Computer-Aided Software Engineering in a Distributed Workstation Environment

David B. Leblang
Robert P. Chase, Jr.

Apollo Computer
15 Elizabeth Drive
Chelmsford, Ma. 01824

ABSTRACT

Computer-Aided Software Engineering environments are becoming essential for complex software projects, just as CAD systems have become essential for complex hardware projects. DSEE, the DOMAIN Software Engineering Environment, is a distributed, production quality, software development environment that runs on Apollo workstations. DSEE provides source code control, configuration management, release control, advice management, task management, and user-defined dependency tracking with automatic notification.

DSEE incorporates some of the best ideas from existing systems. This paper describes DSEE, contrasts it other systems, and discusses some of the technical issues involved in the construction of a highly-reliable, safe, efficient, and distributed development environment.

Background

The phrase "programming environment", while used in many contexts, generally refers to an operating system environment and a collection of tools or subroutines. Because each programming environment has different goals, this makes direct comparisons of functionality difficult. The intent of this paper is to describe DSEE and contrast it with other systems in terms of goals and functionality. Though there are many programming environments [San78, Hab82, Ost83, SDE81], this paper will focus comparisons on a few well known systems:

- **UNIX/PWB**[Ivi77] includes the SCCS[BTL81] source code control system and the MAKE[Fel79] configuration tool. RCS[Tic82] is a more powerful source control system that also runs on UNIX systems. CMS and MMS [DEC82] are the VAX/VMS equivalent to SCCS and MAKE. CMS provides a richer set of source control capabilities than does SCCS or RCS; MMS is virtually the same as MAKE. SCCS/MAKE, RCS, and CMS/MMS work with the standard compilers, editors, and debuggers found on the host system.
- **ALS**[Tha83], the Ada Language System, was developed by Softech to meet Stoneman[Sto80] requirements for an Ada programming support environment. ALS includes an Ada compiler, debugger, binder, and execution environment. In addition, the ALS has a source code control system that keeps successive generations and variants of packages. The ALS does not have a single configuration management tool, but it provides the primitives needed to build one. The ALS Ada compiler/linker detects the need to recompile (as required by the Ada standard).
- **Cedar**[Lam83,Tei83] is built on the Xerox PARC Computer Science Laboratory system. Although the Cedar system does not provide for source code control, it does allow several copies of a module to exist, each stamped with a date/time. The Cedar "System Modeller" is a configuration management tool that notices when a new version of a module comes into existence (via coordination with the editor), and can build a complete program from an arbitrary set of module versions. Cedar requires that only the Cedar editor and compiler be used.

Introduction to DSEE

DSEE is implemented as one program, with instances running at various nodes in the network. DSEE is designed to manage large-scale development efforts involving engineers, technical writers, managers, and field support. Since these organizations, and their data, are typically spread among many locations, DSEE must recognize and support, distributed development environments. The underlying Apollo DOMAIN architecture helps by providing network-wide virtual address space, transparent remote file access, and remote paging [Lea83]. DSEE uses a distributed database management system (D3M) to store historical information; reliable, immutable, files to store deltas and tasks; server processes that watch for asynchronous events; and a store and forward inter-process communication mechanism (used in case the network becomes temporarily partitioned). The DOMAIN system supports multiple windows, each of which may run a seperate process. Some windows provide general system commands through a standard shell; others run dedicated applications like mail and calendar. DSEE runs as a dedicated window that provides commands for activities directly related to software development.

A DSEE product goal requires that it work with any language or text processor; in addition, users may pick any editor. In order to work its "magic", without changing existing tools, parts of DSEE had to be implemented directly in the operating system. Thus, without changes to any existing tools, the compilers, editors, print spoolers, etc. are all able to understand DSEE file formats and obey DSEE Configuration Manager version constraints. This powerful capability distinguishes DSEE from all of the systems described above.

DSEE consists of several "managers":

- o The history manager controls source code and provides complete version histories.
- o The configuration manager builds systems from their components and detects the need to rebuild.
- o The task manager relates source code changes made thoughout the network to particular high-level activities.
- o The monitor manager watches user-defined dependencies and alerts users when such dependencies are triggered.
- o The advice manager holds general project related information and provides templates for re-doing common tasks.

History Management

The History Manager (HM) provides source code control within the DSEE environment. The HM is a reserve/replace and incremental change (delta) oriented system. Related source elements are grouped into DSEE libraries. Source elements are stored in a special, highly-compressed, format (see HM technical notes below). Users reserve an element for modification and edit a local copy. When they have finished changing and testing the element copy, users replace the element, creating a new version. The HM inquires about the reasons behind a change, then records that information, along with the date/time, node id, and person's name in the history database associated with the library. At a later time, the history of an element can be reviewed, specific changes viewed line by line, and any past version ot the element retrieved. If a user attempts to "reserve" an element that is already reserved, DSEE warns the user of the conflict by stating why the element was reserved and by whom. Parallel development is allowed, but only on distinct branches (see "variant branches" below).

In a distributed workstation environment it is usually other instances of DSEE managers that need to be informed when a new version is created. For this reason, the DSEE process creating the new version sends inter-process communication messages to other nodes. The sections on the Task and Monitor Managers in this paper discuss this feature in more detail.

Another aspect of a distributed environment is that "partial" failures can occur. DSEE provides reliable recovery for partial failures. The distributed DBMS used by the HM uses journal files and semaphores to implement transactions. The HM updates delta files within the transactions by creating unnamed temporary files, force writing them to disk, and then making them named permanant files. The mechanisms described are somewhat expensive in terms of compute cycles, but since the time between reserves and replaces is measured in tens of minutes to hours, the overhead of several cpu seconds is worth it, considering the added reliability.

An element normally evolves along a linear line of descent. However DSEE supports three types of variant evolution. The first type of variant evolution answers the problem of what to do when a bug is discovered in a previous release of an element. The maintainer wants to modify the old version of the element without affecting, or being affected by, current development activities. The DSEE HM provides a mechanism for creating a new, independent, line of descent for the element that branches off an existing version (as does RCS, and SCCS). On a large project, some members may work exclusively on branches, while others continue to develop the main line of descent. Just prior to a new release, the branch may be merged into the main line, thus incorporating bug fixes into the main development work. DSEE provides a multi-window interactive merge command that automates much of the merging process. The user can override an automatic merge decision or make edits to the resulting file as the merge proceeds.

DSEE, like ALS, is a HOST/TARGET oriented system; that is, it assumes that the code being developed is intended to execute on a variety of target machines, not just the machine which hosts the software development environment. The second type of "variant" allows for alternate, radically different implementations of the same module -- a requirement of some projects. (For example, an I/O control module may have one implementation in 68000 assembler, and another in PDP-11 assembler). The ALS provides "variation sets" to deal with exactly this problem. DSEE relies on the Configuration Manager and Monitor Manager (discussed below) to solve this problem. DSEE users can create two distinct elements and let the Configuration Manager pick the right element based on the current configuration description. The Monitor Manager ensures that when one implementation is changed, the user is notified that the other implementation requires changes too.

The third type of variant arises when the alternate implementations are subtly different, and may be implemented in a single element with embedded conditional compilation statements. Again, the Configuration Manager is relied upon to pass the appropriate flags to the compiler based on the current configuration description. With this approach, most of the element is shared and so changes made in the common sections affect all target variants.

Technical Notes on History Management

The ALS and Cedar store full copies of old versions of elements. Because DSEE is designed to support large systems over a long period of time, and on moderately sized disks, it stores only the incremental difference (delta) between successive generations. RCS and SCCS are also delta based.

The use of deltas saves an enormous amount of space. Statistics on typical Pascal modules managed by the HM showed that each new version makes the delta file about 1%-2% larger. In other words, **50-100 versions of a module can be stored in the same amount of space as 2 copies of that module.** These space savings answer those who say that source code control systems use too much disk space and that users should just keep each module and its backup (i.e., module.BAK).

In addition to deltas, DSEE saves space by compressing leading blanks in source files to a space count byte. Again the savings are enormous. Statistics on Pascal modules held by the HM showed that 20% of each module consists of leading blanks. The combination of deltas and space compression leads to an interesting phenomenon: an HM element, with 5-10 versions, is often smaller than a single clear text copy of that element.

DSEE/HM and SCCS use "interleaved" deltas (ie. there is only one file containing all of the versions of the element). Intermixed control records allow the source code control system to extract any version of the element in a single pass over the file. RCS uses "seperate" deltas; i.e., a whole, plain text, copy of the most recent version is kept along with deltas describing how to go "backwards" from the current version to old versions. RCS can provide the most recent version very quickly, but has more trouble implementing variant branches. This is described in Tichy's paper [Tic82], which also gives an excellent discussion on the various styles of deltas. DSEE uses a variant of the delta algorithm described in [Hec78]. This choice was made for functionality reasons, not for performance. The ability to construct any version in a single pass over the interleaved delta file is critical to the implementation of DSEE extended streams. Extended streams offer ordinary, unmodified programs transparent access to any version of a DSEE element.

DSEE's element history files, like all file system objects in the Apollo 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. For each object type there exists a corresponding stream manager implementing standard stream operations on objects of that type (eg. open, close, get_record, put_record, seek, etc).

When the Apollo DOMAIN I/O subsystem is asked to open a "stream" on a system object, it allocates and initializes a file-descriptor and then dispatches, based on the object's type UID, to the appropriate stream manager to complete the "open". The dsee_history_manager_file stream manager determines and records the desired version number in the file-descriptor. The default is the most recent version in the main line of descent. However, the per-window global version map can indicate that some alternate version is desired. As subsequent calls are made to obtain the next record from the file, the DSEE stream manager is invoked to implement DSEE-specific behavior, which includes applying deltas and determining the next record in the desired version. The version maps, set-up by the DSEE Configuration Manager, are described below.

Configuration Management Background

MAKE looks at each item in the makefile and find its date-time modified (DTM). If the DTM of an object pre-dates the DTM of any of the objects it depends on, the object is rebuilt. This DTM based approach is fine when you are trying to build a system from all most recent sources, but it fails to deal with more complicated cases involving old versions, variant branches, or multiple targets. Moreoever, MAKE is very "binary" oriented; the user must describe the system in terms of the object modules that go into it, rather than in terms of the source modules. MAKE supports a **dynamic** style of development, in which each user sees other users' changes as soon as the become available.

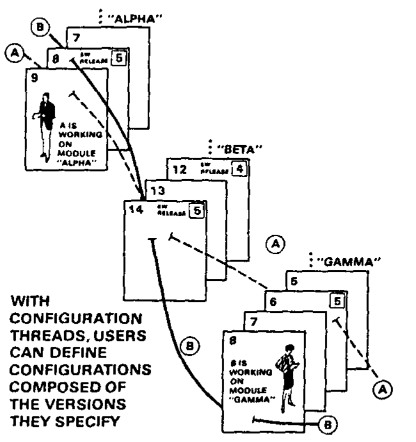
The Cedar "system model" [Lam83a] allows users to name specific versions of files (basically by giving the desired creation date). This allows Cedar to rebuild old systems, and to let individual users build their own versions. Cedar is source oriented; that is, the model is given in terms of source modules and the Cedar builder goes off and searches for the binary (if any) that corresponds to the requested version of the source. If no binary is found, Cedar will re-build it from the source. Cedar supports a **cautious** style of development in which each user is isolated from other users' changes until an explicit request to incorporate someone else's changes is made.

DSEE Configuration Management

The DSEE Configuration Manager support both cautious and dynamic styles of development. The DSEE CM separates the concept of the "system model", which it treats as the blueprint for the construction of a system, from that of "version specification".

A DSEE system model is a description of the components that comprise an application, the "build" dependencies inherent in each component, and the build rules that must be applied to a component in order to rederive an object module from source. The CM considers a component's "build dependencies" to be the set of objects that are relevant to the rederivation of the component (e.g., "include files"). The system model does not state which **versions** to use in a build; it simply defines the static properties of the application. The system model may reference elements in several DSEE HM libraries as well as non-DSEE objects. The DSEE system model is source oriented; binaries are not mentioned in the model. Instead, binaries are referred to as the result of translating the corresponding sources (eg. "%result (foo.pas)"). The system model language supports multi-step derivations and translators that have multiple outputs.

CONFIGURATION THREADS


"ALPHA"
B
A
7
8
SW RELEASE
5
9
A IS WORKING ON MODULE "ALPHA"
"BETA"
12
SW RELEASE
4
13
14
SW RELEASE
5
A
"GAMMA"
5
6
5
7
8
B IS WORKING ON MODULE "GAMMA"
B
A
B
WITH CONFIGURATION THREADS, USERS CAN DEFINE CONFIGURATIONS COMPOSED OF THE VERSIONS THEY SPECIFY

A DSEE configuration thread (CT) states which version of each component named in the system model should be used for a build. The CT is very flexible. That is, it may state version information very explicitly (ie. "for foo.pas use version 22"), or in more dynamic terms (ie. "for foo.pas use MOST RECENT version"). Wildcard patterns are allowed, as are variant branch specifications. At build time, the CT is evaluated and used to "bind" the components in the system model to particular versions. The fully bound object is similar to a CT, in that the versions are given explicitly and the default translator rules are filled in. This bound configuration thread (BCT) is used to perform the build. The BCT used in a build is a valuable record of that build, since it lists the actual versions of all constituent components and include files for all of the components in the system model. The BCT also lists the translator command lines used. and additional, user-defined, information. A BCT may refer to another BCT. This happens when a component is built from the results of translating other components; e.g., the BCT for a program refers to the BCT of each of the modules that comprise it.

Common Configuration Threads

"Dynamic Developer"
What I have RESERVED
+ Most Recent on main lines of descent

"Cautious Developer"
What I have RESERVED
+ What I REPLACED
+ Last Base Level

"Dynamic Maintainer"
What I have RESERVED
+ Most Recent on BUGFIX lines of descent
+ Last Release

"Cautious Maintainer"
What I have RESERVED
+ What I REPLACED
+ Last Release

Building Programs with the DSEE CM

The CM maintains a derived object pool which holds several version of each object that was produced as the result of building a component named in the system model (e.g., binaries). Each derived object in the pool is associated with the BCT used to build it. When asked to build, the CM determines a "desired" BCT by binding the system model to the versions requested by the user's current CT. The CM then looks in the derived object pool to see if there is a BCT that exactly matches the one desired. If a match is found, the derived object associated with that BCT is used. Otherwise, the component is rebuilt in accordance with the desired BCT, and the new derived object and BCT are written to the pool. In all cases, the user is given exactly what he asked for.

Releasing Software

In addition to a number of binaries and BCTs, each full system build results in an entry in the build log. This log keeps track of exactly what was built, when and why, and by whom. Given a build log entry, DSEE can find the BCT in the pool that corresponds to that build. A release consists of the system that was built, its BCT, and keywords (such as "rev5") that describe the system. These objects are stored in a safe, stable database. DSEE can perform various checks by analyzing the BCT; for example, it can warn when more than 1 verson of the same element is used. Later, when a bug is reported in a released version of the system, the maintainers can use keywords to locate the version in the database and find the BCT - which will describe the exact versions used in the system. Since the History Manager has all of the old sources, users can base their CT on the BCT of the release, thereby re-establishing the environment that existed when the release was made. By making minor edits to an explicit CT, users can fix bugs without disturbing most modules.

DSEE can create a shell in which all programs executed in that shell window transparently read the exact version of an element requested in the user's configuration thread. The History Manager, Configuration Manager, and extensible streams mechanism (described above) work together in this way to provide a "time machine" that can place a user back in a environment that corresponds to a previous release. In this environment, users can print the version of a file used for a prior release, and can display a readonly copy of it. In addition, the compilers can use the "include" files as they were, and the source line debugger can use old binaries and old sources during debug sessions. All of this is done without making copies of any of the elements.

Technical Notes on Configuration Management

Because maintenance and development proceed in parallel on a large project, the derived object pool will contain binaries that correspond to a previous release as well as binaries used for current development. Over time, a large number of binaries can accumulate. However, the derived object pool may contain binaries that no one requests anymore because they are too old, or binaries that have been superseded by new development. Since the DSEE history manager holds the sources needed to reconstruct any binary, a binary may be safely deleted from the pool. A pool "garbage collection" algorithm runs occasionally to discard binaries that haven't been used in a long time.

The version map, mentioned earlier, provides the DSEE stream manager with per-element version information. When a component must be rebuilt, the DSEE CM sets the version map to reflect the version of each subcomponent listed in the desired BCT, then the CM executes the build command declared for the component in the system model. When subcomponents are opened by the DSEE stream manager, it consults the version map to determine which version of the subcomponent should be read.

Users need the configuration manager to ensure the consistency of their systems, and they expect to pay for the added safety by sacrificing some performance. However, they do expect the CM's performance to be acceptable. The DSEE CM maintains several caches to improve rebuild performance. For example, a "latest version" cache is used to speed the resolution of CT references to "most recent" versions. Other performance improvements are gained by using hash values to cut down on the number of pool objects that must be examined when the CM is trying to match a desired BCT against the BCTs in the pool.

Task Management

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 which 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 which were performed as part of one higher-level task.

Most of the steps taken to accomplish a task modify elements, but not only program module elements. For instance, adding an enhancement to a system may also require updating the system's design specification, user manual, and on-line help files. 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 programmming.**

To effectively manage a large task, a software development environment needs a convenient way to record ALL related sub-tasks perfomed by any number of persons on any of the nodes in the workstation network.

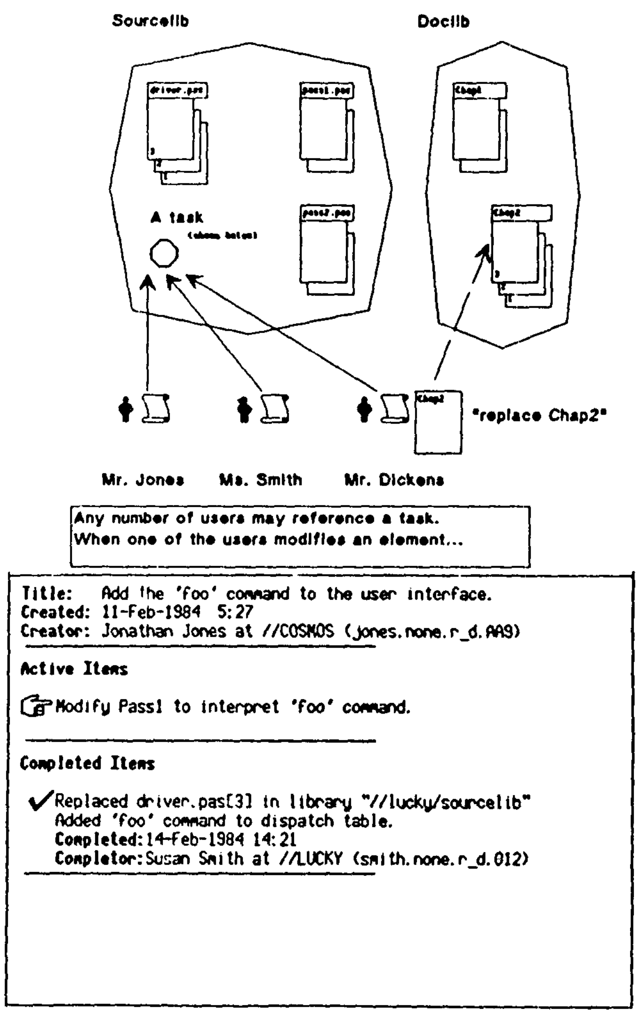
DSEE provides tasks and tasklists for this purpose. A DSEE task is a structure used to plan and record the low-level steps involved in a high-level activity. A task consists of a title, which describes the high-level activity, and a list of textual items, which are the sub-tasks that must be performed. Tasks are displayed graphically as having active items awaiting action, and completed items. The list of completed items is referred to as the transcript. A user has a current task as part of his per-window context. The current task is orthogonal to the current library, so the user can switch from a code library to a design or documentation library as part of the same current task.

The History Manager interacts directly with the Task Manager by "tagging" each newly created version of an element with the current task. Later, a user looking at a single element modification can easily find and examine all the related modifications which were part of a given task. When the HM records a user's modification comments for a new version, it sends a copy of the comment (along with the library name, element name, username and time) to the transcript of the current task. This provides for more useful "audit trails", organized by task, as compared to the History Manager's typical audit trails, which list all modifications to one library. Tasks reside in libraries, and normally are NOT deleted upon completion, since the task transcript contains information that may prove useful long after the task is completed.

Besides the element modification information, other items in the task may describe ANY ACTIVITY which was part of the task (e.g. "Call customer x", or "run the new system through Q/A"). A graphic task editor is provided, which allows a user to add a new active item (or delete an item which is determined to be unnecessary), and to "check off" an active item that has been completed. (This moves the item from the task's "active list" to the transcript.)

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, and each library contains two tasklists -- one for active tasks, and one used mainly for completed tasks. For flexibility in project organization, an arbitrary number of additional tasklists may be created. A task may be referenced by several tasklists if several people need to take part in its completion. In this case, each user sees items completed by other users immediately, since tasklists contain references to tasks, rather than tasks. Users may add task references to other users' tasklists, subject to access control considerations.

knowing what steps were part of the task. A complete task can serve as a guide for future, similar, tasks; for instance, a completed task entitled "Add the 'foo' command to the user interface" could provide a useful outline for a new programmer who was instructed to "add the 'bar' command to the user interface". This last concept is called advice, and is discussed in more detail below.



Any number of users may reference a task.
When one of the users modifies an element...

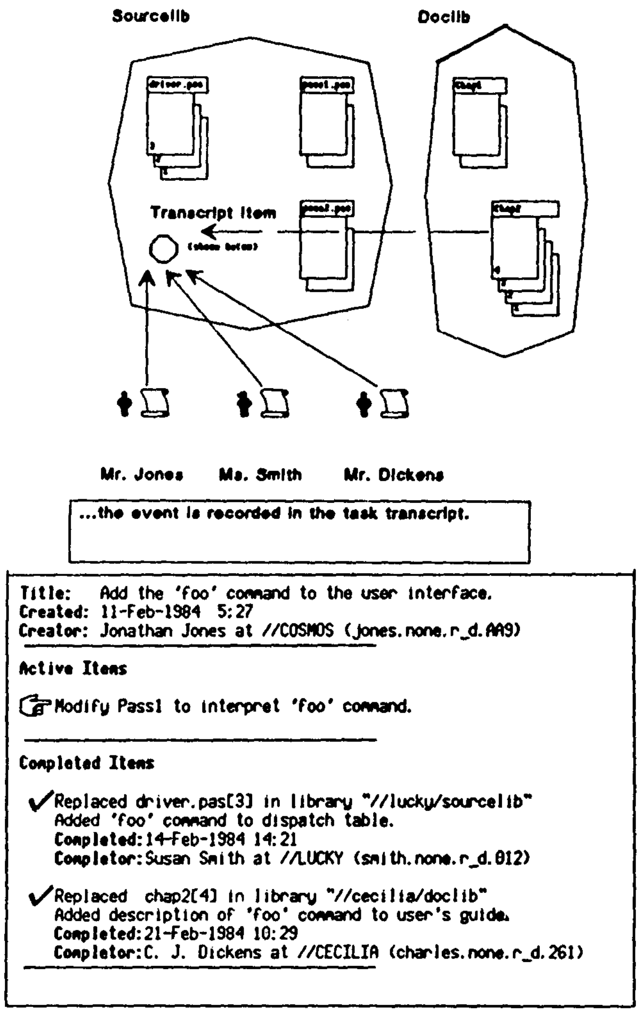
```
Title:   Add the 'foo' command to the user interface.
Created: 11-Feb-1984  5:27
Creator: Jonathan Jones at //COSMOS (jones.none.r_d.AA9)

Active Items

Modify Pass1 to interpret 'foo' command.

Completed Items

 ✓Replaced driver.pas[3] in library "//lucky/sourcelib"
  Added 'foo' command to dispatch table.
  Completed:14-Feb-1984 14:21
  Completor:Susan Smith at //LUCKY (smith.none.r_d.012)
```



...the event is recorded in the task transcript.

```
Title:   Add the 'foo' command to the user interface.
Created: 11-Feb-1984  5:27
Creator: Jonathan Jones at //COSMOS (jones.none.r_d.AA9)

Active Items

Modify Pass1 to interpret 'foo' command.

Completed Items

 ✓Replaced driver.pas[3] in library "//lucky/sourcelib"
  Added 'foo' command to dispatch table.
  Completed:14-Feb-1984 14:21
  Completor:Susan Smith at //LUCKY (smith.none.r_d.012)

 ✓Replaced  chap2[4] in library "//cecilia/doclib"
  Added description of 'foo' command to user's guide.
  Completed:21-Feb-1984 10:29
  Completor:C. J. Dickens at //CECILIA (charles.none.r_d.261)
```

When a user has finished with his part of the task, he removes the task reference from his tasklist. When no tasklists reference the task and all of the active items have been completed and "checked off", the task is completed. The task is probably not deleted, however, since it contains a record of ALL of the sub-tasks that were performed for the task, which is potentially useful information. For example, the project leader may wish to examine the completed task to verify that it was correctly performed. Bug fixes may require

Technical Notes on Task Management

Because DSEE is used in a distributed environment, special attention was given to the implementation of task and tasklist operations which may involve more than one node in the workstation network. For example, if a user creates a new version of an element, the event is recorded in the task transcript of his current task. However, the library where the new version is created 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 will the task transcript be updated? DSEE would not be a very supportive environment if it disallowed the creation of the new version when the network was partitioned. Therefore, a reliable (store and forward) message passing utility is used to guarantee that the update will occur. There will be 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 which 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 for a task to be referenced by users on more than one Apollo local network.

Advice Management

Often, a user needs to perform a task which is quite similar to a task that someone has performed before. The example of adding a new command to a user interface was mentioned above. Many of the same modules in the system will need to be changed, the same chapters in the design document and user manual will need to be updated, and the same customer might need to be called on the telephone. Examination of the earlier task transcript can be helpful in determining the steps necessary to complete the new task.

The DSEE Advice Manager helps to manage common tasks. The main component of a "piece of advice" is the task template. A task template is similar to a task, but contains no completed items, only model active items. Hence, task templates are models from which tasks are built. New tasks may be instantiated from task templates. When a new task is instantiated, it inherits all of the model active items from the task template. As work proceeds, items in the tasks are completed and moved to the transcript, as usual.

The simplest type of advice is the form, which is a named task template. Forms may be created from scratch, or may be copied from tasks. When they are copied from tasks, all transcript items in the task become active items in the form. A command is provided to instantiate a new task from a form. To see this, consider the example used above. Let's assume that when the task entitled "Add 'foo' command to the user interface" was completed, the project leader created a form from the task, and stored it in the file "add_new_command" in a project advice directory. The form may be edited to generalize it from how to add the 'foo' command to how to add any new command. Later, when the 'bar' command is to be added to the user interface, a new task entitled "Add 'bar' command" could be instantiated from the "add_new_command" form. The new task could then be edited to make it specific to the particular task at hand.

Forms are a simple type of advice that make it easy to manage tasks that are similar to other tasks. A more intricate type of advice is presented in the next section.

Monitoring Dependency Relationships

Many software development environments have some mechanism for tracking "build" dependencies; i.e., there is some way to detect when a module needs to be rebuilt because one or more of its constituents has been modified. MAKE[Fel79] MMS[DEC82] and Cedar[Lam83] all provide automatic building functionality, as does the DSEE Configuration Manager (discussed earlier in this paper). There is another type of dependency tracking which is not addressed by these other systems, and which is more people-oriented than build-oriented. Users should be able to define dependencies on elements such that other users will be informed of those dependecies before modifying the elements, and such that the user defining the dependency will be informed when the elements ARE modified.

The types of dependencies that require automatic notification involve semantic dependencies, which cannot be detected by builders or other software tools, but can only be detected by people. Communication between developers is necessary to properly track such dependencies. Unfortunately, most software development organizations have imperfect communication paths, a problem which is particularly acute in large scale software development efforts, where many persons work on the different phases of product development (e.g. design, implementation, quality assurance, documentation, release coordination, etc.). The problem is best demonstrated by a few examples:

- Technical documents' dependence on implementation:
  When the programs that implement the user interface for a system are modified, the system's help files and user documentation may need to be changed. Therefore, the technical writers for the product need to be notified.
- Inter-module semantic dependencies:
  Programmers sometimes design code that depends on functionality in a module that is not reflected in the procedural interface. For instance, some modules might depend on the fact that a certain command line parser always converts the command line to uppercase. Before modifying the parser, it would be helpful to know that certain modifications would cause problems for other modules that depend on the parser. Incompatible changes could be avoided by consulting with the user who declared the dependency before making any changes. With or without consultation, the implementor of the dependent module needs to be notified when a semantic dependency may have been violated. Note that an automatic builder wouldn't give advanced warning, or even detect the problem described. Furthermore, the problem isn't simply one of

rebuilding; the modules which used the parser routine would have to be modified to no longer exploit their dependency on uppercase command lines.

- The "Common Module" Problem:
Suppose the programs for two products share a common module, but the products are maintained by seperate implementation groups (e.g. common back-end for a family of compilers). There is a "build" dependency involved here, but note the difference between this problem and the problem addressed by automatic builders. When one group changes the common module, then the automatic builder correctly rebuilds that group's system. The other group's system will be correctly rebuilt the next time they invoke the automatic builder, but they may be working on another project, and not even know that their system needs to be rebuilt. They need to be notified, so they can invoke the automatic builder to rebuild their system. For complete automation of this type of dependency tracking, it might even be desirable for the automatic builder to be automatically invoked for both systems whenever the common module was changed.

Certainly, other examples can be found. A flexible solution is needed to address the general problem of dependency tracking with automatic notification.

DSEE provides for setting monitors on elements, and the DSEE Monitor Manager tracks the dependencies defined by the monitors that users create. A monitor may contain a piece of advice (i.e. a task template) and a list of tasklists to receive that advice. It may also contain a list of activation commands, executable by the shell. When a user creates a monitor, he enters comments describing the dependency involved. A monitor is set on one or more elements in a DSEE library (referred to as the target elements of the monitor). When some user RESERVEs a monitored element, he is informed by DSEE that the element is monitored and is shown the description of the dependency and the name of the user who declared the monitor. This way, the element won't be modified without first considering the dependency.

A monitor is activated when a new version is created for any of the target elements. When a monitor is activated, a new task is instantiated from the task template and a reference to the new task is added to each tasklist named by the monitor. The users who had the dependency are therefore advised that they should check the new version to see if their dependency is still met. The new task instantiated from the monitor's task template advises them about what to do.

If the monitor contained executable commands, they are automatically executed when the monitor is activated. For the "common module" example above, activation commands could be used to automatically invoke the automatic builder to rebuild the systems which use the common module.

The automatic notification basically adds a new task reference to a user's tasklist. Of course, the user must look at his tasklist in order to notice that the monitor was activated. For users who examine their tasklists infrequently, DSEE provides a tasklist alarm server, which "watches" tasklists specified by the user. When some user activates a monitor by modifying an element and other users are notified by a task new reference on their tasklists, the alarm server notices and pops up small windows on their screens, informing them that they have a new task. (The title is also displayed).

Technical Notes on Monitor Management

We have temporarily restricted the use of executable activation commands in monitors for security reasons. The securty problem is this: if a user creates a monitor with executable commands, and another user activates the monitor, the commands are executed in the activator's protection domain. This creates a "trojan horse" security loophole (see [Ame83] for a discussion of the "trojan horse" problem). Eventually, a server process acting on behalf of the monitor's creator will execute the commands, and the restriction will be removed.

As with certain task operations in DSEE, monitor activation may involve accessing objects on different nodes in the network, and network partitioning may cause some of the nodes to be temporarily inaccessible. The store and forward message passing utility (as described in the section on tasks) handles this complication gracefully, and allows monitor activation to work across inter-network gateways.

Security and Protection

DSEE is designed to be usable by projects in large software development environments, where many autonomous, but not necessarily trustworthy, users share the same network. Some mechanism for controlling access to files is needed in such an environment. The DOMAIN environment provides a normal access control list (ACL) mechanism to protect users' files, and DSEE provides a protection mechanism based on ACLs, to ensure that DSEE objects can be protected to the same degree as other files in DOMAIN. Special care is taken by DSEE not to introduce any security loopholes into the environment (see the Technical Notes on Monitor Management for a description of "trojan horse monitors").

Support for HOST/TARGET Development

A practical software development environment should support the development of systems which will run on multiple target machines. In addition to providing support for branches and variations in elements and extensive support for different configurations of a system, DSEE is general enough to exploit other software in the DOMAIN environment. For instance, since DSEE does not require the use of a particular programming language or compiler, users may choose from a variety of 3rd party cross-compilers to develop systems to run on different target machines. Network gateways are available for communication with other vendors' networks, and the store and forward message passing utility can be used in conjunction with gateways to support inter-network tasks and monitors. These capabilities simplify the process of transporting a system from the host machine to the target.

Current Status and Futures

At the time of this writing, DSEE is in preliminary field test. It is also being used extensively by very diverse in-house groups developing microcode, graphics software, languages, documentation, and end-user applications. More than a million lines of code, in several thousand modules, are currently being managed. Initial results have been excellent.

In addition to enhancing the current system, future plans for DSEE will involve language directed tools such as structure editors [Leb82, Tei81], interpretive debuggers [HLD83], and graphical program representations. The Advice Manager will be extended to provided some sort of indexed keyword advice and more finely honed monitors. We will try to incorporate more third party tools for cross-machine developement, formal software design (like PSL/PSA), and program verification.

Acknowledgements

In addition to the authors, DSEE project members Gordon McLean, Howard Spilke, John Yates, and Pat Bergeron contributed to this paper.

References

[Ame83] S. Ames, M. Gasser, R. Schell
"Security Kernel Design and Implementation"
IEEE/Computer, Jul 1983.

[BTL81]
Source Code Control System User's Guide
UNIX System III Programmer's Manual, Oct 1981.

[DEC82]
CMS/MMS: Code/Module Management System Manual
Digital Equip. Corp., 1982.

[Fel79] S.I. Feldman
"Make - A Program for Maintaining Computer Programs"
Software Practice and Experience, Apr 1979.

[Hab82] N. Habermann, et al.
The Second Compendium of Gandalf Doc.
CMU Comp Sci Dept, May 1982

[Hec78] P. Heckel
"A Technique for Isolating Differences Between Files"
CACM, Apr 1978.

[HLD83] Several Papers
Software Eng. Sym. on High-Level Debugging
ACM/SIGSOFT/SIGPLAN, Aug 1983.

[Ivi77] E.L. Ivie
"The Programmer's Workbench"
CACM, Oct 1977.

[Lea83] P. Leach, P. Levine, B. Dorous, J. Hamilton, D. Nelson, B. Stumpf
"Architecture of an Integrated Local Network"
IEEE Jrnl on Selected Areas in Comm, Nov 1983.

[Leb82] D.B. Leblang
"Abstract Syntax Based Programming Environments",
ACM/AdaTEC Conf. on Ada, Wash. D.C., Oct 1982.

[Lam83] B. Lampson, E. Schmidt
"Organizing Software in a Dist. Environment"
SIGPLAN Jun 1983.

[Lam83a] B. Lampson, E. Schmidt
"Practical Use of a Polymorphic Applicative Language"
10th POPL Conf., Jan 1983.

[Ost83] L.J. Osterweil, W.R. Cowell
"The TOOLPACK/IST Programming Environment"
IEEE/SOFTFAIR, Jul 1983.

[San78] E. Sandewall
"Programming in an Interactive Environment: The "LISP" Experience"
Computing Surveys, Vol 10, No. 1, Mar 1978.

[SDE81] Collected papers
Tutorial: Software Development Environments
IEEE/COMPSAC-81, Nov 1981.

[Sto80]
"STONEMAN: Requirements for Ada Programming Support Environments"
U.S. Department of Defense, Feb 1980.

[Tha83] R. Thall
"Large-Scale Software Development with the Ada Language System"
Proc. of ACM Computer Science Conf., Feb 1983.

[Tic82] W.F. Tichy
"Design, Implementation, and Evaluation of a Revision Control System"
6th Int'l Conf on Software Eng., Sep 1982.

[Tei81] T. Teitelbaum
"The Why and Wherefore of the Cornell Program Synthesizer"
SIGPLAN. Jun 1981

[Tei81a] W. Teitelman, L. Masinter
"The Interlisp programming environment"
Computer, Apr 1981.

[Tei83] W. Teitelman
"Cedar: An interactive programming environment for a Compiler-Oriented Language"
LANL/LLNL Conference on Work Stations in Support of Large Scale Computing, Mar 1983.