Object Management in a CASE Environment

Evan W. Adams Masahiro Honda Terrence C. Miller

Sun Microsystems, Inc.
2550 Garcia Ave
Mt. View, Calif. 94043

Abstract

The Sun Network Software Environment (NSE) is a network-based object manager for software development. The NSE supports parallel development through an optimistic concurrency control mechanism, in which developers do not acquire locks before modifying objects. Instead, developers copy objects, modify the copies, and merge the modified objects with the originals. Objects managed by the NSE are typed, and the set of types can be extended by tool builders. The NSE is designed to work with heterogeneous implementations and poor communication.

1 Introduction

A large software product consists of a wide variety of *objects*. It consists not only of source, object, and executable code objects, but also of requirement, specification, design, schedule, test plan, test data, and documentation objects. Systems to manage these objects must address a number of problems. These problems include:

1) multiple people problems—large software projects involve multiple people working together on a common set of objects. Developers must worry about other developers concurrently updating objects they depend on.
2) multiple object problems—large software projects involve multiple objects, often numbering in the thousands. Developers must deal with sets of objects as units. They must be able to identify the versions of *all* the objects that make up a consistent unit, such as a release.
3) multiple release problems—large software projects involve development and maintenance of multiple releases at the same time, which requires the same object to have multiple versions undergoing change. Problems arise in trying to merge changes made in one release with changes made in another release.
4) multiple machine problems—workstations and networking exacerbate the above problems. Users must know where objects are located on the network. In addition, copies of objects can proliferate over the network, making it difficult to locate consistent versions of objects. Cooperative development on heterogeneous and geographically distributed networks must also be supported.
5) multiple tool problems—software development organizations have substantial investments in their existing tools. An object management system must be able to manage the objects that these tools manipulate without requiring that the tools be modified extensively.

A CASE environment which attempts to deal with these problems must include a distributed object manager capable of handling multiple versions of objects of different types.

2 The NSE Approach

The NSE's approach to solving the problems mentioned above is derived from three basic principles:

1) Parallel development should be encouraged and supported.
2) Management of objects manipulated by a wide variety of existing and future tools should be supported.
3) Cooperation between users who are physically as well as logically separated and employing heterogeneous implementations should be supported.

The following three sections describe the results of applying these principles.

©1989 ACM 0270-5257/89/0500/0154$00.75

Recommended by: Mark Dowson

3 Parallel Development

Parallel development is supported by the NSE through an optimistic concurrency control scheme we call the *copy–modify–merge* transaction paradigm. Simply stated, a developer copies a set of objects without locking them, modifies a subset of the copies, and then merges the modified copies with the originals. This paradigm allows developers to work in isolation from one another since changes are made to copies of objects. Because locks are not used, development is not serialized and can proceed in parallel. Developers, however, must merge changes to objects before the transaction can be committed. In particular, a developer must resolve conflicts when the same object has been modified by someone else.

The NSE design assumes that it is possible to provide tools that make the cost of resolving conflicts less than the cost of delays due to serialization. Our experience using the NSE to develop itself has provided good evidence to support this assumption.

Locks also do not adequately prevent other developers from updating files you depend on. You often only lock the files you intend to update, so you have no protection against others updating files that you depend on and assume will not change. You must at least read-lock all the files you depend on in order to get this level of protection, which serializes development even more.

In addition, we have seen that developers often subvert locks in systems that require them. Developers, in order to avoid waiting, make copies of files without getting locks, and only acquire locks when the changes have been made and the files are ready to be checked in. This paradigm is essentially copy–modify–merge. The NSE formalizes and legitimizes this paradigm by directly supporting development without locks.

3.1 Copy–modify–merge Problems

For a tool to successfully manage objects with the copy–modify–merge paradigm, a number of problems must be addressed. First, since software development involves multiple objects, it is not sufficient to just make copies of the objects that are to be modified. Instead, the entire set of objects that make up a program or system must be copied as a unit, so that after the changes are made, the changed objects can be tested to make sure they are consistent with all other objects in the set. Developers must therefore be able to locate and copy consistent versions of objects. Note that consistent versions are needed of not only the source code, but also of other objects such as documents, design specifications, and test scripts. These objects must all be kept consistent with each other.

Second, making physical copies of objects can be costly in both time and space. Methods for reducing the number of physical copies are needed.

Third, it is desirable to not have to change the names of objects when they are copied. That is, the name space for the copies should be identical to the original name space. Otherwise, references made from one object to another, such as absolute references to included files from source files, need to be changed, which is not only a nuisance but another source of error.

Fourth, after the changes have been made and tested, the objects that have been modified need to be identified and merged back with the originals. Due to the large number of objects that can be involved in a change, this step is very error-prone if done manually. Furthermore, conflicts must be detected so that another developer's changes made in parallel are not overwritten.

Finally, effective mechanisms are needed to rapidly resolve conflicts between two versions of an object.

3.2 NSE Copy–modify–merge Solutions

The NSE operates on sets of objects called *components*. Components are used to group objects together so that they can be managed as a single unit. Snapshots of components can be taken to create *components revisions*, so that consistent versions of objects can also be grouped together as a unit. Component revisions are immutable. Component revisions are implemented using sparse copies: only files modified since the previous revision are copied. The extensions to the Sun file system that enable sparse copies are described in [7]. Components are described further in the discussion of object types in section 4.1.3.

The NSE implements the copy step of the copy–modify–merge paradigm through an operation called `acquire`. `Acquire` takes a component revision as an argument and creates a copy of the versions of all objects in that component revision. Consistent versions of related objects are therefore copied together. The `acquire` operation normally does not create physical copies of file objects. Instead, the file is shared until the copy is modified. The extensions to the Sun file system that implement file sharing and copy-on-write semantics are also described in [7].

Components are acquired between NSE *environments*. An environment is a work space that contains (logical) copies of objects. Each environment can have a different copy of an object. An environment also provides a virtual name space for its objects, so the same name can refer to different versions of the object in different environments. For example, the file `/usr/src/diff/diff.c` in environment E1 can be a different version of diff.c than the file `/usr/src/diff/diff.c` in environment E2.

Environments have network-wide names, and can be accessed from any machine on a network. Developers need not know the physical network topology in order to access environments.

A developer acquires a component from a common *parent* environment to a private *child* environment. The parent environment serves as an integration area for a group of developers working on the same project. Each developer updates acquired objects in the developer's child environment in isolation.

Figure 1 shows an example environment hierarchy. The parent environment, called **languages**, contains a component called **compiler**. The **compiler** component contains all the files for a simple compiler, which are **scan.c** and **parse.c**, together with the **Makefile**. There are two developers, Jon and Mary. Jon works on the scanner so he acquires the **compiler** component from the **languages** environment into his own environment called **scanner_dev**. Mary works on the parser, so she acquires the **compiler** component from the **languages** environment to her own **parser_dev** environment. Note that both Jon and Mary acquire the complete set of files for the compiler, **scan.c** and **parse.c**, plus the **Makefile**, since the component **compiler** is acquired. Both Jon and Mary will need the three files to test their changes.

After making changes and testing them in the child environment, a developer attempts to commit the modified objects in the parent using the `reconcile` operation. The `reconcile` operation copies the changed objects in the component back to the parent environment. The NSE keeps track of what objects have been changed in an environment. This information is used to determine what objects to copy back and also to determine what objects are in conflict as a result of parallel changes.

A `reconcile` by a developer will fail if any object contained in the component has changed in the parent since it was acquired. This usually means that a second developer working in another child environment has concurrently

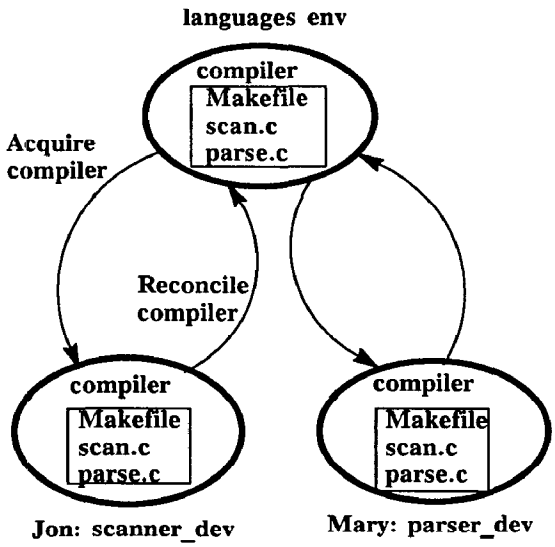

Figure 1

changed and reconciled objects that are also in the component that the first developer is trying to reconcile. The `reconcile` fails because the first developer's changes may no longer be consistent with the second developer's changes. For example, the second developer may have changed a common include file, which could cause the first developer's changes to no longer compile.

In such a case the `reconcile` operation does not copy back the changed objects. Instead, it calls the `resync` operation to acquire into the first developer's environment the new versions of objects in the parent. If an object has been changed only in the parent, the new version replaces the old one in the child. If the object has been changed in both parent and child, a conflict exists. `Resync` acquires the information needed to resolve conflicts; this is usually a copy of the new version from the parent and the version which is the common ancestor of the versions now in both the parent and child. The NSE `resolve` operation can then be called to resolve conflicts on each conflicting object. `Resolve` invokes the appropriate merging tool on each object depending on the object's type (object types are explained in the next section). Each object type integrated with the NSE provides a merging tool.

For ASCII files, the NSE provides a window-based merging tool, called `fileresolve`, as shown in Figure 2. `Fileresolve` uses information from the common ancestor of the two versions being merged. The merge technique is a simple three-way merge based on the Unix†

†Unix is a trademark of AT&T.

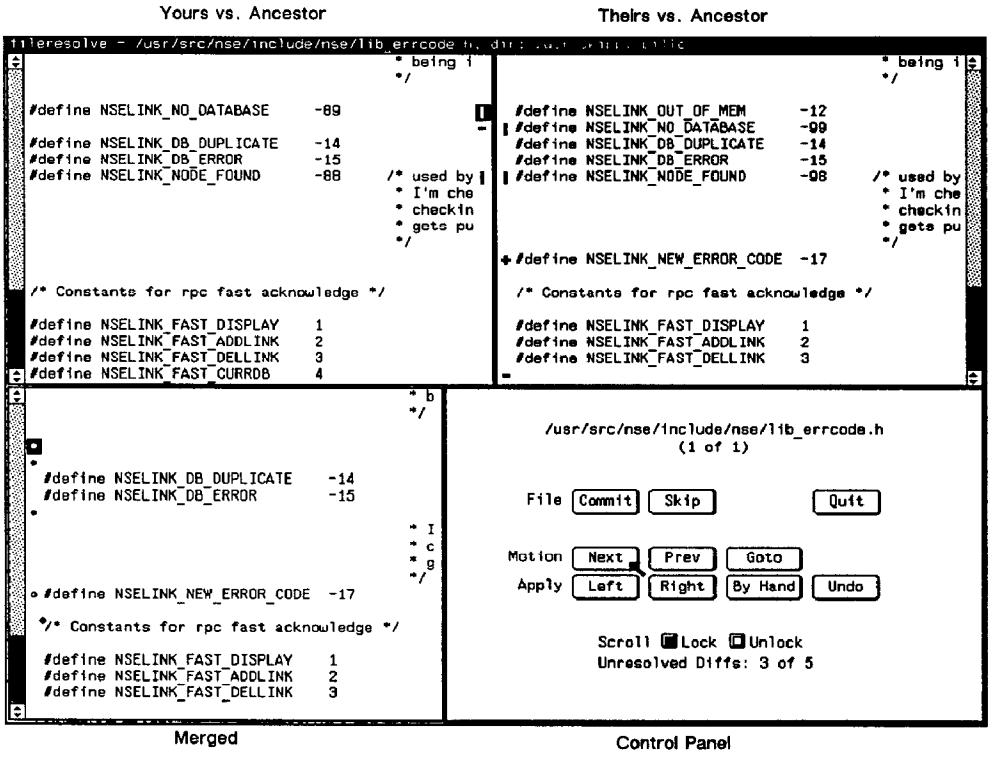

Figure 2

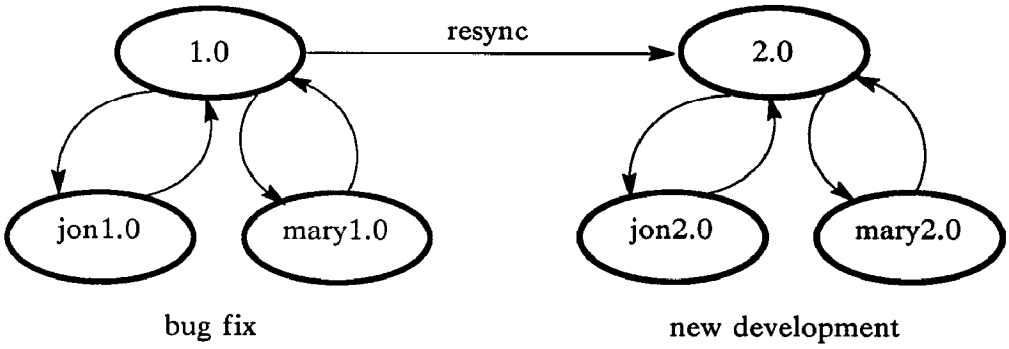

Figure 3

`diff` program. In our experience, this simple technique has proven to be very effective. When two versions of a source file are merged, for example, it is usually the case that each version is modified in different places; conflicts due to modifications to the same lines rarely occur. Furthermore, each version is usually modified for different purposes and so semantic conflicts are also rare. We have found that source files can be merged with little or no help from the user most of the time. Since the merge is not semantic-based, however, testing is required after all files have been merged.

The `resync` operation can also be used to update a new release with changes, such as bug fixes, made in an old release. When work on a new release is to start, a parent integration environment for the new release is initialized by acquiring objects from the old release's integration environment. Changes can then occur in parallel to both releases. As bugs are fixed in the old release, the changes can be `resync`-ed to the new release. Figure 3 illustrates this method.

4 Typed Objects

The NSE can manage objects produced by tools used in all phases of the software development life cycle. The key to the NSE's generality is the notion of *typed objects*. Each object has a type. A type manager, which controls the type-specific aspects of managing objects of each type, can be integrated with the NSE. Examples of object types are data flow diagrams managed by a CASE tool and documents produced by electronic publishing software—as well as source and object files.

An NSE object may be *compound* (containing a set of subobjects) or *primitive* and includes:

1) a standard set of attributes used to control NSE operations.
2) optional type-specific attributes.
3) for compound objects—either an enumeration of the set of subobjects or an algorithm for computing them.
4) uninterpreted data (the object's contents)

4.1 Standard Object Types

Three principal object types are built into the NSE; these are files, targets, and components. These are general-purpose objects applicable to any software project.

4.1.1 Files

In a software project, the most common objects are *source* and *derived* files. Derived files, such as object files, libraries, and executable programs, are built from source files by programs such as compilers and linkers. The NSE *file object* covers both source and derived files.

4.1.2 Targets

An NSE *target* is a compound object which dynamically computes its set of subobjects. It automates much of the bookkeeping associated with derived files generated using the `Unix make` utility [4]. The subobjects of a target are a derived file and the objects needed to build it. The objects needed to build a derived file consist of a *Makefile* and a collection of *dependencies*. A Makefile describes how to construct a derived file in the fewest possible steps. Dependencies are files such as source files, object files, libraries, and header files that, if changed, require the derived file to be rebuilt. The NSE automatically keeps the contents of targets up to date when Makefiles and dependencies are changed. The NSE also does not require that users explicitly list all dependencies in the Makefile. For example, it automatically includes files referenced in `#include` statements in C programs.

During such operations as `acquire` and `reconcile`, the NSE will compute the validity of derived files using the same time-based algorithm used by `make`. Invalid files are neither acquired nor reconciled.

4.1.3 Components

Whereas a target is a special compound object to hold a derived file and a set of consistent dependencies together, an NSE component is a general-purpose compound object that can be used to group any collection of related objects. Objects of any type can be members of a component, including other components, and one object can be a member of two or more components. Although the NSE does not restrict the contents of components, a typical component might contain a target object representing a program, and all other objects related to that program. These might include specifications, design diagrams, documentation, test data—in short anything that someone interested in the program might want to have available in one place to simplify examination or modification.

Because components can contain other components, they can represent the hierarchical structure of a complex software product. A typical system might consist of one top-level component representing the entire system; this component might have one component per subsystem, and these components might have subcomponents for programs. When components are used in this way to represent

levels of abstraction, each component can contain an intellectually manageable number of objects, say, five to ten. Some of these objects will be components representing the next lower level of detail. To see the next level of detail you can examine the subcomponents.

Components are the basic building blocks of NSE-managed software projects. Components are a project's work units, on which programmers can work in parallel.

4.2 Type Extensibility

The NSE allows tool writers or third party tool vendors to add object types to the NSE. These additional object types are "first-class" citizens of the NSE; users cannot distinguish built-in object types from those that have been added.

4.2.1 Type-Specific Operations

To implement a new type within the NSE, the type integrator writes a collection of procedures. These procedures implement a set of type-specific operations defined by the NSE. The type integrator then compiles the procedures and links them with the NSE. Types may also inherit procedures (methods) from a parent type.

Figure 4 shows how the code that implements an NSE command is divided into four levels of procedures. When you invoke an NSE command such as `acquire`, you start the execution of a program of the same name. This main program parses command line arguments and then calls a corresponding *generic procedure*; for example, `acquire` calls `nse_generic_acquire`.

A generic procedure performs type-independent command processing and then calls a corresponding type-specific procedure based on the type of the object. Generic procedures call type-specific procedures indirectly through a mechanism called an *ops vector*. There is one ops vector per operation, and each vector contains the addresses of the procedures that handle each object type. For example, the `acquire` ops vector contains the addresses of the component `acquire` procedure, the target `acquire` procedure, and so on. New types are accommodated by adding corresponding entries to the ops vectors, a job done by a table driven configuration utility.

There are two classes of type-specific operations, called primary and secondary. There are seven primary operations, which correspond to NSE commands such as `acquire` and `reconcile`. There are about 20 secondary operations which mainly retrieve attributes of the objects such as last modification time. Secondary operations are mainly called by generic procedures.

Note that a primary type-specific procedure may call a generic procedure. This is how the NSE recursively applies an operation to all subobjects of compound objects. For example, the component `acquire` procedure calls `nse_generic_acquire` for every object in the component.

In each environment are several small databases which type-specific and generic procedures can use to store and retrieve information about objects. In general, these simple databases are used to store object attributes and revision histories. They are usually not used for the contents of an object; these are stored in the same way they were before integration with the NSE. In particular, files are stored as files in the underlying file system. As explained below, this greatly simplifies tool integration.

4.2.2 Inheritance

The NSE allows a limited form of operation inheritance. A new type may use some or all of the operations of an existing type. In the simplest case, only the type name need be different. This type renaming is surprisingly useful, since it allows, for example, simple file types to be managed by the NSE with very little work.

4.2.3 Tool Integration

Tools should be able to integrate with the NSE (i.e., have their objects managed by the NSE) without sacrificing the tool builders' freedom to represent objects the way they want. To support this desire, file objects in an NSE environment are accessed using the same interfaces as files outside of NSE. CASE tools that use files to store their objects need not change their data representations in order for their objects to be managed by the NSE. Integrating such tools into the NSE framework requires minimal modification (often none). If the tool creates objects whose presence cannot be discovered by the corresponding type-specific operations, the tool must register the objects in the databases maintained by the NSE. Tools that can delete or rename objects must also register those events. This allows the NSE to maintain object revision histories for deleted or renamed objects.

5 Heterogeneous Distribution

From the viewpoint of a single user, NSE environments are divided into two classes, *local* and *remote*. The user activates, does normal work in, and makes revisions in

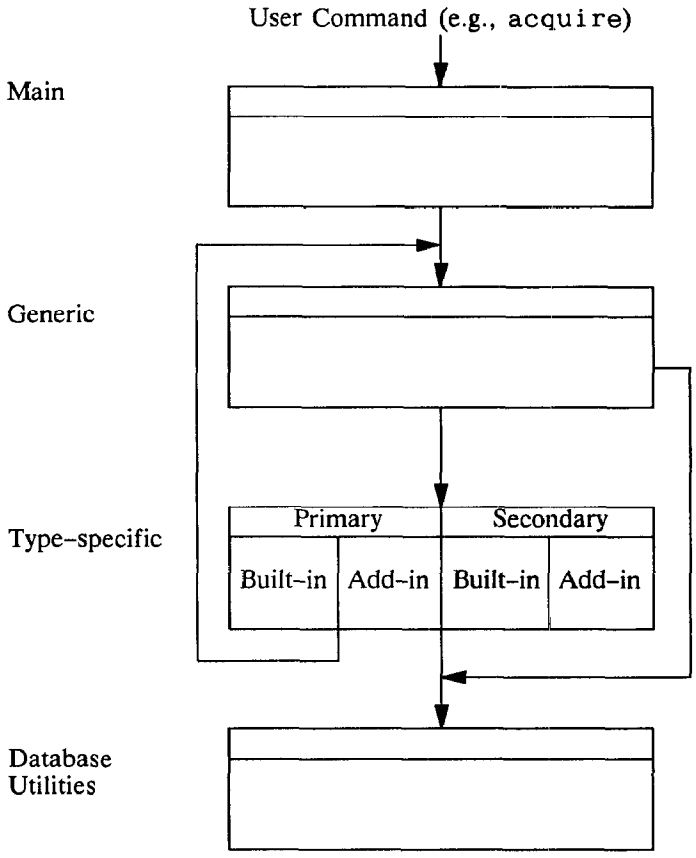

Figure 4 - Command Procedure Calling Hierarchy

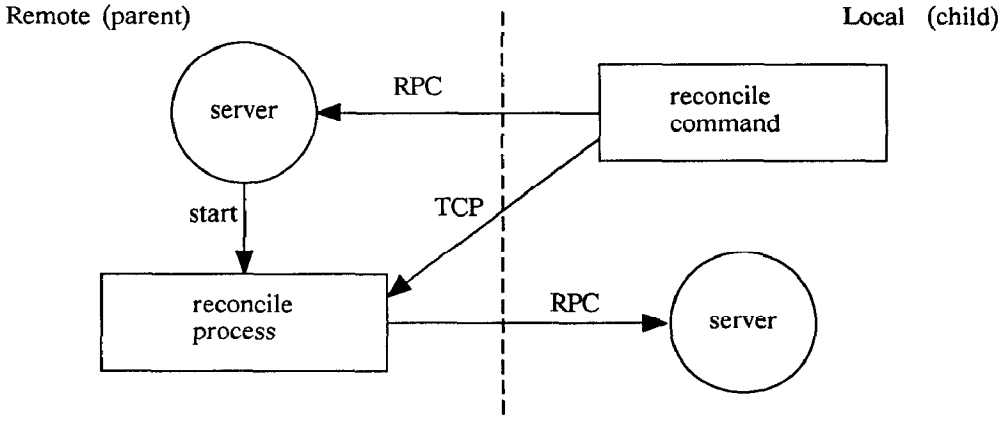

Figure 5 - Reconcile

local environments (the files and databases which make up a local environment need not be on the users machine but must be accessible by NFS, the Network File System). While working in a local environment, the user may acquire (or resync) from or reconcile to a *remote* environment.

The NSE programs which perform `acquire` or `reconcile` never access the remote environment directly. They communicate via remote procedure call (RPC) with a server running on the machine at which the environment is located. The remote server supplies information about objects in its environment but never makes changes. There is never more than one server per environment, allowing the server to prevent overlapping operations on the same environment.

The `reconcile` operation requires that the reconcile command produce changes in the parent environment, which it cannot make directly. Figure 5 shows the process structure which preserves the local/remote distinction during `reconcile`. The steps are:

1) The `reconcile` command invokes the `reconcile_check` operation which uses information obtained from local databases and the parent's server to verify that the `reconcile` may take place (that there are no conflicts).
2) The command then asks the server to start a reconcile process local to the parent environment and sends that process a list of objects which need to be reconciled. The server itself does not perform the `reconcile` operation in order to free itself to serve other requests while the `reconcile` is in progress.
3) The reconcile process executes the `reconcile` operation which makes requests to the child's server in order to obtain the new objects (the process is the reverse of `acquire`).

By carefully preserving the local/remote distinction, the NSE allows `acquire/resync/reconcile` to work between environments with widely varying implementations. Except during those operations, there is no communication between environments and local work can continue even if the communications channel is interrupted.

6 Relation to Other Work

A number of software tools have addressed some of the problems NSE addresses. SCCS [1], RCS [13], and DSEE [10] address software engineering problems for software objects that are individual files, primarily source code files (although DSEE also handles object code files). These systems provide no integrated method for managing files along with other types of objects.

SCCS, RCS, and DSEE all use file locking to control concurrent updates to files. Locking, as mentioned, has the disadvantage of serializing development. Developers are forced to wait until locks on files they need to modify are released. Locking also does not adequately protect a developer from inconsistencies imposed by changes made by others on files the developer depends on but is not updating.

RCS can associate symbolic names with versions of files that belong to a revision, so consistent versions can be tagged with a common name. DSEE configuration threads provide a mechanism for tools to transparently access versions of files for a particular configuration. SCCS controls versioning of individual files but not sets of files.

To handle multiple releases, SCCS, RCS, and DSEE support the concept of branching version histories for individual files. Because software development involves multiple files, however, branching needs to be supported for sets of files, rather than just on individual files.

A system closer to the NSE is the DF system [9, 11] found in Cedar [6]. The DF system provides mechanisms for defining sets of consistent files. Developers copy a set of files to their workstations and modify the copies in isolation. The modified copies are then copied back to be merged with the originals. The DF system does not provide the same degree of support for logical work spaces as the NSE does. For example, it uses physical workstation disks as work spaces.

PCTE[5] is a more general object manager but does not provide a mechanism for isolating users working in parallel. The DAPSE[2] work does define methods for identifying consistent sets of objects and for controlling parallel changes. However, its approach to tool integration is the opposite of the NSE's: existing tools must be modified to conform to DAPSE's interfaces.

In contrast to NSE's copy-modify-merge paradigm, Pu, Kaiser and Hutchinson [12] have recently introduced the idea of split-transactions to reduce serialization in long transactions. The basic idea is to allow a transaction to be split so that part of the data can be released for others to access without waiting for the entire set of data to be freed. Split-transactions promote parallel development,

but the degree of parallelism is limited to the extent that splitable subsets of the data can be identified. The copy-modify-merge paradigm does not have this limitation.

Type extensibility has been a technique used in programming languages for a long time. NSE's adaptation of this technique to management of software objects makes it similar to ISTAR [3], although ISTAR is more oriented towards project management.

Semantic-based program merge techniques using data flow analysis have been recently investigated by Horwitz, Prins and Reps [8]. Their methods, however, are not yet suitable for general programming languages, such as C.

7 Current Status

The first version of the NSE has been implemented and released as a product and is in use at Sun and at a number of large software development groups outside of Sun. The NSE was developed using itself (180,000 lines of source code) for the last 18 months. Work is in progress to implement NSE types for objects produced by a number of existing software development tools.

8 Conclusions

The NSE provides a uniform mechanism for managing the development and maintenance of different kinds of software objects. Not only can different objects types be managed, but types can be combined in components that are managed together as a unit. New kinds of objects can be added to the NSE. The copy-modify-merge paradigm used by the NSE supports parallel development without the use of locks. The NSE provides the necessary bookkeeping and tools needed to facilitate parallel development. Experience has shown that the cost of merging when using the NSE is less than the cost of delays due to serialization caused by locks. Finally, the NSE is architected to allow development in a heterogeneous distributed network.

9 Acknowledgments

This paper reflects the work of the entire NSE team at Sun Microsystems. The people involved, in addition to the authors, include Azad Bolour, Jonathan Feiber, Jill Foley, David Hendricks, Tom Lyon, Russell Sandberg, and Daniel Scales.

10 References

[1] Bell Telephone Laboratories, "Source Code Control System User's Guide", *UNIX System III Programmer's Manual*, Oct. 1981.

[2] S. Boyd, "Status of the DAPSE Project: A Distributed Ada Programming Support Environment", *ACM Sigsoft Software Engineering Notes 12(3)*, April 1987.

[3] M. Dowson, "ISTAR — An Integrated Project Support Environment", *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Jan. 1987, pp. 27-33

[4] S. I. Feldman, "Make - A Program for Maintaining Computer Programs", *Software: Practice and Experience*, April 1979.

[5] F. Gallo, R. Minot, I. Thomas, "The Object Management System of PCTE as a Software Engineering Database Management System", *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering on Practical Software Development Environments*, April 1984.

[6] D. K. Gifford, R. M. Needham, and M. D. Schroeder, "The Cedar File System", *CACM* March 1988, pp. 288-298.

[7] D. Hendricks, "The Translucent File Service", *Proceedings of the European Unix Systems User Group Autumn 1988 Conference*, October 1988.

[8] S. Horwitz, J. Prins, and T. Reps, "Integrating Non-Interfering Versions of Programs", pp. 133-145, *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages*, (San Diego, CA, Jan, 13-15, 1988) ACM, New York, NY (1988)

[9] B. Lampson and E. Schmidt, "Organizing Software in a Distributed Environment", *Proceedings of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems*, June 1983, pp. 1-13.

[10] D. B. Leblang and R. P. Chase, "Computer-Aided Software Engineering in a Distributed Workstation Environment", *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering on Practical Software Development Environments*, April 1984, pp. 104-112.

[11] B. Lewis, "Experience with a System for Controlling Software Versions in a Distributed Environment", *Proceedings of the Symposium on Application and Assessment of Automated Tools for Software Development*, Nov. 1983.

[12] C. Pu, G. Kaiser, N. Hutchinson, "Split-Transactions for Open-Ended Activities", *Proceedings of the Fourteenth International Conference on Very Large Data Bases*, Aug. 1988.

[13] W. F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System", 6th ICSE, Sept. 1982.