on.ins.pas;
      cqm.ins.pas;
      protect.ins.pas;
      ctm_task.ins.pas;
      ctm_task_msgs.ins.pas;
      ctm_tasklist.ins.pas;
      ctm_tasklist_msgs.ins.pas;
      lsc.ins.pas;
      session_memory.ins.pas;
      [khronos_utl.ins.pas @ case_1];
      cm_global_types.ins.pas @ case_cm;
      bct_hashman.ins.pas @ case_cm;
   end;

element dem_utl.pas =
   depends_source
      [khronos_global_data.ins.pas @ case_1];
      [khronos_utl.ins.pas @ case_1];
   end;

element find_help.pas =
   depends_source
      [khronos_global_data.ins.pas @ case_1];
      find_help.ins.pas;
      [khronos_utl.ins.pas @ case_1];
   end;
```

```
element hf_sm_utl.pas =
   depends_source
      lsc.ins.pas;
      [hf_sm_utl.ins.pas @ case_1];
end;

element his_utl.pas =
   depends_source
      [khronos_global_data.ins.pas @ case_1];
      khr_db.ins.pas;
      khronos_cl.ins.pas @ case_1;
      khr_fault.ins.pas @ case_1;
      khr_semaphore.ins.pas;
      khr_show.ins.pas;
      case_messages.ins.pas;
      khr_msgs.ins.pas;
      lsc.ins.pas;
      khr_recovery.ins.pas;
      cmm.ins.pas;
      cmm_show.ins.pas;
      recov_utl.ins.pas;
      his_utl.ins.pas;
      case_hm_$hidden.ins.pas @ case_hm;
      [khronos_utl.ins.pas @ case_1];
   end;

element khronos.pas =
   depends_source
      [khronos_global_data.ins.pas @ case_1];
      khronos_cl.ins.pas @ case_1;
      khr_lib.ins.pas;
      khr_show.ins.pas;
      khr_his.ins.pas;
      khr_users.ins.pas;
      khr_recovery.ins.pas;
      ctm_tasklist.ins.pas;
      ctm_task.ins.pas;
      khr_db.ins.pas;
      protect.ins.pas;
      session_memory.ins.pas;
      cmm_show.ins.pas;
      cfm_forms.ins.pas;
      khr_version.ins.pas;
      khr_environment.ins.pas;
      cm_global_types.ins.pas @case_cm;
      cm_global_data.ins.pas @ case_cm;
      sm.ins.pas @ case_cm;
      cm_sys.ins.pas @ case_cm;
      sm_switch.ins.pas @ case_cm;
      bp_commands.ins.pas @ case_cm;
      [khronos_utl.ins.pas @ case_1];
      sm_init.ins.pas @ case_cm;
      create_env.ins.pas @ case_cm;
      ct_commands.ins.pas @ case_cm;
      ct_driver.ins.pas @ case_cm;
      rm.ins.pas @ case_cm;
   end;
```

```
            element khr_environment.pas =
               depends_source
                  [khronos_global_data.ins.pas @ case_1];
                  khr_fault.ins.pas;
                  khr_db.ins.pas;
                  khr_environment.ins.pas;
                  case_hm_$user_calls.ins.pas @ case_hm;
                  cm_global_types.ins.pas @ case_cm;
                  sm.ins.pas @ case_cm;
                  common_bct.ins.pas @ case_cm;
                  vbct.ins.pas @ case_cm;
                  bct_hashman.ins.pas @ case_cm;
                  cm_utl.ins.pas @ case_cm;
                  rock_settings.ins.pas @ case_cm;
                  rs_utl.ins.pas @ case_cm;
                  bitvector_utl.ins.pas @ case_cm;
                  [khronos_utl.ins.pas @ case_1];
               end;

            element msg_text.pas =
               depends_source
                  [khronos_utl.ins.pas @ case_1];
               end;

            element sort_utl.pas =
               depends_source
                  [khronos_global_data.ins.pas @ case_1];
                  [khronos_utl.ins.pas @ case_1];
               end;

            element compare_lines.asm =
               depends_source
                  [khronos_utl.ins.pas @ case_1];
            end;

            element khronos_utl_asm.asm =
               depends_source
                  [khronos_utl.ins.pas @ case_1];
            end;

      %include '//orange/case_cm/dsee_common.ins.sml';
            end; { of not_cm }

      end of dsee;
```

DSEE System Model Compiler System Model

```
model system_model_compiler =
    title 'System Model for the DSEE System Model Compiler';

system '//blue/smc/smcsys';
```

```
library
        smclib = '//blue/smc/smclib';
        case_1 = '//orange/case/case_1';
        case_cm = '//orange/case/case_cm';
        lang = '//ylang/lang/source';

%if bl1_pool %then
  pool
    cm_pool = '//black/case/bl1_pool/cmbin';
    hm_pool = '//black/case/bl1_pool/hmbin';
%elseif bugfix_pool %then
  pool
    cm_pool = '//black/case/bugfix_pool/cmbin';
    hm_pool = '//black/case/bugfix_pool/hmbin';
%else
  pool
    cm_pool = '//black/case/cmbin';
    hm_pool = '//black/case/hmbin';
%endif
    smc_pool = '//orange/case/smcbin.new';

shell '/com/sh';

{ Default translation rules are defined in the following
  fragment
}
%include '//orange/case_cm/dsee_default_trans.ins.sml';

    default for ?* =
        @ smclib;
        use_pool smc_pool;
        end of ?*;

    default for %.pas =


    depends_tools
        ['/com/bind'];


    use_pool smc_pool;
```

```
translate
    # generate build time stamp
    eon
    date | readln smc_date
    catf <<! >smc_bldt.pas
    module smc_bldt;
    var smc_bldt : extern string;
    var smc_bldt_len : extern integer;
    define smc_bldt := ^smc_date,
           smc_bldt_len := sizeof (^smc_date);
    !
    /com/pas smc_bldt
    #
    # bind all pieces together.
    #
    /com/bind - <<!
    %result_of(?*.pas).bin
    %result_of(sml.cln)/symstrings.bin
    %result_of(sml.cln)/prodtostr.bin
    %result_of(sml.cln)/bast.bin
    smc_bldt.bin
    -b %result
    -end
    !
    # once the system's built, the time stamp's no longer
    # necessary.  It is deleted below.
    #
    dlf smc_bldt.pas smc_bldt.bin
    #
    # The following link allows the person building the
    # compiler to test the build in his/her own working
    # directory.
    #
    crl system_model_compiler %RESULT -r
    args "Link SYSTEM_MODEL_COMPILER created "
    args "in your working directory"
    %done;

depends_result
    { This section builds macros and nested include files
      on which other components depend.  Note the use of
      the DECLARE_ONLY declarations as well as MACRO and
      PROMOTE_DEPENDS.
    }
    aggregate case_global_types =
        declare_only;
        macro;
        depends_source
            [khronos_global_types.ins.pas @ case_1];
            [cm_global_types.ins.pas @ case_cm];
        end of case_global_types;
```

```
aggregate smc_data_structures =
    { major data structures; these files must
      be included as a group
    }
    declare_only;

    macro;

    depends_source
        smc_global_types.ins.pas;
        fest_utl.ins.pas;
        list_utl.ins.pas;
        ast.ins.pas;
    end of smc_data_structures;


element khronos_utl.ins.pas @ case_1 =
    declare_only;

    promote_depends;

    depends_source
        [hf_sm_utl.ins.pas @ case_1];
        [sort_utl.ins.pas @ case_1];
    end of khronos_utl.ins.pas;

{ Below are the buildable components of the system
  model.  Their results are bound together by the
  Model block's translation rule.
}


element sml.cln =
    depends_source
        { This Element serves as initial input to
          a series of parser generator tools. The
          results of the translation rule are several
          skeleton files plus some include files
          describe the system model grammar.  The
          include files are required by several
          other elements (e.g. recognizer.pas);
          however, since the elements requiring the
          include files aren't owned by this project,
          they can't include environment variable
          references to the preprocessed include files
          Therefore, this Element's translation rule
          creates links to the include files.  The
          elements requiring the include files use the
          link names to reference them.
        }
        application.ins.pas;
        recognizer.ins.pas;
        bast.ins.pas;
        { included by bast.ins.pas for bast.pas: }
        case_global_types;
        smc_data_structures;
```

```
depends_tools
    ['//orange/case/fest/fest'];
    ['//orange/case/fest/lexgen'];
    ['//orange/case/fest/pargen'];

translate
    # Because this translation generates so many
    # files, %RESULT is actually a directory to
    # hold them all.
    #
    crd %result
    args "run fest..."
    //orange/case/fest/fest %source @
        -out %result @
        -inc //blue/smc/smclib/bast.ins.pas


    args "Run LEXGEN ..."
    //orange/case/fest/lexgen %result/sml @
        -out %result

    args "Run PARGEN ..."
    if not //orange/case/fest/pargen %result/sml @
     -out %result %option(-debug) then
        cpf %result/sml.prs -r
        args "Search for 'conflict' in SML.PRS "
        args "in your working directory."
        return -e
        endif

    args "Compile generated sources ..."
    crl appl$ //blue/smc/smclib -r
    crl fest$ //blue/smc/smclib -r
    crl pool$ %result -r
    #
    # Now the include files are compiled
    #
    /com/pas %result/bast %option(-dbs) @
        -b %result/bast -idir //blue/smc/smclib
    /com/pas %result/symstrings @
        -b %result/symstrings
    /com/pas %result/prodtostr @
        -b %result/prodtostr
    dll appl$ pool$ fest$
    %done;

{ Make results temp visible so that other
  Elements' translation rules can reference via
  environment variables (to make sure
  MAKE_VISIBLE works)
}
make_visible;
end of sml.cln;
```

```
(   { All modules in these parentheses make include
      references to files generated from SML.CLN.
      Therefore, this default declaration contains a
      result dependency on smc.cln.
    }
    default for %.pas =
        depends_source
            application.ins.pas;
            recognizer.ins.pas;

        translate
            # Create links mentioned above. Note the
            # of the environment variable, possible
            # because of the MAKE_VISIBLE declaration
            # above.
            #
            crl pool$ $@(sml.cln@) -r
            crl appl$ //blue/smc/smclib -r
            crl fest$ //blue/smc/smclib -r
            /com/pas %source -comchk -opt @
                %option(-dbs) -b %result
            dll appl$ pool$ fest$
            %done;

        depends_result
            sml.cln;
        end of %.pas;

    element fest_utl.pas =
        depends_source
            fest_utl.ins.pas;
        end of fest_utl.pas;

    element recognizer.pas =
        translate
            crl pool$ $@(sml.cln@) -r
            crl appl$ //blue/smc/smclib -r
            crl fest$ //blue/smc/smclib -r
            /com/pas %source -comchk -opt @
                %option(-dbs) -config blank_lines @
                -b %result
            dll appl$ pool$ fest$
            %done;
        end of recognizer.pas;
```

```
    element prepdvr.pas =
        depends_source
            [condcomp.ins.pas @ lang];

        translate
            crl pool$ $@(sml.cln@) -r
            crl appl$ //blue/smc/smclib -r
            crl fest$ //blue/smc/smclib -r
            /com/pas %source -comchk -opt @
                %option(-dbs) @
                -config using_condcomp blank_lines @
                -b %result
            dll appl$ pool$ fest$
            %done;
        end of prepdvr.pas;

    element logerrors.pas;
    );

element smc_main.pas;


element smc.pas =
    depends_source
        case_global_types;
        [khronos_utl.ins.pas @ case_1];
        [fsm.ins.pas @ case_cm];
        sm.ins.pas @ case_cm;
        [condcomp.ins.pas @ lang];
        smc_data_structures;
        smc.ins.pas;
        symbol.ins.pas;
        listing.ins.pas;
        semantic.ins.pas;
        component.ins.pas;
        build_rule.ins.pas;
    end of smc.pas;

element ast.pas =
    depends_source
        case_global_types;
        [khronos_utl.ins.pas @ case_1];
        [bct_stringtab.ins.pas @ case_cm];
        smc_data_structures;
        smc.ins.pas;
        semantic.ins.pas;
        symbol.ins.pas;
        build_rule.ins.pas;
    end of ast.pas;
```

```
element semantic.pas =
    depends_source
        case_global_types;
        [khronos_utl.ins.pas @ case_1];
        [cmsc.ins.pas @ case_cm];
        sm.ins.pas @ case_cm;
        smc_data_structures;
        smc.ins.pas;
        component.ins.pas;
        semantic.ins.pas;
        symbol.ins.pas;
    end of semantic.pas;

element symbol.pas =
    depends_source
        case_global_types;
        [khronos_utl.ins.pas @ case_1];
        [fsm.ins.pas @ case_cm];
        [bct_stringtab.ins.pas @ case_cm];
        [bct_hashman.ins.pas @ case_cm];
        sm.ins.pas @ case_cm;
        smc_data_structures;
        smc.ins.pas;
        symbol.ins.pas;
        component.ins.pas;
    end of symbol.pas;

element component.pas =
    depends_source
        case_global_types;
        [khronos_utl.ins.pas @ case_1];
        [fsm.ins.pas @ case_cm];
        [bitvector_utl.ins.pas @ case_cm];
        sm.ins.pas @ case_cm;
        smc_data_structures;
        smc.ins.pas;
        symbol.ins.pas;
        component.ins.pas;
    end of component.pas;

element build_rule.pas =
    depends_source
        case_global_types;
        [khronos_utl.ins.pas @ case_1];
        [fsm.ins.pas @ case_cm];
        [bitvector_utl.ins.pas @ case_cm];
        sm.ins.pas @ case_cm;
        smc_data_structures;
        smc.ins.pas;
        symbol.ins.pas;
        component.ins.pas;
        brr.ins.pas;
        build_rule.ins.pas;
    end of build_rule.pas;
```

```
element brr.pas =
    depends_source
        case_global_types;
        [khronos_global_data.ins.pas @ case_1];
        [khronos_utl.ins.pas @ case_1];
        [khr_fault.ins.pas @ case_1];
        [bitvector_utl.ins.pas @ case_cm];
        brm_text_internal.ins.pas @ case_cm;
        smc_global_types.ins.pas;
        build_rule.ins.pas;
        brr.ins.pas;
    end of brr.pas;

element listing.pas =
    depends_source
        case_global_types;
        [khronos_utl.ins.pas @ case_1];
        [fsm.ins.pas @ case_cm];
        [bitvector_utl.ins.pas @ case_cm];
        sm.ins.pas @ case_cm;
        smc_data_structures;
        smc.ins.pas;
        symbol.ins.pas;
        component.ins.pas;
        listing.ins.pas;
    end of listing.pas;


element binary.pas =
    depends_source
        case_global_types;
        [khronos_utl.ins.pas @ case_1];
        [fsm.ins.pas @ case_cm];
        [bct_stringtab.ins.pas @ case_cm];
        [bitvector_utl.ins.pas @ case_cm];
        sm.ins.pas @ case_cm;
        sm_object.ins.pas @ case_cm;
        smc_data_structures;
        smc.ins.pas;
        symbol.ins.pas;
        component.ins.pas;
        binary.ins.pas;
    end of binary.pas;


element list_utl.pas =
    depends_source
        list_utl.ins.pas;
    end of list_utl.pas;

element smc_fault.pas =
    depends_source
        [khronos_global_types.ins.pas @ case_1];
        [khr_fault.ins.pas @ case_1];
    end of smc_fault.pas;
```

```
        element condcomp.pas @ lang =
            depends_source
                [condcomp.ins.pas @ lang];

            translate
                crl condcomp.ins.pas @
                   //ylang/lang/source/condcomp.ins.pas -r
                /com/pas %source -comchk -opt @
                   %option(-dbs) -b %result
                dll condcomp.ins.pas
                %done;
            end of condcomp.pas;

{ the following fragment contains elements shared with
  dsee.sml
}
%include '//orange/case_cm/dsee_common.ins.sml';

    end of system_model_compiler;
```

Model Fragment dsee_default_trans.ins.sml

```
{ DSEE_DEFAULT_TRANS.INS.SML
  System model fragment for common default translation rules.
}

default for %.pas =
   depends_tools
      '/com/pas';
    translate
      /com/pas %source -comchk -b %result @
          %option(-nwarn) %option(-ninfo) %option(-nopt) @
          %cr_opt(-config) %option(-exp) %option(-pic) @
%ifdef bl_pool %then
          %cr_opt(-dbs)  %cr_opt(-nb)
%else
          %option(-dbs)  %option(-nb)
%endif
      %done;
end of %.pas;

default for ?*.asm =
   depends_tools
       '/com/asm';
   translate
      /com/asm %source -nl -b %result
      %done;
end;
```

Model Fragment dsee_common.ins.sml

```
{ DSEE_COMMON.INS.SML
  System model fragment for things used in both dsee.sml and
  smc.sml. }
{ This default grouping for things in CM_POOL }
(  default for ?* =
      use_pool cm_pool;
      @ case_cm;
      end of ?*;
   default for %.pas =
      depends_source
         [khronos_global_types.ins.pas @ case_1];
      end of %.pas;


     element bct_hashman.pas =
        depends_source
           [khronos_global_data.ins.pas @ case_1];
           khr_fault.ins.pas @ case_1;
           cm_global_types.ins.pas;
           fsm.ins.pas;
           bct_stringtab.ins.pas;
           bct_hashman.ins.pas;
           [khronos_utl.ins.pas @ case_1];
        end;

      element bct_stringtab.pas =
        depends_source
           [khronos_global_data.ins.pas @ case_1];
           khr_fault.ins.pas @ case_1;
           cm_global_types.ins.pas;
           fsm.ins.pas;
           bct_hashman.ins.pas;
           sm.ins.pas;
           cm_utl.ins.pas;
           bct_stringtab.ins.pas;
           [khronos_utl.ins.pas @ case_1];
        end;

      element bitvector_utl.pas =
        depends_source
           cm_global_types.ins.pas;
           fsm.ins.pas;
           bitvector_utl.ins.pas;
           [khronos_utl.ins.pas @ case_1];
        end;
```

```
element khronos_global_data.pas =
   depends_source
      [khronos_global_data.ins.pas @ case_1];
      session_memory.ins.pas;
      ctm_tasklist.ins.pas;
      cm_global_types.ins.pas @ case_cm;
      cm_global_data.ins.pas @ case_cm;
      cm_sys.ins.pas @ case_cm;
      [khronos_utl.ins.pas @ case_1];
    end;

element khronos_utl.pas =
   depends_source
      [khronos_global_data.ins.pas @ case_1];
      khronos_cl.ins.pas @ case_1;
      khr_fault.ins.pas @ case_1;
      comment_pvt.ins.pas @ case_1;
      lsc.ins.pas;
      protect.ins.pas;
      khr_db.ins.pas;
      [khronos_utl.ins.pas @ case_1];
   end;


element khronos_utl2.pas =
   depends_source
      [khronos_global_data.ins.pas @ case_1];
      khr_fault.ins.pas @ case_1;
      khr_db.ins.pas;
      [khronos_utl.ins.pas @ case_1];
   end;

{ This is the Aggregate that builds the database.
}
   aggregate khr_db =
     default for ?* =
         use_pool
            hm_pool;
     end of ?*;
     default for ?*.pas =
          @ case_1;
         depends_source
            khr_db.ins.pas;
            khr_db_utl.ins.pas;
            lsc.ins.pas;


         translate
            /com/pas %source -comchk -b %result @
               -idir //orange/case %option(-dbs) @
               %option(-nb)
            //dean/case/copy_to_bin.sh %result.bin %source
            %done;

         depends_result
            library_database.ddl;
      end of ?*.pas;
```

```
        use_pool
            hm_pool;

        translate
            /com/bind -b %result -nomes - <<!
            %result_of(?*.pas).bin
            !
            %done;

        depends_result

        element library_database.ddl @ case_1 =
              depends_tools
                  ['/com/sch'];

{ This translation rule looks for the directories /d3m and
  /d3m/schemas and creates them if they don't already exist.
  It then creates a schema file.  Finally, it moves several
  files into the binary pool.
}
             translate
                 if existf /d3m then else crd /d3m endif
                 if existf /d3m/schemas then @
                    else crd /d3m/schemas endif
                /com/sch %source({?*}.ddl) -s %result.sch @
                    -ss pas
                /com/mvf /d3m/schemas/library_database_$p.sub @
                    %result.sub
                 /com/mvf library_database_$p.uwa.pas @
                    %result.uwa.pas
                 %done;
end;

        element khr_db_$$error_handling.pas =
           depends_source
              [khronos_global_data.ins.pas];
              [khronos_utl.ins.pas];
           end;


        element khr_db_$$open_close.pas =
           depends_source
              [khronos_utl.ins.pas];
        end;

        element khr_db_$elements.pas =
           depends_source
              [khronos_utl.ins.pas];
        end;


        element khr_db_$events.pas =
           depends_source
              [khronos_utl.ins.pas];
        end;
```

```
        element khr_db_$monitors.pas =
           depends_source
              [khronos_utl.ins.pas];
        end;

     end; {  of khr_db; }

{ end dsee_common.ins.sml }
```

Index

A

B

C

D

E

F

H

I

L

M

N

O

P

R

S

T

V

W

Reader's Response

Please take a few minutes to send us the information we need to revise and improve our manuals from your point of view.

Document Title: *Engineering in the DSEE Environment*
Order No.: 008790-A00
Date of Publication: July 1988

What type of user are you?

___ System programmer; language _____
___ Applications programmer; language _____
___ System maintenance person
___ System Administrator
___ Manager/Professional
___ Technical Professional
___ Student
___ Novice
___ Other

How often do you use the Domain system?_____

What additional information would you like the manual to include?_____

_____

Please list any errors, omissions, or problem areas in the manual by page, section, figure, etc._____
_____
_____

_____
Your Name

Date

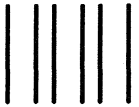
_____
Organization

_____
Street Address

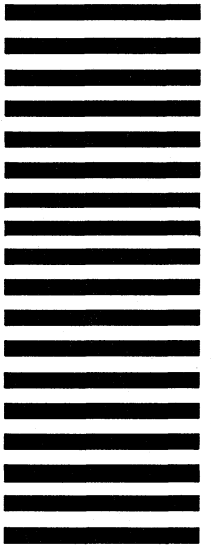
_____
City State

Zip

No postage necessary if mailed in the U.S.



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 78 CHELMSFORD, MA 01824

POSTAGE WILL BE PAID BY ADDRESSEE

APOLLO COMPUTER INC.
Technical Publications
P.O. Box 451
Chelmsford, MA 01824