# VS: An Optimistic Version Management System

*Charles A. Adams*
*B.A., California Polytechnic State University at San Luis Obispo*
*M.A., West Virginia University*

A thesis submitted to the faculty
of the Oregon Graduate Institute of Science and Technology
in partial fulfillment of the requirements for the degree
Master of Science in
Computer Science

December, 1990

The thesis "VS: An Optimistic Version Management System" by Charles A. Adams has been examined and approved by the following Examination Committee:

Robert G. Babb II, Ph.D.
Professor
Thesis Research Advisor

Jonathan Walpole, Ph.D.
Assistant Professor

Errol Crary, M.B.A.
Senior Engineer, Tektronix, Inc.

# ACKNOWLEDGMENT

This thesis was made possible by the support and encouragement of many people. I would like to express my appreciation first and foremost to my wife, Gloria, who provided unwavering support. Robbie supplied much needed guidance and vision throughout my term at Oregon Graduate Institute. Jonathan and Errol furnished invaluable help on this thesis. Finally, I need to thank Kevin Jagla who provided numerous hours of camaraderie during our parallel journeys.

# ABSTRACT


VS: An Optimistic Version Management System

Charles A. Adams
Oregon Graduate Institute of Science and Technology

Supervising Professor: Robert G. Babb II, Ph.D.

Designing, implementing and maintaining large complex software systems can be a difficult task. The emergence of computer networks has made the task even more complex. Managing large software engineering teams has proved to be problematic. Over the years researchers and engineers have developed tools and techniques to improve the productivity of these engineering teams.

VS is a prototype version management system. This study of version management used VS as an example of an optimistic version management tool to analyze the effects an improved tool could have on software engineering productivity. Specifically, the study looked at the domain-specific requirements of Large-Grain Data Flow 2 to develop mechanisms that could help improve productivity during software design and analysis. It also investigated the domain-independent policies and mechanisms needed within the software engineering field.

# Table of Contents

# List of Figures

# List of Tables

# 1. Introduction

Productivity within the software engineering field has been a significant topic for many years. Researchers, engineers and others continue to search for methods and tools to improve productivity within this field. Fredrick Brooks has noted it is likely that there is no "silver bullet" that by itself can provide order-of-magnitude improvements in software engineering productivity [Brooks, 1987]. Thus, substantial gains in productivity are likely to be the result of incremental developments in a variety of areas.

In the past decades, we have witnessed many improvements in software technology. The introduction of structured programming languages, user interface development tools and other software tools have helped improve individual software engineers productivity. Recently, much energy has been expended in developing integrated software development environments. Tools, such as Apollo Computer's Domain Software Engineering Environment (DSEE) [Leblang, 1985], and Sun Microsystems' Network Software Environment (NSE) [Miller, 1989], have shown promise in helping improve software engineering productivity by providing assistance during product development and release. Current research, using tools such as Cosmos [Walpole, 1988], has provided additional insight into these phases of a project.

On the other hand, studies of the software development process have noted that close to three-fourths of all errors occur in the design stage of the product life-cycle [Boehm, 1976], [Hamilton, 1976]. Robert Bailey has claimed that almost half the errors made during the operation of a new system are the result of faulty design decisions [Bailey, 1983]. These errors often lead to major loses in productivity. Barry Boehm has asserted that the cost to fix design errors during the later stages of prod-

uct development is 50 to 100 times higher than fixing them during the design stage [Boehm, 1988]. The recent focus on rapid prototyping is in part due to the need to get the requirements right early in the project life-cycle and thereby reduce the amount of rework required later in the project [Boehm, 1987]. Penny Nii argued in a recent monograph "the coding crisis of the 1960's and 1970's has turned into the design crisis in the 1980's" [Nii, 1990].

Major gains in software productivity are expected to occur, if gains in productivity are achieved in all stages of software production. Better design tools, such as the Visible Analyst Workbench and System Architect, have helped with the production of diagrams for structured design methodologies [Nii, 1990]. The use of Data Flow Diagrams to represent software designs has become a standard part of many large software development projects. Yet, these tools and techniques do not by themselves fulfill the designers' need for accommodating, communicating and controlling changes during product design. For designers to be as productive as possible, the software design environment must become a medium for communication which integrates people, tools and information [Curtis, 1988].

## 1.1 Integrated Software Development Environments

Software development environments are being investigated by an increasing number of researchers and engineers. The number of tools for configuration management is growing almost daily. One type of tool used in all configuration management systems, the version management tool has been around for many years. For instance, the Source Code Control System (SCCS) dates back to 1979 and the Revision Control System (RCS) was developed in the early 1980's. These two tools have found

wide acceptance in the UNIX™ community. The recent development of tools like DSEE, and NSE provides a means for integrating source code development and systems management. In general, these configuration management tools provide the glue to integrate version management with system construction and maintenance.

The field of computer-aided design of mechanical and electrical systems has also been examining its need for version management tools. In the most general sense, the issues of version management in this related field are similar to the ones in software design [Katz, 1990]. In each case, the changes in an object need to be tracked, and controlled. Also, many of these changes must be addressed in a timely and appropriate fashion by the affected development team(s). On the other hand, the artifacts produced by a chip designer, a programmer and a software designer are significantly different. Thus, one should expect that there will always be the need for domain-specific tools.

Although much has been written about version management tools and policies needed during the development and maintenance of source code and chip designs, little has been written about version management for software designs.

## 1.2 Integrated Software Design Environments

Software projects are growing in size and complexity. As the number of designers on a project grows, the need for tools to assist in tracking, controlling and communicating change also grows.

Version management of software designs is as important as the version management of source code or customer documentation for a variety of reasons. First, as at-

---

™ UNIX is a trademark of AT&T Bell Laboratories, Princeton, New Jersey.

tempts are made to increase software engineering productivity through the use incremental development, each incremental change is expected to be as error free as possible. In addition, since this technique is characterized by a great many small changes there is an increased need for communication between the affected engineers. Second, as attempts are made to increase productivity through software reuse, that software's design becomes more important. If a module is to be used effectively, each new project or engineer using the module will need an up-to-date and easy to understand design for that module. And, when changes are made in the design and thus the module being reused, the effect of those changes can be widespread.

We are now seeing situations where projects overlap so much that the design of the software being used by one project team is simultaneously being modified by another project team. This need for concurrent read and write access to shared software introduces new complexity into the management of software designs and source modules. Typically, these teams are trying to cooperate with each other, but often they are forced into competing for shared resources. It has been asserted, that this competition can be reduced through the use of better version management tools [Berliner, 1990].

The tools for version management need to be flexible and powerful because they are needed by a wide variety of software design projects. Each project has somewhat different requirements. Tools for version management and tracking must meet both domain-dependent and domain-independent requirements of these projects.

## 1.3 Outline of Study

This thesis reports on an investigation into the requirements for version management. To aid this investigation VS, an example of a version management system,

was developed for use with Large-Grain Data Flow 2 diagrams. This study shows that such a tool can be constructed and could be used for version management. This work indicates that the use of version management tools of this type at the design stage could help increase the productivity of software design teams.

Chapter 2 examines the current approaches to version management and structured design/structured analysis. Chapter 3 presents the implementation of the version management tool used for this research. Chapter 4 shows examples of how this tool could be used in a large-scale software design effort. Chapter 5 gives a critique of the accomplishments of this research and an assessment of how this research could fit in with future investigations of software engineering.

# 2. Current Approaches

While most research on improving productivity in software engineering has treated the software design process and version management as distinct, this investigator asserts that studying the overlap will help us in further improving productivity. Research on version management of source code has shown that it can contribute to improved productivity during the code development and maintenance phases of a project. It is thus expected that techniques of version management for software designs could lead similar or greater improvements, if they were tailored for use in the design process. These tools could assist in communication, control and maintenance.

## 2.1 Version Management

The following review of version management has been divided into three parts:

- a review of several text-based version management tools
- an examination of several configuration management tools
- a look at several tools for version management used within the computer-aided design field

This introduction to version management serves as the basis for the development of VS. It also provides the basis for the analysis of the example scenarios used in this research.

### 2.1.1 Text-based Version Management Tools

A significant early version management tool was the "Source Code Control System (SCCS)" [Rochkind, 1975]. This system was developed to:

- reduce the amount of space needed to store multiple versions of a single

piece of source code

- ensure fixes that need to be applied to other versions are propagated properly
- make changes readily identifiable
- provide for the cross reference of changes across modules

This tool provided these features by:

- storing all the changes to a module in a single file
- giving programmers the means to protect source modules from updates
- implementing an easy method for identifying versions
- storing who made the change, what the change was, when the change was made, and why the change was made

This work introduced the concept of software change "deltas." In SCCS, each time a module is changed, the difference between the new version and the old version is stored, see Figure 2-1.

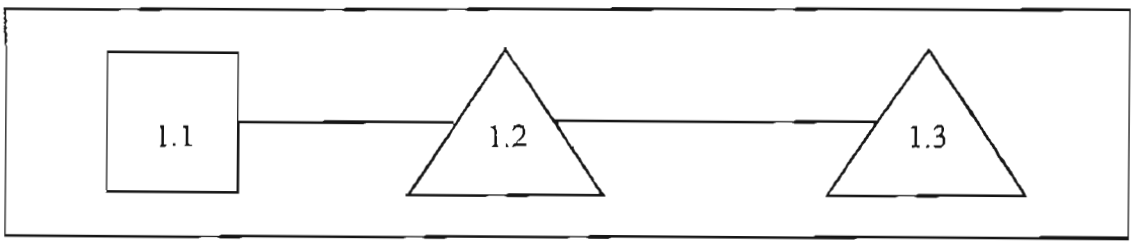


**Figure 2-1. SCCS Deltas.**

SCCS's protection scheme allowed programmers to set long term locks on particular deltas; thereby, reduce the risk of someone damaging work on a delta that was in progress. This scheme was expected to be supplemented by more sophisticated protection offered by the operating system and project teams.

SCCS proved extremely useful to experienced programmers, who needed to identify how changes introduced problems into the software systems that they were developing or maintaining. These facilities for documenting and identifying changes have made SCCS very popular.

A second notable development in version management occurred with the development of the "Revision Control System (RCS)" [Tichy, 1982]. The goals for RCS are very similar to those for SCCS. The improvements noted in RCS have proved themselves over the years that this system has been in use.

RCS introduced separate, reverse deltas to improve its retrieval performance. It featured increased module protection by preventing two or more persons from putting their changes on the same revision. RCS also provided a mechanism for branching of deltas, see Figure 2-2. This branching mechanism has helped multiple users and multiple projects work in parallel using shared modules.

This branching mechanism allowed the development of revision trees. The main branch, i.e., those revisions numbered 1.1, 1.2, ..., 2.1, 2.2, etc., were called the *trunk*. The highest numbered revision on the trunk, i.e. the most current revision, is referred to as the *top-of-trunk*. Each branch from the trunk is forked from a particular revision. Thus, the numbering scheme for a branch includes the revision from which it emanated, the branch number, and its location on the branch. For instance, the revision 1.1.1.2 emanated from the revision 1.1, is the first branch from that revision, and is the second revision on this particular branch.

RCS, like SCCS, provides a long term locking mechanism to prevent two or more users from placing competing changes on a revision. It is expected that each user, who will be making changes to a controlled module, will want to set a lock on the revi-

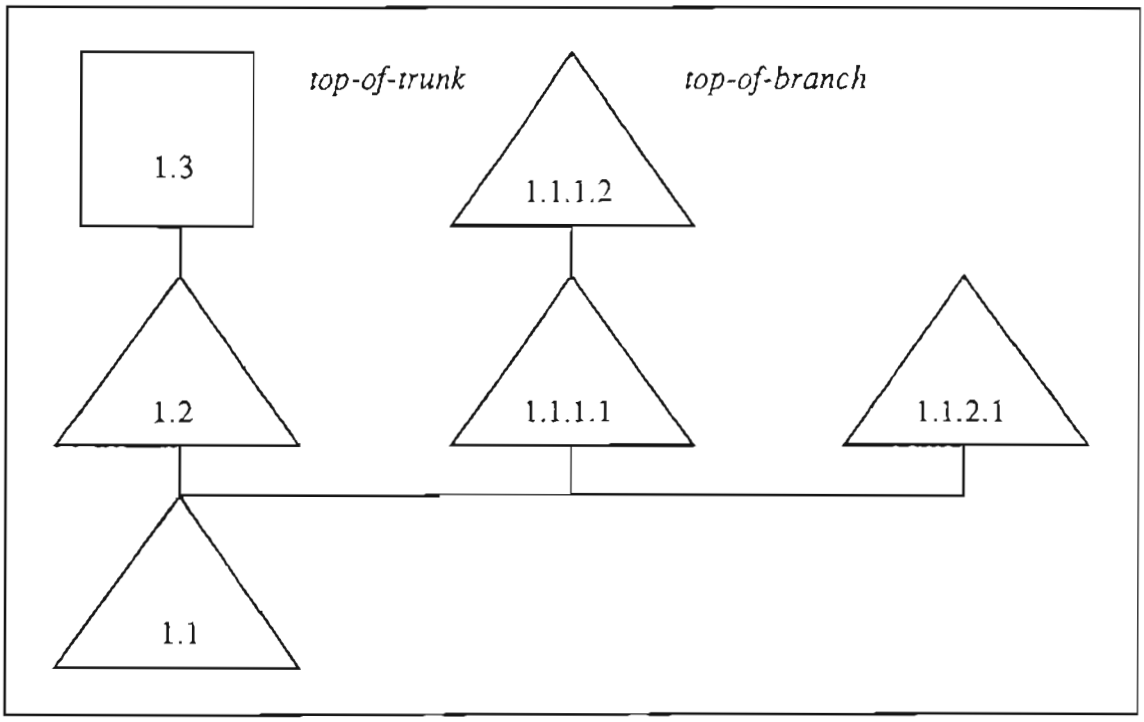


**Figure 2-2. RCS Deltas with Two Branches.**

sion at the time of check out, make modifications and then release the lock at the time of check in. In RCS, only one user at a time can own the lock on a revision. Thus, while the revision is locked, it is under exclusive control of the locker.

This locking mechanism provides users with a means to enforce a first-to-check-out, first-to-check-in policy. This mechanism helps ensure that each revision will be based on the revisions that preceded it, because if the user is required to acquire the lock at check out and release the lock at check in, all changes will be based on the preceding ones.

This mechanism has proved very useful for projects that have users that tend to step on one and another's changes, or that have users who rarely need concurrent update access to shared modules. This mechanism essentially assures that changes

done by each engineer are done with knowledge of all the preceding work. With this mechanism, branching and merging of branches are rarely needed because each user is working serially. The existence of a lock can also serve as a flag to others that changes are to be expected.

RCS also provided a command, *rcs -l*, which allows a user to place a lock on a revision. Thus, a person could do a check out, make changes, place a lock and then do a check in. This means that even though a file is locked at the time of check out, the user can proceed but then it is up to the user to ensure that his or her changes do not unknowingly "undo" previous work. In this case and when merging changes from branches, checking the changes for errors and conflicts can sometimes be a non-trivial task; because, the changes may be very subtle causing subtle execution errors, or the size of the change may be very large, in which case it can be difficult to separate the important changes from the chaff.

### 2.1.2 Configuration Management Tools

Recently version management has been studied within the auspice of configuration management. Configuration management is concerned with system construction, version management and the management of derived objects and releases. Systems, such as Cosmos, DSEE and NSE, have been developed to provide version management mechanisms within an integrated software development environment.

Apollo Computer's "Domain Software Engineering Environment (DSEE)" [Leblang, 1984] was one of the first systems to provide this integration. Leblang pointed out that in large complex development projects that may span multiple projects, users need to be able to isolate themselves from some changes and at the same time share as many modules as possible [Leblang, 1987]. DSEE also gave us-

ers the ability to specify who is to be notified when changes are made. This becomes very useful when project team members need to know about updates that affect them at the time they are made. This timely communication of changes can reduce the time wasted in working with an out-of-date module. In DSEE, the change in the local copy of the system could be made automatically by a cooperating program or at the discretion of the affected user.

The work on "Cosmos," an integrated software development environment [Walpole, 1988], [Walpole, 1989], addressed the integration of version management tools with concurrency control. Cosmos provided a mechanism for long term transactions based on cooperation rather than competition. Whereas, DSEE provided distributed, concurrent read access to shared modules, Cosmos also provided a mechanism which gave multiple users concurrent update access to shared modules. Using an immutable object model, Cosmos was able to guarantee the consistency of these long term transactions. Immutability was achieved by leaving old versions untouched and always creating a new version when a change was made. This process was labeled "transformation."

The use of immutable objects gave Cosmos two significant advantages over previous systems. First, Cosmos was eminently suitable for supporting multiple users and versions because of the concurrency control provided by the transaction mechanism. Second, the problem of consistency between related objects was reduced to a problem of naming groups of consistent immutable versions of objects. A configuration object is provided by Cosmos to name groups of objects which define consistent domains [Walpole, 1989].

Like each of these previous systems, Sun Microsystems' "Network Software En-

vironment (NSE)" provided a set of network-based version management commands. Thus modules could reside anywhere in the available workspace. A user of NSE works with design objects, typically files, and a configuration which is a collection of one version each of the configuration's design objects. Object management in NSE is based on a *copy-modify-merge* paradigm. Locks are not used because modifications are merged onto the controlled original in a serial fashion.

A large number of other systems have been proposed and implemented in the past few years. Systems, such as CVS II [Berliner, 1990], and Software BackPlane™ [Black, 1989], have implemented configuration management and version management features similar to the ones mentioned above.

### 2.1.3 Version Management Tools for Computer-Aided Design

The field of computer-aided design (CAD) has shown considerable interest in version management of electrical and mechanical designs. Like software engineers, these designers must manage large, complex, interrelated objects that change over time [Katz, 1990]. Although the artifacts being manipulated by each of these users may be unique, their version management systems must meet similar requirements for storing, retrieving and communicating changes.

Many version management schema for CAD artifacts have some notion of a workspace, i.e., "a named repository for design objects" [Katz, 1990]. A workspace can be either private, shared or archival. A private workspace belongs to an individual designer. A shared workspace is used by group of designers working in parallel. An archival workspace is used for storing and retrieving versions from public stores.

---

™Software BackPlane is a trademark of Atherton Technology, Sunnyvale, California.

Change notification is also important to large CAD projects. ORION [Banerjee, 1987], a prototype object-oriented database system, was developed at the Microelectronics and Computer Technology Corporation. ORION contains an change notification mechanism that has been integrated with the object-oriented database [Chou, 1989].

ORION incorporates both message-based and flag-based techniques for change notification. When the flag-based approach is used, a data structure is updated and the affected users are not notified of changes until they access the object. With the message-based approach, users are notified of changes as they occur. With this later approach the users can further select either immediate or deferred notification. Finally, with this approach the notification can be based on the type of change made.

Version Server developed at the University of California, Berkeley provides version management services for the electrical CAD designs [Chang, 1989]. It organized design elements into collections of component hierarchies and version histories and provided for equivalences. This system's interface was quite similar to that of NSE but did not provide as comprehensive an environment as DSEE or NSE. It did however, provide facilities to handle a much wider range of data types than these other systems.

Many others have been working on the problems of configuration and version management of computer-aided design artifacts. The recent monograph by Randy Katz presents a survey of the version models developed for the computer-aided design field [Katz, 1990].

## 2.2 Motivation for Management of the Design Process

Structured Design and Structured Analysis techniques have been advocated as a method for managing the software design process since the middle 1970's. Early proponents of software engineering, such as Victor Basili, Barry Boehm, Edward Yourdon, have generally agreed that these techniques can help increase the productivity of software designers [Basili, 1978], [Boehm, 1976], [Yourdon, 1979], [Zelkowitz, 1979].

An early structured design tool PSL/PSA (Problem Statement Language/Problem Statement Analyzer) [Teichroew, 1977] proved to be effective for formalizing system requirements of large business-oriented applications. Users experiences with this system have shown that the system helped force discipline on the designer [Reifer, 1978]. In addition, it assisted in the identification of errors in requirements specifications.

Whereas PSL/PSA was primarily textual, SADT™ (Structured Analysis and Design Technique) is a manual, hierarchical, graphical system for software design [Ross, 1977]. Users of SADT have found it beneficial, allowing the end-user (i.e., someone without a formal software background) to evaluate designs [Combelic, 1978]. This evaluation can help decrease the cost of software and increase overall quality. Often the diagrams proved more useful than the accompanying prose.

The use of Data Flow Diagrams (DFDs) to reduce the complexity of the software design effort has been advocated by many over the years [DeMarco, 1978], [Page-Jones, 1980], [Babb, 1982]. The early research on DFDs [Babb, 1982], [Babb, 1984] has shown that they could reduce the design errors made and help in producing more

SADT is a trademark of SofTech, Inc., Waltham, Massachusetts.

efficient programs. This systems approach to design has now become widely used throughout the software engineering field.

Further formalization of the Data Flow paradigm has been introduced by David Di-Nucci and Robert Babb as Large-Grain Data Flow (LGDF) [DiNucci, 1988] and LGDF2 [DiNucci, 1990]. Large-Grain Data Flow 2 (LDGF2) is a declarative graphical language in which a program or software system is represented as an oriented graph. Processes are represented as circles. Dataswitches are represented as vertical rectangles selectively connected by pairwise arcs. The dataswitch contains the data which is cross referenced to the data dictionary associated with the design and the read/write permissions for that dataswitch. An example network is shown in Figure 2-3.

LGDF2's language syntax is based on the F-Net model of Portable Parallel Software Engineering [DiNucci, 1990]. Each F-Net contains a set of variables[1] ("dataswitches"), a set of operations ("processes") and set of instructions ("process calls") which reference both operations and variables.

An F-Net variable contains both the imperative-language variable and the finite-state machine. Thus each variable possesses both its data state and its control state. A data state is equivalent to the data value in a traditional programming language. A control state is equivalent to the current state of an element within a finite-state machine.

An F-Net operation contains a signature and an implementation. The signature provides access to the variables connected to the process. It also identifies whether the operation will use each variable for reading, writing, both reading and writing, or

[1] Variables are often referred to as datapaths or switches.

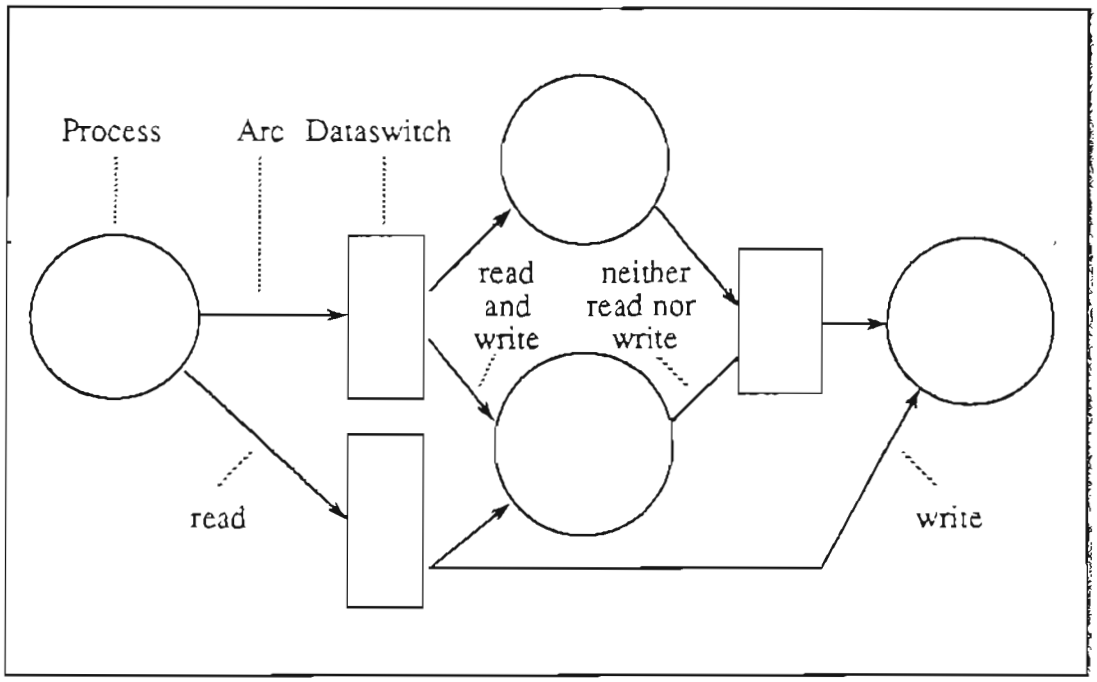


Figure 2-3. LGDF2 Network Example with Assumed Transitions.

neither reading nor writing. Finally, the signature contains the allowed transitions for each variable.

Each F-Net instruction provides a mechanism for instantiating an operation, by binding each of its arguments to an F-Net variable and each transition to that variable. The instructions thus control when an operation can be invoked by naming the control state of each variable to which it arguments are bound.

The research on LGDF has shown it to greatly simplify debugging complex parallel programs [DiNucci, 1988]. This new formalism shows much promise for improving productivity in the production of parallel processing software and algorithms.

## 2.3 Incremental Development and Software Reuse

The promotion and use of incremental development [Boehm, 1981] of large com-

plex software systems places even more burdens on the design phase of projects. Developing software in increments forces designers to scrutinize each module at every step during its development. With this approach, each increment of software that is designed and implemented must fit together with software that will be designed and implemented in the future. Because most engineers are better at modifying that which is already designed than they are at predicting the future, it is essential that the design team be able to modify designs and modules quickly and painlessly as possible.

"Advancemanship" techniques of software development [Boehm, 1981] require designers and programmers to specify the software scaffolding that is needed for any large project. This scaffolding can include "dummy design elements" and "dummy modules" which will be completed later in the project. These incomplete designs and modules are used until there is a clear and present need for more complete and thoroughly analyzed designs and modules. This paradigm forces the project team to track and control the versions of the objects being used. Without version management, modifications may be lost or modules may be used which are out-of-date.

Today, there is a concentrated effort within the software engineering community to increase the amount of software reuse [Boehm, 1987], [Boehm, 1988]. A module that can be shared across projects is a module that increases the productivity of the second through $n^{th}$ project using that module. On the other hand, if a bug is discovered in a shared module, its effect also goes across projects thereby having the opposite effect. This software reuse is further complicated by the need to modify modules as the requirements for those modules change. As modules change, some projects will need the changes, and some projects are going to want to continue to use their

own version. Thus, the design of that particular version is as important as the design of the most current version or any other version on the change hierarchy.

# 3. Implementation

The work reported in this thesis is based on the version control tool called RCS [Tichy, 1982]. RCS is available on many implementations of the UNIX Operating System. VS, as a prototype version management tool, extends RCS to handle the issues involved with large-scale, multi-person, multi-project, network-based software development. VS provides both domain-independent and domain-dependent facilities for version storage, control and tracking.

This implementation of VS was used to study two questions. First, can a domain-specific tool for distributed, optimistic version management be built on top of a readily available version control tool? Second, will the use of such a tool improve productivity in software engineering?

## 3.1 Overview of VS: An Optimistic Version Management System

VS is a user program written in the C programming language that runs under UNIX. It uses the UNIX system call interface mechanism to issue the RCS commands. For instance, VS can issue the RCS check in (`ci`) and check out (`co`) commands. VS uses the mechanisms provided by RCS for storing and retrieving text, logging changes, identifying versions and controlling access to files under version management. It also extends the basic RCS capabilities. These extensions are described in detail below.

VS uses an interface[1] similar to RCS except that the interface was formalized to allow it to be used by either end-users or by other programs using the same protocol.

[1] For this prototype a simplified user interface protocol was used. This protocol was adopted to make the development of VS easier.

The interface protocol consists of either workspace commands or version management commands. The User Manual for VS is provided in Appendix C. A UNIX man page for VS is given in Appendix D.

The remainder of this overview of VS is divided into three parts. First, the workspace commands are presented. Second, an overview of the version management commands is given. Third, VS's output protocol is presented.

### 3.1.1 Workspace Commands

The workspace commands provide a limited workspace control mechanism which allows the user to query and connect to the available workspaces. The workspace commands are given in Table 3-1.

| Command | Usage |
|---|---|
| PROJECT? | Query the available workspaces |
| PROJECT: *name* | Connect to a workspace |
| GAMEOVER | End the program |

**Table 3-1. The Workspace Commands.**

The PROJECT? command provides a mechanism to query the available workspaces. The GAMEOVER command is used to exit from the VS program. The PROJECT: command takes the single argument, i.e., the name of the workspace one wishes to work in. If the workspace exists, VS places the user in that workspace. If the workspace does not exist, VS can create the workspace and then place the user in it.

The PROJECT: command is implemented using the UNIX system call `chdir`. In

this prototype, a workspace is a UNIX directory. This simplified workspace mechanism gives VS the ability to connect to either local or remote file systems.

The workspace commands were given a fixed protocol. Each identifier is exactly eight characters long to simplify parsing. The argument for the `PROJECT:` command follows the identifier and is expected to start with the argument separator, the space character (ASCII SP), and end with the command separator, a carriage return (ASCII CR). The other commands are likewise terminated with a carriage return. Once VS receives the command separator, it will parse the input line and either report a parsing error or attempt to execute the command given.

### 3.1.2 Version Management Commands

The version management commands provide access to the entire functionality of RCS with the exception of the `rcs` command. The version commands implemented are presented in Table 3-2.

| Command | Usage |
|---|---|
| CHECKIN- *arguments* | Store a new version |
| CHECKOUT *arguments* | Retrieve a version |
| DFDMERGE *arguments* | Merge a branch on to top of trunk |
| GAMEOVER | End the program |
| RCSDIFF- *arguments* | Compare two versions |
| RCSLIST- | Query the controlled files |
| RCSMERGE *arguments* | Merge differences on to a file |
| SENDRLOG *arguments* | Query the status of a file |
| ZZZZZZZZ | Return to the previous workspace |

Table 3-2. The Version Management Commands.

The version management commands were given a fixed protocol. Each identifier is exactly eight characters long to simplify parsing. The arguments for a command are expected to start with the argument separator, the space character (ASCII SP), and end with the command separator, a carriage return (ASCII CR). Once VS receives the command separator, it will parse the input line and either report a parsing error or attempt to execute the command input.

The version management commands extend the capabilities of RCS in several significant ways. Two of the commands, `RCSLIST-` and `DFDMERGE` are completely new with VS. The `RCSLIST-` command lists all of the files in this workspace that are under version management. The `DFDMERGE` command will be explained in detail later in this section. The improvements `CHECKIN-` and `CHECKOUT` offer over `co` and `ci` from RCS are presented in detail later in this section. The `GAMEOVER` command is both a version management command and a workspace command.

The other commands `RCSDIFF-`, `RCSMERGE`, and `SENDRLOG` are identical in functionality and interface to the corresponding RCS commands `rcsdiff`, `rcsmerge`, and `rlog`. They provide access to the normal functionality of RCS without impacting the extensions to RCS made by VS. These VS commands also further extend the equivalent RCS commands by providing for a simplified input and output protocol. The purpose of this simplification is to make it easier for the VS commands to be issued and understood by other programs thus extending the functionality of VS.

In order to maintain the immutability of controlled files, the RCS command `rcs` was not implemented. In particular, the `-o` option of the rcs command allows the user to delete particular revisions. With RCS, this action is not recorded and can not be undone. Since, a deletion could be detrimental to the operation of VS, this functional-

ity is not supported. Also, since the functionality provided with the `rcs` command was not needed by this study, the entire command was not implemented.

### 3.1.3 Output from VS

A simplified output protocol consisting of a set of output tokens and strings is used in this prototype. These tokens are shown in Table 3-3.

| Output | Usage |
|---|---|
| VPROMPT> | VS is expecting an input |
| VMESSAGE *string* | An informative message follows |
| VSERROR- *string* | An error message follows |
| ZZZZZZZ | There are no more VMESSAGE or VSERROR- strings |

**Table 3-3. The VS Output Tokens.**

This simplified protocol provides the user and programmatic interface for VS. The VPROMPT> indicates that VS is awaiting input. This input could then come from either a user or a program. The VMESSAGE response by VS indicates that the following text, up until the ZZZZZZZ, is an informational message or uninterpreted text from the RCS commands or other UNIX system calls. The VSERROR response by VS indicates that the following text up until the ZZZZZZZ is an error message and that the last command has failed.

## 3.2 Extension of RCS Check Out

The co command in RCS retrieves a revision from a RCS file and places it in a working file. In currently available implementations of RCS, there is no mechanism

provided to record check outs. Thus, there is no means to inquire about who is using or who has used a particular version of a RCS file. The RCS locking mechanism will indicate whether a particular version is locked, and by whom if it is locked. It however does not indicate when the lock was applied.

The `CHECKOUT` command in VS provides the currently available options of `co` along with a mechanism to record check out transactions. This command is used to retrieve revisions from a RCS file. It also logs the successful `CHECKOUT` commands issued.

This prototype has limited the `CHECKOUT` transaction to a simple linear sequence of actions on a single file. The actions taken are: lock the status file, issue the RCS `co` command, wait for the completion of the command, if the `co` command was successful, record the check out and finally, always unlock the status file.

The record of check outs is stored in a status file. Files with the suffix `-vlog` in the RCS directory are the depository of VS status history. For instance, a working file `foo.dfd` will have a RCS file `foo.dfd,v` and a VS status file `foo.dfd-vlog` accompanying it, if it is under VS control. Under RCS, the versions of the controlled file are placed in the `,v` file. In this implementation the RCS directory or link is assumed to be directly below the current working directory, i.e. the workspace, as set by the `PROJECT:` command.

The VS status file contains a record of each successful check out. The record of the check out contains the user name, the date, the time, and the version number with the indicator that this was a check out transaction. The status file is considered immutable and is updated via a transformation mechanism. This transformation is done by

VS appending the latest record on the end of the file. No records in this file are ever intentionally deleted by VS.

To ensure that the check out transaction is atomic and no records are lost because of concurrent[1] writes to a status file, the status file is first locked, updated, and then unlocked. VS uses the UNIX system call `flock` to implement this single exclusive lock mechanism. In RCS, no short term locking was implemented because the assumption was that the long term strict locking mechanism would be sufficient for concurrency control.

### 3.3 Extension of RCS Check In

The RCS `ci` command stores new revisions into the RCS file. In the implementations now available, when locking is set to "strict" (which is recommended [Tichy, 1982]), the user must first lock the tip of an existing branch before checking in the new version. This requirement for long term locks has been shown to lead to conflicts when multiple users have need for modifying shared resources [Walpole, 1988].

The VS `CHECKIN-` command provides the functionality of `ci` along with a mechanism to provide automatic branching rather than requiring "strict" locking. During a check in, VS looks at the status file for any intervening check in between the version to be checked in and the current top of trunk. If no intervening version is found, VS can append this version onto the top of trunk. For instance, if the current top of trunk is version 1.1 and version 1.1 is the current working version then this working file can be checked in as version 1.2. On the other hand, if an intervening version is found

---

[1] More powerful concurrency control mechanisms have been suggested and implemented by other researchers [Miller, 1989], [Walpole, 1989].

then VS will automatically create a new branch from the version of the working file. For instance, if the current top of trunk is version 1.2 and the working file is version 1.1 then VS will create a version 1.1.1.1, as shown in Figure 3-1. This branch numbering scheme is explained in detail in Chapter 2.

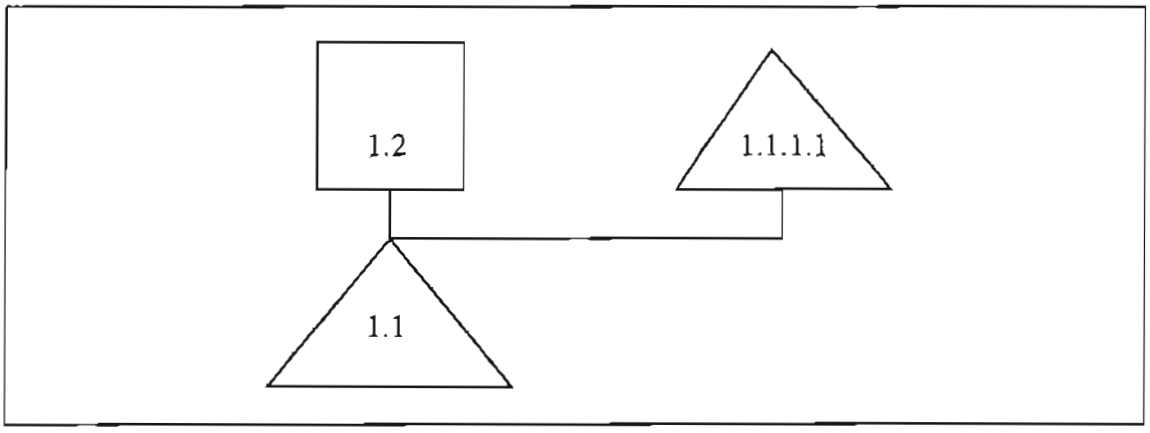


**Figure 3-1. RCS Tree After Automatic Branching.**

This automatic branching mechanism allows users to adopt a first-come, first-served check in policy as opposed the first-to-check-out, first-to-check-in policy encouraged by RCS. Thus, with VS, multiple users are allowed to have concurrent write access to shared modules since concurrent modifications can proceed because version serialization and version consistency are maintained without need for locking.

Automatic branching ensures that each version is consistent to the version it was created from [Walpole, 1989]. In the above example, both versions 1.2 and 1.1.1.1 were based on version 1.1. These versions are then parallel but consistent with respect to version 1.1. Also, the serialization of the changes is also maintained since the merging would necessarily occur after the intervening check in.

Using the above example, the changes from version 1.1 to version 1.1.1.1 could be merged on to the project's branch, i.e., the top-of-trunk. On the other hand, if the

branch 1.1.1 were to be used as the project branch, then the changes from version 1.1 to version 1.2 could be merged on top of version 1.1.1.1. In either case, the task of merging is done at the discretion of the user.

This is an optimistic mechanism for three reasons. First, it assumes that the merging required can be completed satisfactorily. In cases, where it is expected that the merging will not be possible, the user can elect to use the first-to-check-out, first-to-check-in mechanism provided with RCS. On the other hand, often the merge can be completed quite easily either by hand or with help from tools, such as DFDMERGE implemented as part of VS. Second, conflicts during merging are expected to be rare [Berliner, 1990]. In most cases, the versions created are going to be the result of parallel, non-overlapping development. For instance, two users may want to change the file `foo.c` but it less likely that they both will want to modify the same element of that file. Finally, the forced branching is expected to rarely be needed [Berliner, 1990]. In cases where work is being done serially, no overlapping updates would be noted and thus forced branching would not be used. In addition, locking would not be needed since no concurrent update conflicts would be noted.

It is believed that this optimistic check in policy will encourage more parallel development, and thereby increase productivity for a variety of reasons. First, users will not be forced to wait for any locks to be released before checking in their changes. This policy should also encourage users to make many, small changes which is encouraged by the incremental development methodologies [Boehm, 1981]. Finally, locks will not have to be broken, (released by the super-user). With RCS, a user can keep a version locked indefinitely and thus a situation can arise which requires that the lock be released arbitrarily. This situation can never arise with the use of this op-

timistic mechanism.

Since VS has a record of each successful check out, a variety of change notification policies based who is currently accessing a particular version could be enforced automatically. In this prototype, a mechanism is established that automatically notifies the user at the time of check in of other users who have check outs logged against the version that was used as the basis for the new version. It is then up to the user to notify the other users that the modification has been completed.

More sophisticated change notifications mechanisms could be built on top of VS using policy enforcing systems such as FOREST [Garlan, 1990]. For instance, one could use FOREST to monitor the output from VS, and then have FOREST automatically notify users of changes based on the policy established. Systems, such as ORION [Chou, 1989], have provided other change notification mechanisms, such as restricting the mail messages to the persons on the notification list for each version.

Like the `CHECKOUT` command, the status file is updated upon successful conclusion of every check in. The same locking and transformation mechanisms developed for `CHECKOUT` are used by `CHECKIN-`.

### 3.4 Extension of RCS Merge

The RCS commands, `rcsmerge` and `co -j`, provide a means for incorporating changes between two versions into a third version. In the case of `rcsmerge`, the modification is done on the working file specified. Whereas, with `co -j`, a working file is created as result of the command. In both cases, the modifications are done automatically. It is then up to the user to decide how to proceed once the working file has been updated or created.

A typical command sequence for `co -j` would be:

```
co -l1.3 -j1.2.:1.2.1.1 foo.dfd
```

This would cause the version 1.3 to be locked and the changes from version 1.2 to version 1.2.1.1 would be applied to version 1.3. The working file could then be examined by the user. The user then has the option of accepting, rejecting or modifying the working file.

In current versions of RCS, this examination must be completed by hand. Typically, a user would do a comparison of version 1.3 to the current working file to see if the modifications meet the requirements and policies of the project being worked on. This examination can be extremely labor intensive. It can also be error prone; especially, when the examination requires a human to check a large number of subtle differences that often are result of the merge. In addition, this examination often takes an expert who can distinguish between changes that are cosmetic and those that could significantly impact the project.

In VS, a new command `DFDMERGE` has been implemented which provides assistance in this merging process. The `DFDMERGE` command executes a RCS `co -j` command and checks the differences between the working file and the current top-of-trunk version. This implementation provides a mechanism to enforce a policy on which changes to accept, which to reject and which to refer to human authority. This mechanism is provided in the form of a decision tree. The decision to accept, or reject each portion of a change is based on the hard-coded checking done by VS.

In the prototype implementation, the following policy is enforced:

- Accept all changes made to comments within the LGDF2 file

- Reject additions that leave a process with no connected dataswitches
- Reject additions that leave an arc without both a valid process and a valid dataswitch
- Reject additions that leave a switch with no arcs connected to it
- Notify the user if any change is not covered by an existing decision

Although this mechanism is implemented within VS, it is easily modifiable based on the needs of the end user. This could be done by editing the decision tree portion of VS and recompiling VS. Also, a more powerful and flexible mechanism could be developed on top of VS using a policy enforcing system such as FOREST [Garlan, 1990].

Once the automated merge, is completed, the user then has the option of checking the modifications in, modifying the changes, or rejecting the changes, based on the informational messages that are returned. The user could also have another program monitoring the output to increase the amount of automation in this process.

In the current implementation, the *rev* option to the DFDMERGE command is a branch version number such as 1.2.1.3. VS uses the version number given to determine the version number of the trunk version from which the version was forked and checks the differences between the version given and the trunk version. The changes are then applied to the top of the trunk for the named file provided. This should be modifiable to provide more flexibility by using the same techniques of `co -j` which lets the user specify all three versions.

The generation of differences is done by the `rcsdiff` command and stored in a temporary file for parsing by the automated decision tree mechanism in VS. The ac-

cepted changes are left on the top of trunk version of the working file. The changes that are rejected are placed in an ASCII file using a format compatible with the UNIX `patch` program.

The choice of the particular reject decisions in VS show its ability to recognize error conditions specific to a software design method, in this case Large-Grain Data Flow 2. The accepting comments decision shows the ability of VS to differentiate the comments from other types of data. Finally, the "notify" decision shows the ability of VS to recognize changes that need to be handled by a human.

## 3.5 The LGDF2 File

The LGDF2 file, i.e., a controlled module, is an ASCII file. It contains the representation of a LGDF2 diagram which was created either by a design aid, such as a diagram editor, or by a human designer. The text in the file contains the LGDF2 representation data and the RCS identification markers. VS requires the markers `$Header:$`, `$Revision:$`, and `$Source:$` for internal use. The use of the additional markers is recommended for design maintenance.

The choice of an ASCII file was made for two reasons:

1) An ASCII file could be edited by a human designer and shared among a variety of different computers and design tools.

2) the ASCII file could be easily stored and manipulated by RCS.

There are three classes of LGDF2 data representation tokens in the file: keywords, comments and strings. The keywords are shown in Table 3-4.

| Token | Meaning/Usage |
|---|---|
| CCCCCCCC | End of the LGDF2 diagram data |
| CENTRXY- *argument* | Center XY position |
| DNUMBER- *argument* | Dataswitch number |
| ENDXY--- *argument* | End XY position |
| HEIGHT-- *argument* | Height of a rectangle |
| LINE---- | Arc type specifier |
| LPNUMBR- *argument* | PNUMBER for a connected arc |
| LSNUMBR- *argument* | SNUMBER for a connected switch |
| NAMESTR- *argument* | Name string |
| NODE---- argument | Diagram node type specifier |
| PERMISS- *argument* | The read/write permission |
| PNUMBER- *argument* | Process number |
| PROCESS- | Process type specifier |
| RADIUS-- *argument* | Radius of the process |
| SNUMBER- *argument* | Dataswitch number |
| STARTXY- *argument* | Start XY position |
| SWITCH-- | Dataswitch type specifier |
| TITLE--- *argument* | Title of the diagram |
| ULEFTXY- *argument* | Upper left XY position |
| WIDTH--- *argument* | Width of a rectangle |
| ZZZZZZZZ | End of the LGDF2 diagram file |

**Table 3-4. The LGDF2 Diagram Keywords.**

The keywords are expected to be at the beginning of a line of text and some can be followed by a string. The keywords are all eight characters wide. A string is separated from a keyword by one or more spaces and terminated with a carriage return (ASCII CR). Comments start with a # on the beginning of a line and are terminated with a carriage return (ASCII CR).

The LGDF2 data is separated from the RCS markers by the keyword `CCCCCCCC`. The keyword `ZZZZZZZZ` indicates the end of both the LGDF2 and RCS markers. The keywords `NODE----`, `LINE----`, `PROCESS-` and `SWITCH--` are a kind of type specifier. Each type specifier contains a series of keywords, strings and comments. An example LGDF2 file is shown in Figure 3-2. The format for each type specifier is presented in the user's manual in Appendix C.

This implementation of the LGDF2 diagram as an ASCII file allowed VS to be developed without the need for access to a design editor for creating LGDF2 diagram files. It also provided VS the advantage of being able to use the RCS tools and the UNIX tool `patch`. Finally, parsing of the file was simplified by the use of the fixed format for the LGDF2 data.

```
NODE----
TITLE--- Test Program
ULEFTXY- 100 100
#
PROCESS-
CENTRXY- 180 180
RADIUS-- 50
NAMESTR- Process # 1
ULEFTXY- 130 170
PNUMBER- 1.1
ULEFTXY- 190 190
#
LINE----
DNUMBER- 1.1
STARTXY- 230 180
LPNUMBR- 1.1
ULEFTXY- 240 190
PERMISS- AL
ENDXY--- 250 180
LSNUMBR- 1.1
PERMISS- AR
#
SWITCH--
ULEFTXY- 250 150
WIDTH--- 100
HEIGHT-- 200
TITLE--- Switch # 1
ULEFTXY- 260 170
SNUMBER- 1.1
ULEFTXY- 290 390
CCCCCCCC
$Author: adamsc $
$Date: 90/11/04 16:08:01 $
$Header: dfdnew,v 1.1 90/11/04 16:08:01 adamsc Exp $
$Locker:  $
$Log: dfdnew,v $
$Source: /usr/test/OGC/THESIS/src/PROJECT.dfdone/RCS/dfdnew,v $
$State: Exp $
$RCSfile:        $
$SymbolicNames:  $
$CheckoutLog:    $
$AccessList:     $
ZZZZZZZZ
```

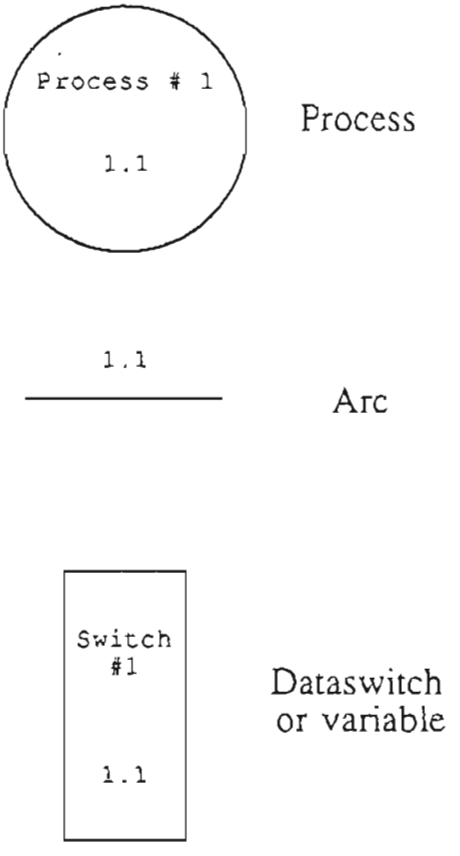


**Figure 3-2. An Example LGDF2 File.**

# 4. Example VS Usage Scenarios

The current VS prototype can be used as a general-purpose tool for version management of shared, network-based software files. It has features which provide both domain-independent version control built on top of RCS, and specific support for Large-Grain Data Flow 2 diagrams. Experience with VS in this design development environment has led to some general recommendations for its application to larger designs and other environments. The primary motivation for studying the use of this version management tool is to determine the applicability of this class of tool to the larger problem of version management of software design development. The use of LGDF2 is particularly suited for this analysis since it is relatively simple, but contains elements common to many other software design techniques.

In this example, the particular elements of the design are made up from the elements of the VS program itself. The choice of this example provided a means for examining the interplay between the version management system and the design process when using a top-down structured design methodology. In order to demonstrate the effects of multiple designers using this particular tool concurrently, several of the examples use fictitious users "*test*" and "*adamsc*". This allowed the testing and evaluation of VS in more depth even though there was only one designer for this implementation of VS.

## 4.1 The Structure of VS

VS began its life in the way advocated by most top-down structured methodologies as a single process with no hierarchy. This single process LGDF2 diagram is shown in Figure 4-1. Then as the designer's knowledge of the requirements increased and the development progressed, arcs, dataswitches and processes were added and modified. The LGDF2 diagram for the current version of the VS prototype is shown in Figure 4-2.

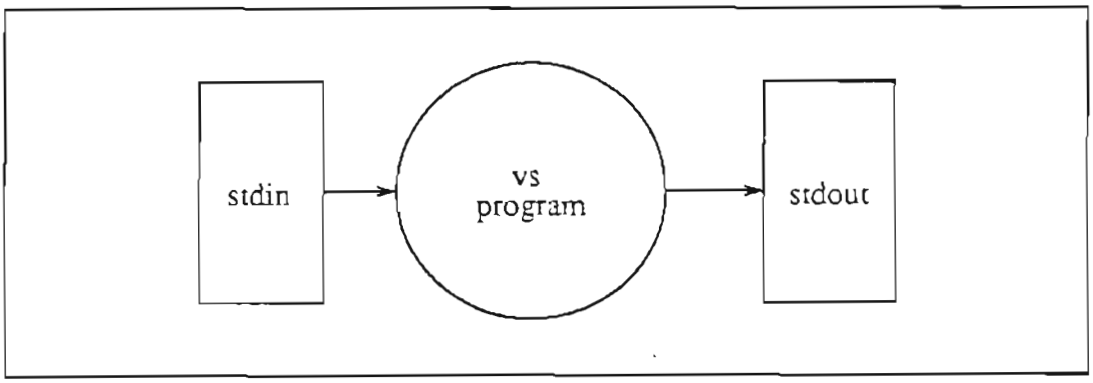


**Figure 4-1. The Single Process LGDF2 Diagram.**

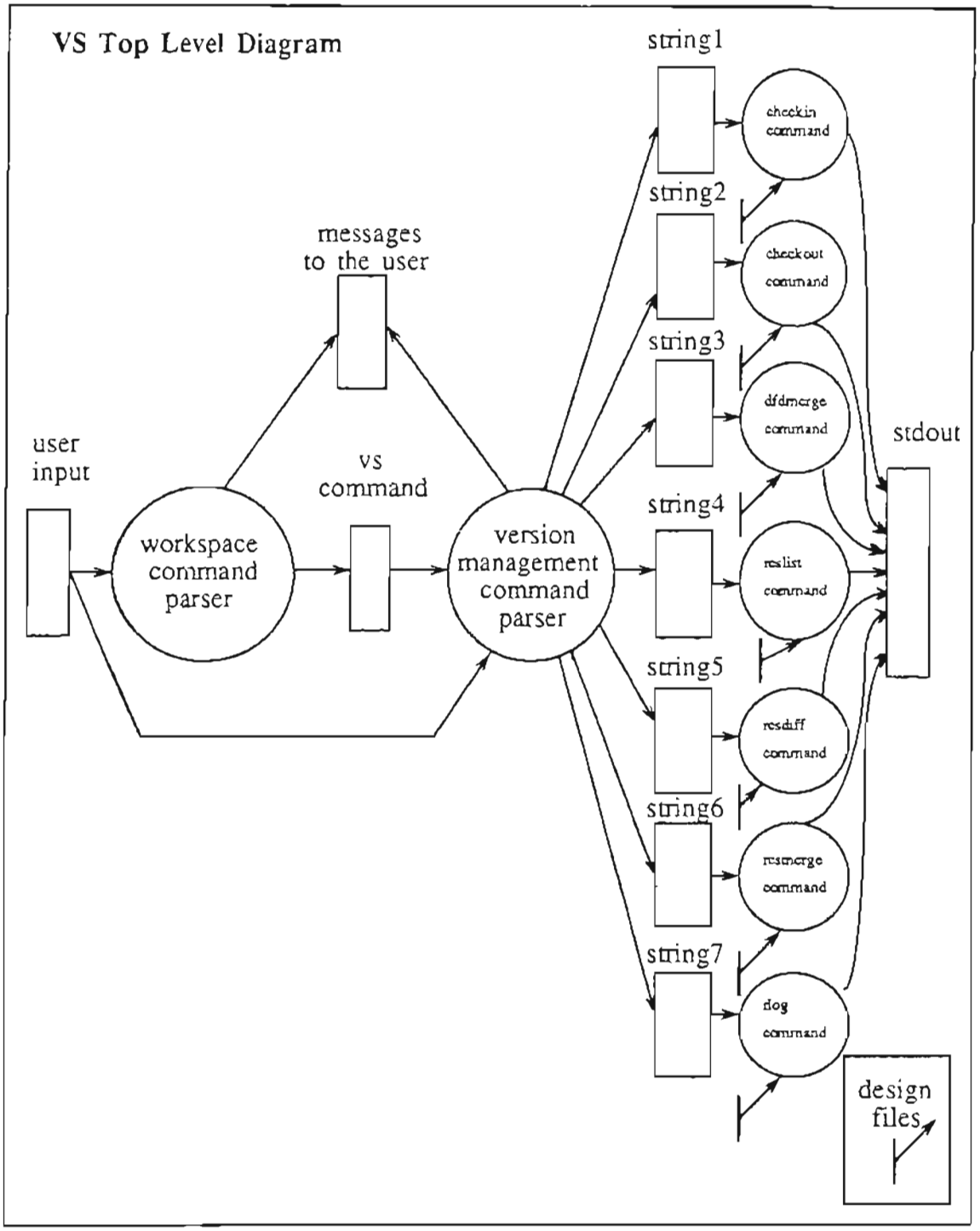


Figure 4-2. The Top Level LGDF2 Diagram for VS.

The example interactions presented in this chapter use the following format:

- The column on the left shows the input and output from VS.
- The column on the right provides comments about the interaction.

In the left-hand column, all output from VS is shown in bold-face, all output from the UNIX operating system is shown in italics, and user or program input is shown in normal type.

## 4.2 Sample Check Out Session

A typical VS check out transaction would take place as shown below.

| | |
|---|---|
| *script started on Sun Oct 14 13: 18 : 56 1990* | script is a program in UNIX to record everything printed on the terminal. |
| % whoami | This is a program to print the current user name. |
| *adamsc* | |
| % vs | This is the name of the program. |
| **VMESSAGE vs: started** | This is a informational message indicating VS has started successfully. |
| **VPROMPT>** PROJECT? | The VPROMPT indicates VS awaiting input. The user has asked VS to list the available workspaces. The response is two informational messages that workspaces dfdone and dfdtwo are available. |
| **VMESSAGE vs: dfdone** | |
| **VMESSAGE vs: dfdtwo** | |
| **ZZZZZZZZ** | The ZZZZZZZZ indicates the end |

| | |
|---|---|
| | of the informational messages. |
| **VPROMPT>** PROJECT: dfdone | The user has asked VS to connect to the project workspace named dfdone in the directory directly below the current one. Note that the choice of the name "dfdone" is arbitrary and with VS's access to all the NFS networking capability, the user could have selected any workspace in the currently mounted directory space. |
| **VMESSAGE vs: PROJECT.dfdone** | The VMESSAGE contains an informational message indicating the workspace to which VS has connected. |
| **ZZZZZZZZ** | |
| **VPROMPT>** CHECKOUT dfd1 | The user has asked VS to check out the LGDF2 diagram dfd1. |
| **VMESSAGE co: RCS/dfd1,v --> dfd1** | The VS command echoes the RCS co command generated output to make the transition to VS easier for users familiar with RCS. |
| **VMESSAGE co: Revision 1.9** | Like RCS, VS defaults to a check out on the top-of-trunk . It is likely that a higher level program or the user may want to change this policy and if |

| | |
|---|---|
| | needed, the user can employ any of the RCS capabilities to implement different policy. |
| **VMESSAGE co: done** | The RCS co command output indicating that the co was successful. |
| **ZZZZZZZZ** | The end of the CHECKOUT messages. |
| **VPROMPT>** GAMEOVER | The user has asked to exit from VS and remain in the current workspace. |

In the above sample check out, the user was *adamsc*. For purposes of this test, the second user *test* issued the same command sequence. A portion of the resulting dfd1-vlog file is shown in Figure 4-3. Note that the check outs for both *adamsc* and

```
co Sun Feb 25 11:11:41 1990                    adamsc 1.6
ci Sun Mar  4 12:22:19 1990                    adamsc 1.7
co Fri Oct  5 16:40:53 1990                      test 1.7
ci Fri Oct  5 16:47:34 1990                      test 1.8
co Fri Oct  5 16:48:17 1990                      test 1.8
co Sun Oct 14 13:15:09 1990                      test 1.8
ci Sun Oct 14 13:16:24 1990                      test 1.9
ci Sun Oct 14 13:20:21 1990                    adamsc 1.8.1.1
co Sun Oct 28 13:10:45 1990                    adamsc 1.9
co Sun Oct 28 13:22:05 1990                      test 1.9
```

**Figure 4-3. A Portion of the dfd1-vlog File.**

*test* were recorded in the `-vlog` file for version 1.9 in case a project wants to track who is using or has used a particular version. In RCS no record of these transactions is kept and thus there is no means to identify who is using any particular version of the design.

## 4.3 Sample Check In Session with Notification

A typical VS check in transaction with multiple persons having the top-of-trunk checked out, but no intervening check in, is shown below.

| | |
|---|---|
| *script started on Mon Oct 15 12: 18 : 56 1990* | |
| % whoami | |
| *adamsc* | |
| % vs | |
| **VMESSAGE vs: started** | |
| **VPROMPT>** PROJECT: dfdone | |
| **VMESSAGE vs: PROJECT.dfdone** | |
| **ZZZZZZZZ** | |
| **VPROMPT>** CHECKIN- dfd1 | The user has asked VS to check in the LGDF2 diagram dfd1. |
| **VMESSAGE ci: RCS/dfd1,v <-- dfd1** | VS echoes the outputs generated by RCS ci. |
| **VMESSAGE ci: New revision: 1.10; previous revision: 1.9** | Like RCS, VS defaults to a check in on the top-of-trunk . If a higher level program or a user wants to change this policy, all the RCS capabilities of branching are available within VS. |
| **VMESSAGE ci: Notify user test Has check out of version 1.9** | The VS command has read the -vlog file and notes that either another |

| | |
|---|---|
| | program or the user should notify the user *test* that this modification has taken place. |
| ZZZZZZZZ | The final CHECKIN- message. |

VPROMPT> GAMEOVER

Because the VS check out transaction for *test* has been recorded in the `-vlog` file for the dfd1 design, the VS `CHECKIN-` command can notify either the user, or another program that others may be interested in the modification that has taken place. Under RCS, no record was kept of the check outs and thus only a much more limited mechanism for change notification could be implemented to enforce a change notification policy. As noted in Chapter 1, the dissemination of information about modifications to design is one of the primary concerns during the design phase of any project. It has also been shown that some projects need to establish formal communication channels, i.e. project leaders are asked to track changes as they happen [Lord, 1988]. With this check in/check out mechanism it would be possible to automate that communication process.

The graphical representation of version 1.9 of the LGDF2 diagram for the above transaction is presented in Figure 4-4. Note that version 1.9 could be considered a template since the form matched a portion of the VS design, even though the names bore no resemblance to VS names.

The version 1.10 shows the first transformation from a template to the actual

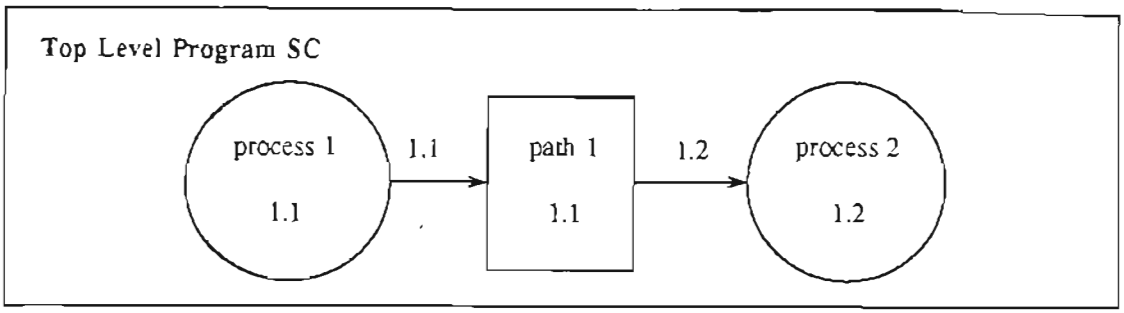


Figure 4-4. VS LGDF2 Diagram Version 1.9.

LGDF2 representation for a portion of the VS program. Figure 4-5 shows the result of the changes checked in by the user *adamsc*. The LGDF2 file for Version 1.10 of VS is presented in Appendix A.

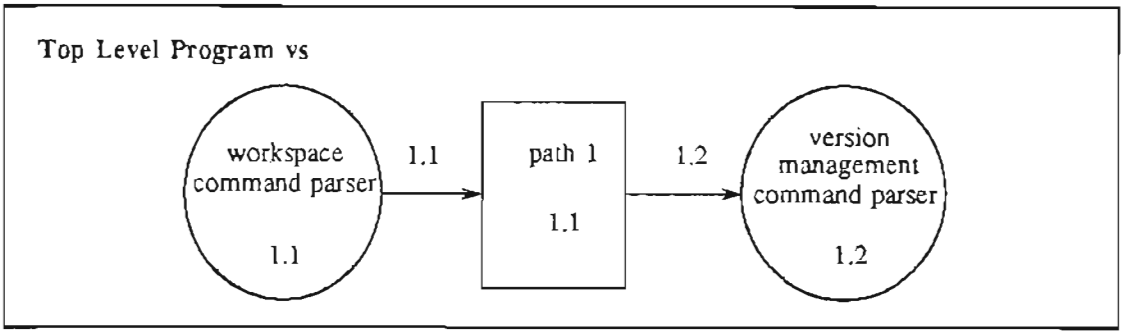


Figure 4-5. VS LGDF2 Diagram Version 1.10.

## 4.4 Sample Check In Session with Intervening Check In

A typical VS check in transaction with someone having updated the top-of-trunk is shown below.

---

*script started on Tues Oct 16 13: 18 : 56*

*1990*

% whoami

*test*

% vs

```
VMESSAGE vs: started
VPROMPT> PROJECT: dfdone
VMESSAGE vs: PROJECT.dfdone
ZZZZZZZZ
VPROMPT> CHECKIN- dfd1
VMESSAGE ci: RCS/dfd1,v <-- dfd1
VMESSAGE ci: New revision:1.9.1.1
previous revision: 1.9
```

Unlike RCS, VS defaults to a check in on an unused branch when the top of trunk is higher than the revision the user checked out. In this case, the user had checked out version 1.9, and version 1.10 already existed. The user or higher level program was notified that the branch has been created. It is then up to the user or higher level program to decide how to respond to this automatic branching.

```
ZZZZZZZZ
VPROMPT> GAMEOVER
```

---

In the current system, the automatic branching mechanism forces the user to run the DFDMERGE command as a separate transaction. In more sophisticated systems, the controlling program could reduce this multiple step transaction to a single operation by combining both CHECKIN- and DFDMERGE.

The LGDF2 diagram after the check in is presented in Figure 4-6. Note that some

of the changes made by user *adamsc* are not in version 1.9.1.1 since both parties were working in parallel. It should also be noted that the changes in version 1.9.1.1 may not be desired by the project until they are "error free" or "blessed by a design review."

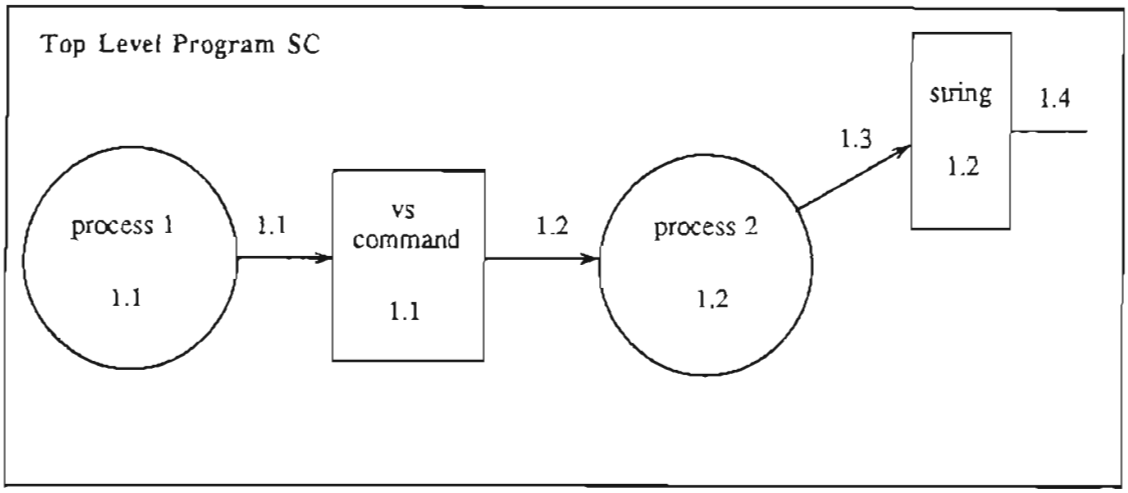


**Figure 4-6. VS LGDF2 Diagram Version 1.9.1.1.**

In this example, the change of the title of the SWITCH 1.1 from "path 1" to "vs command" would be a helpful change for the entire design team. On the other hand, if the project had established a policy that the top-of-trunk should only contain lines that either establish inputs or outputs from off the graph, or are connected to both a process and a switch. The addition of the lines 1.3 and 1.4 and the switch 1.2 would be a violation of such a policy. Thus, this part of the change would not be useful to the entire design team until the bubble joining line 1.4 was developed or the arc 1.4 was extended to an output that is off the graph.

Also, since the users *adamsc* and *test* were working in parallel there is a productivity advantage to be gained if the work done be both users can supplement each other without changes done by one user conflicting with those done by the other user. In

this example, the users could have agreed in advance that the names of the datapaths connected to process 1.2 would be established by user *test* and the names of the processes would be established by user *adamsc*. In this case, user *test* is going to leave the naming of the process 1.2 alone or the version manager is going to have to be supplied with a mechanism to accept changes from user *adamsc* and not from user *test*. Otherwise, changes made by user *test* are going to be in conflict with changes done by user *adamsc*. These conflicts are not handled by this implementation of VS.

## 4.5 Sample DFDMERGE Session with Error Policy Checking

A typical VS DFDMERGE transaction with the mechanism that checks for additions which result in unconnected lines, switches and processes is shown below.

---

*script started on Tues Oct 16 14: 18 : 57 1990*

% whoami

*test*

% vs

**VMESSAGE vs: started**

**VPROMPT>** PROJECT: dfdone

**VMESSAGE vs: PROJECT.dfdone**

**ZZZZZZZZ**

**VPROMPT>** DFDMERGE 1.9.1.1 dfd1 — The user has asked VS to merge the changes from version 1.9 to version 1.9.1.1 in the LGDF2 diagram dfd1 onto the top-of-trunk.

**VMESSAGE rcsmerge: RuleAccept Change** — For informational purposes VS reports its actions during the merge. In this

| | |
|---|---|
| | case this section of the change was accepted because a rule for this change was found in the decision tree. |
| VMESSAGE rcsmerge: RuleDelete Change | In this case, this section of the change was rejected because a rule in the decision tree indicated this section of the change did not meet project policy. |
| Hmm... Looks like a context diff to me<br>The text leading up to this was:<br>...<br>Patching file dfd1 using Plan A<br>Hunk #1 succeeded at 28.<br>done<br>ZZZZZZZZ | This informational message comes from the patch program. Since VS uses the program patch to reverse the affect of any rejected changes there will be a series of such messages up until the output token ZZZZZZZZ. |
| VPROMPT> GAMEOVER | |

VS implemented this merge mechanism in a five step process. At step one, the RCS co -j command was used to create a new version of the working file. Then, VS removed any notices of overlapping changes. The program then created a context diff between the current working file in this case dfd1 and the top-of-trunk which in this case was version 1.10. The changes were checked using a decision tree mechanism so that the project policy would be followed. Finally, any changes to be rejected are removed by applying a reverse patch. In this case the addition of the lines 1.3 and 1.4 and switch 1.2 is removed. The rejected portions of the patch were placed in the

file `diffs.notaccepted`. Figure 4-7 shows the LGDF2 diagram after the acceptance of the changes that met the project policy. Figure 4-8 shows the `diffs.notaccepted` file for this modification.

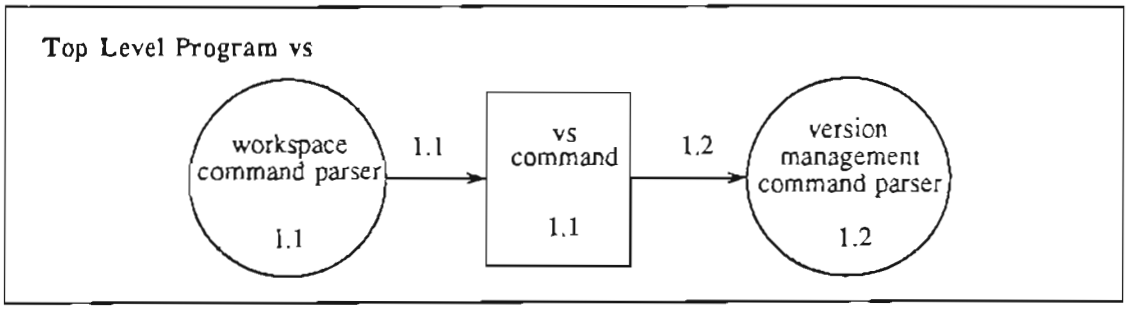


Figure 4-7. VS LGDF2 Diagram Version 1.11.

Now that this transaction has been completed the user has the option of checking the changes in using the `CHECKIN-` command, making more modifications prior to doing a check in or deciding these changes are unnecessary, and leave the project branch unchanged.

```
*** /tmp/,RCSt1017580  Mon Dec  3 11:30:34 1990
--- dfdnew  Mon Dec  3 11:30:32 1990
***************
*** 50,55
  # Put my new change on top of trunk
  # I don't like this comment line
  # So I added another comment line
  CCCCCCCC
  $Author: adamsc $
  $Date: 90/11/04 16:21:49 $

--- 50,86 -----
  # Put my new change on top of trunk
  # I don't like this comment line
  # So I added another comment line
+ #
+ # I have added another two lines and a switch
+ LINE----
+ DNUMBER- 1.2
+ STARTXY- 205 180
+ LPNUMBR- 1.3
+ ULEFTXY- 280 150
+ PERMISS- AL
+ ENDXY--- 300 120
+ LSNUMBR- 1.2
+ PERMISS- AR
+ #
+ SWITCH--
+ ULEFTXY- 300 90
+ WIDTH--- 100
+ HEIGHT-- 200
+ TITLE--- string
+ ULEFTXY- 305 100
+ SNUMBER- 1.2
+ ULEFTXY- 310 120
+ #
+ LINE----
+ DNUMBER- 1.3
```

**Figure 4-8.** The diffs.notaccepted File for Version 1.11.

```
+ STARTXY- 325 100
+ LPNUMBR- 1.4
+ ULEFTXY- 310 90
+ PERMISS- AL
+ ENDXY--- 350 100
+ LSNUMBR- 1.2
+ PERMISS- AR
+ #
  CCCCCCCC
  $Author: adamsc $
  $Date: 90/11/04 16:08:01 $
***************
*** 92,98
  # Revision 1.1  89/10/22  11:47:30  adamsc
  # Initial revision
  #
! $Revision: 1.11 $
  $Source: /usr/test/OGC/THESIS/PROJECT.dfdone/RCS/dfdnew,v $
  $State: Exp $
  $RCSfile:        $

--- 129,135 -----
  # Revision 1.1  89/10/22  11:47:30  adamsc
  # Initial revision
  #
! $Revision: 1.9.1.1 $
  $Source:/usr/test/OGC/THESIS/PROJECT.dfdone/RCS/dfdnew,v $
  $State: Exp $
  $RCSfile:        $
```

Figure 4-8. The diffs.notaccepted File for Version 1.11 (continued).

# 5. Conclusion

The management of large, complex software development projects can be a very difficult task. Problems of communication and control increase dramatically as project size increases. It is generally accepted that productivity during source code development and maintenance can be enhanced by the use of version management techniques and tools. It is thus likely that productivity during the design phase of large complex software design development could be enhanced through the use of improved version management tools and techniques.

## 5.1 Assessment of VS for Version Management

In the example application of VS, a version management system for Large-Grain Data Flow 2 diagrams, the DFDMERGE command was used to check design changes against the project policy. DFDMERGE was able to accept or reject changes based on the policy established. This reduced the need for hand checking of design changes in the example scenario.

This automated checking of designs could be extended in several ways. A system could be built that would allow the user to define the policy and then the tool would be used to enforce that policy. Also, a design checker, much like the Problem Statement Analyzer, could be built to assist the user in identifying errors in LGDF2 designs.

In the example check in scenario, the automated branching mechanism was used to enforce a first-come, first-served check in policy. This check in mechanism was able to maintain the consistency of design changes without requiring the use of long term locks.

This check in mechanism could be further extended by combining the branching and

merging steps. In a more integrated system, the functionality of CHECKIN- and DFDMERGE could be combined into a single command. This combined command could also be improved by providing it with a mechanism for automated error checking.

The example scenarios also showed VS's ability to record check outs and provide a simple change notification mechanism. Because the record of check outs existed, VS was able to notify users of overlapping check outs. The user was then expected to notify the other user(s) of modifications that may have affected them.

This storage mechanism would allow a very sophisticated update mechanism be built using a configuration management tool. For instance, such a tool could monitor the output from VS and then automatically check out the new version of the module after the change is noted. The storage mechanism could be further extended to provide a means to distinguish between users who are reading the version, and those who are modifying the module.

It is contended that as the size of design project increases, the amount of communication required increases. Thus, tools or techniques that reduce this communication should increase productivity. Tools, like VS, should help reduce the amount of communication needed about changes, because only the parties affected by a change are notified of the change. This ability to better target change notification messages and subsequent updates should help improve the efficiency of large software design teams.

Also, as the number of engineers on a design project increases, the likelihood of errors introduced into the design increases, since there are more interfaces that affect more engineers. With VS, it is possible to reduce these errors. This automated checking should lead to increased productivity by reducing the amount of costly rework done in the later stages of a project. Also, unneeded processes and variables

are identified and discarded early, so the amount of code developed and referenced but not used at runtime is reduced.

In addition, when larger and more complex design projects use LGDF2, the competition for the locked versions of designs increases. Thus, it is important to reduce the amount of competition for controlled resources to further increase productivity. With VS, this competition can be reduced through the elimination of the use of long term locking. Thus, one user is not waiting for another to release a version lock. Even if branches are used, with the current implementation of RCS the conflict is not eliminated, it is merely postponed because the check in onto the project's branch can not take place until the version lock is released. With VS, locks are not used, and in most cases, the merge can be done at the time that the branch is created. This should help reduce the delay between the time that the branch is created and the time the merge can take place. Since this delay could cause an arbitrary context switch which may be at an inopportune time, the use of a first-come, first-served check in policy could be noted by a reduction in the number of errors made during merges.

VS is an example of a version management tool for Large-Grain Data Flow 2 diagrams. This tool, with appropriate modifications, would also be useful for projects that use similar structured design and analysis techniques. A version management tool of this type could be developed for many different types of design development techniques that employ a formal semantics and syntax. For instance, state transition diagrams, conceptual object oriented designs and SCOOP-3, an Ada-based graphical design method [Cherry, 1990], could be handled in a similar fashion. Although the detailed implementation of VS was limited to LGDF2, the issues of checking for changes that do not meet project policy, communication of changes to the affected parties in

a timely manner and the need for parallel development are universal within the software engineering field.

VS was used on a very limited test. For a full verification of its effectiveness in a large, distributed, development environment, a larger number of LGDF2 diagrams and a larger number of projects would have to be analyzed. Although VS has shown promise for the development of a fully integrated version management tool for LGDF2 diagrams it was not tested in conjunction with a higher level configuration management tool, such as DSEE or Cosmos. A complete evaluation of VS would require its use within such a tool.

## 5.2 Assessment of Need for Version Management

Version management has been an important part of software development for many years. From the inception of SCCS through the recent developments in configuration management, version management has proved beneficial in increasing software engineers productivity. These productivity increases result from the ability of the tools, such as RCS, to identify changes, isolate problematic changes, isolate experimental versions from released versions and reduce the amount of storage space needed for multiple versions of controlled files. Improvements in version management have helped improve its ability to handle multiple person, multiple project software development.

Systems, such as RCS and SCCS, have gained widespread acceptance within the software engineering community, particularly in the area of source code development. On the other hand, little work has been done in studying how version management can improve the productivity of software design teams. Few researchers have investigated what type of tools and techniques for version management are needed during

the design phase of the project life cycle.

There has been little research on what types of policies are used or needed for version management of software designs. Recent policy setting systems such as FOREST [Garlan, 1990], which make it easier to define project/system level policies, have not been used in the design phase of a project; and typically, place most of their emphasis on the code development phase of the project life-cycle. This research on the other hand, has shown that the existing domain-independent tools do not have the mechanisms to implement all the policies desired by most projects. More recent version management and configuration management tools, such as DSEE and NSE, have improved the state of the art when it comes to handling multiple person, multiple project source code tracking and communication. But they have not addressed the need for tools to enforce the error checking policies desired in large scale software development.

This research has shown that changes in graphical software design artifacts, such as Large-Grain Data Flow 2 diagrams can be tracked with tools such as RCS. Although, the software change deltas of SCCS and RCS will work fine for recording changes in the LGDF2 diagrams, these older tools force competition for long term locks on the controlled files which limits their effectiveness in large projects. Newer tools, such as Cosmos, have addressed the competition problems but have had difficulty gaining large-scale acceptance in part because of their lack of similarity to the tools currently being used. In addition, none of these other systems provide much help in enforcing policies about which changes should be added to the project's release branch and what changes should remain on an experimental branch. VS was able to enforce a project policy that no unconnected dataswitches should be merged

onto the top of trunk. In the future, software engineers may want the ability to mark parts of the design they are currently modifying "as work in progress" and have the system only accept changes that do not touch those parts of the design.

Finally, the version management tool must work effectively in the larger environment of both design and source code development. The current available version management tools provide little assistance in this integration. For instance, with most version management tools, it is easy to make changes via a check out, edit and check in process on a single module, but these tools do not have the ability to note how that change will affect other parts of the project such as design documentation. It is hoped, that sometime in the future that the change could be noted to a configuration management tool which in turn could note the other modules that also need to be modified. Eventually, automated tools could be provided to assist in making these other modifications.

## 5.3 Assessment of Need for Domain Specific Tools

As has been noted by Fred Brooks, improvements in productivity within software engineering are going to be a result of incremental developments in a variety of areas [Brooks, 1987]. Since close to three-fourths of all errors in software development can be traced back to errors made in the design phase of the project, tools must be developed which try to keep these errors from finding their way into the development phase of a project. Breakdowns during the design phase can spell disaster or at best result in much rework during the development phase. Thus, tools to help designers continue to be as important as tools to help source code developers.

Software design artifacts from systems like Large-Grain Data Flow 2, Data Flow Diagramming and SADT, which use graphical techniques for project design develop-

ment need version management. Software design systems, like LGDF2 and PSL, that have a formal semantics and syntax also provide the opportunity for development of design analysis tools. For instance, VS and PSA provide design checking in much the same way a program checker, such as lint does. Thus, with LGDF2 and PSL, it should be possible to reduce the number of design errors in the project's released design.

Since the syntax and semantics of the design languages LDGF2 and PSL are quite different, the further development of the error checking tools for these languages will have to be domain-specific. And since error checking tools in general are language specific, further developments in this area must be targeted at specific languages. On the other hand, tools such as VS, which combine both domain-independent and domain-specific features will be needed to continue to improve productivity within the field of software engineering. VS, a prototype version management system, should provide insight regarding future software tools development.

# REFERENCES

Babb, Robert G. II, "Data-Driven Implementation of Data Flow Diagrams", in the Proceedings of the 6th International Conference on Software Engineering, Tokyo, Japan, September, 1982, pp. 309-318.

Babb, Robert G. II, "Parallel Processing with Large-Grain Data Flow Techniques", IEEE Computer, Vol. 17, No. 7, July, 1984, pp. 55-61.

Babb, Robert G. II, "A Data Flow Approach to Unifying Software Specification, Design, and Implementation", in the Proceedings of the 3rd International Workshop on Software Specification and Design, London, England, August 1985, pp. 9-13.

Bailey, Robert W., *Human Error in Computer Systems*, Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1983.

Banerjee, Jay, et. al., "Data Model Issues for Object-Oriented Applications", ACM Transactions on Office Information Systems, Vol. 5, No. 3, January, 1987, pp. 3-26.

Basili, Victor R., "A Panel Session-User Experience with New Software Methods", in the Proceedings of the National Computer Conference, Anaheim, California, June, 1978, pp. 629-630.

Berliner, Brian, "CVS II: Parallelizing Software Development", Unpublished paper, available from author at Prisma, Inc., 5465 Mark Dablings Blvd. Colorado Springs, Colorado, 1990.

Black, Eric, "Software Configuration Management with an Object-Oriented Database", in the Proceedings of the Winter 1989 USENIX Conference, San Diego, California, January, 1989, pp. 257.272.

Boehm, Barry W., R. McClean, and D. Urfrig, "Some Experience with Automated Aids to the Design of Large Scale Reliable Software", in the Proceedings of the International Conference on Reliable Software, Los Angeles, California, April, 1975, pp. 105-113.

Boehm, Barry W., "Software Engineering", *IEEE Transactions on Computers*, , Vol. 25, No. 12, December, 1976, pp. 1226-1241.

Boehm, Barry W., *Software Engineering Economics*, Prentice-Hall, Inc. Englewood

Cliffs, New Jersey, 1981.

Boehm, Barry W., "Improving Software Productivity", IEEE Computer, Vol. 10, No. 9, September, 1987, pp. 43-57.

Boehm, Barry W. and Philip N. Papaccio, "Understanding and Controlling Software Costs", IEEE Transactions on Software Engineering, Vol. 14, No. 10, October 1988, pp. 1462-1477.

Brooks, Frederick P Jr., "No Silver Bullet: Essence and Accidents of Software Engineering", IEEE Computer, Vol. 10, No. 4, April, 1987, pp. 10-19.

Chang, Ellis E., David Gedye, and Randy H. Katz, "The Design and Implementation of a Version Server for Computer-Aided Design Data", Software-Practice and Experience, Vol. 19, No. 3, March, 1989, pp. 199-222.

Chou, Hong-Tai, and Won Kim, "Versions and Change Notification in an Object-Oriented Database System", in the Proceedings of the ACM/IEEE Design Automation Conference, Anaheim, California, June, 1989, pp. 275-281.

Cherry, George W., *Software Construction by Object-Oriented Pictures: Specifying Reactive and Interactive Systems*, Dorset House Publishing, New York, New York, 1990.

Combelic, Don, "Experience with SADT", in the Proceedings of the National Computer Conference, Anaheim, California, June, 1978, pp. 631-633.

Curtis, Bill, Herb Krasner, and Neil Iscoe, "A Field Study of the Software Design Process of Large Systems", Communications of the ACM, Vol. 31, No. 11, November, 1988, pp. 1268-1287.

DeMarco, Tom, *Structured Analysis and System Specification*, Yourdon, Inc., New York, New York, 1978.

DiNucci, David C. and Robert G. Babb II, "Practical Support for Parallel Programming", in the Proceedings of the Hawaii International Conference on System Science, Vol. II. Software Track, Kailua-Kona, Hawaii, January, 1988, pp. 109-118.

DiNucci, David C., "The LGDF2 Language and Preprocessor", available from author at Oregon Graduate Institute of Science and Technology, Beaverton, Oregon, October, 1990.

Garlan, David and Ilias, Ehsan J., "Low Cost, Adaptable Integration Policies Tool for Integrated Environments", to be published in the Proceedings of the Fourth Symposium on Software Development Environments, Irvine, California, December, 1990.

Hamilton, Margaret and Saydean Zeldin, "Higher Order Software: A Methodology for Defining Software", IEEE Transactions on Software Engineering, Vol. 2, No. 1, March, 1976, pp. 9-32.

Katz, Randy H., "Towards a Unified Framework for Version Modeling", Unpublished paper available from author at University of California, Berkeley, Computer Science Division, Electrical Engineering and Computer Science Department, Berkeley, California, 1990.

Leblang, David B. and Robert P. Chase, Jr., "Computer-aided Software Engineering in a Distributed Workstation Environment", in the Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Software Development Environments, January, 1984, pp. 104-112.

Leblang, David B. and Robert P. Chase, Jr., "Parallel Software Configuration Management in a Network Environment", IEEE Software, Vol. , No. , November, 1987, pp. 28-35.

Lord, Thomas, "Tools and Policies for the Hierarchical Management of Source Code Development", in the Proceedings of the Summer 1988 USENIX Conference, San Francisco, California, June, 1988, pp. 95-106.

Miller, Terrence C., "A Schema for Configuration Management", in the Proceedings of the 2nd International Workshop on Software Configuration Management, Princeton, New Jersey, October, 1989, pp. 26-29.

Nii, Penny H., "A Proposed Research Initiative in Knowledge-Based CASE Tools: Biting the Silver Bullet", Stanford University, Computer Science Department, Palo Alto, California, Technical Report KSL 89-75, 1990.

Page-Jones, Meilir, *The Practical Guide to Structured Systems Design*, Yourdon Press, New York, New York, 1980.

Reifer, Donald J., "Experience with PSL/PSA", in the Proceedings of the National Computer Conference, Anaheim, California, June, 1978, pp. 630-631.

Rochkind, Marc J., "The Source Code Control System", IEEE Transactions on Software Engineering, Vol. SE-1, No. 4, December, 1975, pp. 364-370.

Ross, Douglas T. and Kenneth E. Schoman, Jr., "Structured Analysis for Requirements Definition", IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, January, 1977, pp. 6-15.

Teichroew, Daniel and Ernest A. Hershey III, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing", IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, January 1977, pp.41-48.

Tichy, Walter F., "Design, Implementation, and Evaluation of a Revision Control System", in the Proceedings of the 6th International Conference on Software Engineering, Tokyo, Japan, September, 1982, pp. 58-67.

Walpole, Jonathan, Gordon S. Blair, J.R. Malik, and John R. Nichol, "A Unifying Model for Consistent Distributed Software Engineering Environments", in the Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Software Development Environments, Boston, Massachusetts, November, 1988, pp. 183-190.

Walpole, Jonathan, Angus Barber, Gordon S. Blair and John R. Nichol, "Software Development Environment Transactions: Their Implementation and Use in Cosmos", Unpublished paper, available from author at Oregon Graduate Institute of Science and Technology, Computer Science Department, Beaverton, Oregon, 1989.

Yourdon, Edward and Larry L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program Design*, Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1979.

Zelkowitz, Marvin V., Alan C. Shaw, and John D. Gannon, *Principles of Software Engineering and Design*, Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1979.

# APPENDIX A. Vs LGDF File Version 1.10

```
NODE----
TITLE--- Top Level Program vs
ULEFTXY- 120 120
#
PROCESS-
CENTRXY- 180 180
RADIUS--  50
NAMESTR- workspace command parser
ULEFTXY- 130 170
PNUMBER- 1.1
ULEFTXY- 190 190
#
LINE----
DNUMBER- 1.1
ULEFTXY- 240 190
STARTXY- 230 180
LPNUMBR- 1.1
PERMISS- AL
ENDXY--- 250 220
LSNUMBR- 1.1
PERMISS- AR
#
SWITCH--
ULEFTXY- 250 200
WIDTH--- 100
HEIGHT-- 200
TITLE--- path 1
ULEFTXY- 260 220
SNUMBER- 1.1
```

```
ULEFTXY- 290 390
#
LINE----
DNUMBER- 1.2
ULEFTXY- 270 190
STARTXY- 270 180
LPNUMBER- 1.2
PERMISS- AL
ENDXY--- 500 220
LSNUMBER- 1.1
PERMISS- AR
#
PROCESS-
CENTRXY- 500 180
RADIUS-- 50
NAMESTR- version manage command parser
ULEFTXY- 450 170
PNUMBER- 1.2
ULEFTXY- 460 190
#
# Put my new change on top of trunk
# I don't like this comment line
# So I added another comment line
CCCCCCCC
$Author: adamsc $
$Date: 90/10/29 07:33:55 $
$Header: dfd1,v 1.10 90/10/29 07:33:55 adamsc Exp $
$Locker:  $
$Log:dfd1,v $
# Revision 1.10  90/10/29  07:33:55  adamsc
# *** empty log message ***
```

```
#
# Revision 1.9  90/10/14  13:16:23  test
# *** empty log message ***
#
# Revision 1.8  90/10/05  16:47:33  test
# *** empty log message ***
#
# Revision 1.7  90/03/04  12:22:18  adamsc
# *** empty log message ***
#
# Revision 1.6  90/02/21  19:57:39  test
# *** empty log message ***
#
# Revision 1.5  90/02/21  19:43:05  adamsc
# *** empty log message ***
#
# Revision 1.4  90/01/11  08:31:55  adamsc
# test message using -m
#
# Revision 1.3  90/01/03  19:36:54  adamsc
# Changed the specification of the dfd files.
#
# Revision 1.2  89/11/07  08:25:48  adamsc
# Made a change from EEEEEEEE to ZZZZZZZZ to reflect new
spec.
#
# Revision 1.1  89/10/22  11:47:30  adamsc
# Initial revision
#
$Revision: 1.10 $
$Source:
```

```
/usr/test/OGC/THESIS/src/PROJECT.dfdone/RCS/dfd1,v $
  $State: Exp $
  $RCSfile:         $
  $SymbolicNames:   $
  $CheckoutLog:     $
  $AccessList:      $
  ZZZZZZZZ
```

# APPENDIX B. Vs LGDF File Version 1.9.1.1

```
NODE----
TITLE--- Top Level Program SC
ULEFTXY- 120 120
#
PROCESS-
CENTRXY- 180 180
RADIUS-- 50
NAMESTR- process 1
ULEFTXY- 130 170
PNUMBER- 1.1
ULEFTXY- 190 190
#
LINE----
DNUMBER- 1.1
STARTXY- 230 180
LPNUMBR- 1.1
ULEFTXY- 240 190
PERMISS- AL
ENDXY--- 250 180
LSNUMBR- 1.1
PERMISS- AR
#
SWITCH--
ULEFTXY- 250 150
WIDTH--- 100
HEIGHT-- 200
TITLE--- vs command
ULEFTXY- 260 170
SNUMBER- 1.1
```

```
ULEFTXY- 290 390
#
LINE----
DNUMBER- 1.2
STARTXY- 270 180
LPNUMBR- 1.2
ULEFTXY- 290 190
PERMISS- AL
ENDXY--- 500 180
LSNUMBR- 1.1
PERMISS- AR
#
PROCESS-
CENTRXY- 500 180
RADIUS-- 50
NAMESTR- process 2
ULEFTXY- 450 170
PNUMBER- 1.2
ULEFTXY- 460 190
#
# Put my new change on top of trunk
# I don't like this comment line
# So I added another comment line
#
# I have added another two lines and a switch
LINE----
DNUMBER- 1.2
STARTXY- 205 180
LPNUMBR- 1.3
ULEFTXY- 280 150
PERMISS- AL
```

```
ENDXY--- 300 120
LSNUMBR- 1.2
PERMISS- AR
#
SWITCH--
ULEFTXY- 300 90
WIDTH--- 100
HEIGHT-- 200
TITLE--- string
ULEFTXY- 305 100
SNUMBER- 1.2
ULEFTXY- 310 120
#
LINE----
DNUMBER- 1.3
STARTXY- 325 100
LPNUMBR- 1.4
ULEFTXY- 310 90
PERMISS- AL
ENDXY--- 350 100
LSNUMBR- 1.1
PERMISS- AR
#
CCCCCCCC
$Author: adamsc $
$Date: 90/11/04 16:08:01 $
$Header: dfd1,v 1.9.1.1 90/11/04 16:08:01 adamsc Exp $
$Locker:  $
$Log:dfd1,v $
# Revision 1.9.1.1  90/11/04  16:08:01  adamsc
# *** empty log message ***
```

```
#
# Revision 1.9.1.1  90/11/04  09:48:05  test
# *** empty log message ***
#
# Revision 1.9  90/10/14  13:16:23  test
# *** empty log message ***
#
# Revision 1.8  90/10/05  16:47:33  test
# *** empty log message ***
#
# Revision 1.7  90/03/04  12:22:18  adamsc
# *** empty log message ***
#
# Revision 1.6  90/02/21  19:57:39  test
# *** empty log message ***
#
# Revision 1.5  90/02/21  19:43:05  adamsc
# *** empty log message ***
#
# Revision 1.4  90/01/11  08:31:55  adamsc
# test message using -m
#
# Revision 1.3  90/01/03  19:36:54  adamsc
# Changed the specification of the dfd files.
#
# Revision 1.2  89/11/07  08:25:48  adamsc
# Made a change from EEEEEEEE to ZZZZZZZZ to reflect new
spec.
#
# Revision 1.1  89/10/22  11:47:30  adamsc
# Initial revision
```

```
  #
  $Revision: 1.9.1.1 $
  $Source:
/usr/test/OGC/THESIS/src/PROJECT.dfdone/RCS/dfd1,v $
  $State: Exp $
  $RCSfile:          $
  $SymbolicNames:    $
  $CheckoutLog:      $
  $AccessList:       $
  ZZZZZZZZ
```

# APPENDIX C. Users Manual for VS

## VS User's Manual

*Charles Adams*

Oregon Graduate Institute of Science and Technology

31 December 1990

# VS User's Manual

*CONTENTS*

# VS User's Manual

*Charles Adams*

Oregon Graduate Institute of Science and Technology

## 1. HOW TO USE THIS MANUAL

Readers wanting to get an overview of *vs*'s features should consult Section 2.

Readers wanting to invoke the program should consult Section 3.

Readers wanting to execute any of the commands available within *vs* should consult Section 3.

Readers who have just received the source code for *vs* and wish to compile it or modify it should consult Section 4.

Section 5 presents a quick lookup reference guide to the commands and parameters of *vs*.

Rather than repeat all that is written in the documentation on the Revision Control System by Walter F. Tichy, I will suggest starting by reading the following documents:
*Design, Implementation, and Evaluation of a Revision Control System* by Walter F. Tichy, in *Proceedings for the 6th International Conference on Software Engineering*, IEEE, Tokyo, Japan, September, 1982.
Unix Programmer's Manual documentation *RCSINTRO* and the man pages for each element of rcs used.

## 2. OVERVIEW

Vs is a Version Management System for Large-Grain Data Flow Diagrams. It provides the command interface to the UNIX† Revision Control System along with the added functionality never before available with RCS. The check-in and check-out commands have additional functionality for handling optimistic version control on any file that can be handled by RCS. The branch merge command gives the user the additional capability of automatic branch merging when processing Large-Grain Data Flow Diagram 2 (LGDF2) files. The automated checking of LGDF2 files undertaken during merging is an example of the project policy that can be implemented with a tool like vs.

### 2.1 Invoking VS

### 2.2 The VS Commands

Vs is divided into two functional areas. First, the workspace commands allow the user to set and query the project workspaces available to vs. Second, the version management commands allow the user to manipulate controlled modules.

#### 2.2.1 Workspace commands

The workspace commands provide a limited namespace control via vs. The commands available are "`PROJECT:`", "`PROJECT?`" and "`GAMEOVER`". These commands have a fixed length of eight characters and an ending space character plus the workspace name in the case of the "`PROJECT:` " command.

#### 2.2.2 Version management commands

The version management commands consist of the complete set of RCS commands with the exception of the `rcs` command, plus the added capabilities of vs. The commands "`RCSLIST-`", and "`DFDMERGE`" are new with vs. The commands "`CHECKIN-`", and "`CHECKOUT`" are modified versions of `ci` and `co` from RCS. The commands "`RCSDIFF-`", "`RCSMERGE`" and "`SENDRLOG`" are unmodified except for the user interface from RCS. Finally the commands "`GAMEOVER`" and "`ZZZZZZZZ`" are commands which allow the user to end the program or connect to the previous directory. Again, like the workspace commands each command is eight characters in width plus a space character and those commands that take arguments, the arguments are placed after the space and before the carriage return.

† UNIX is a Trademark of Bell Laboratories.

## 3. INVOKING VS

The *vs* command is invoked by:

**vs** *options*

where *options* is none, or any combination of the two option parameters. The options can appear in any order.

### 3.1 VS Command Line Options

| OPTION | EFFECT |
|---|---|
| -echo | Display on stdout the commands input from stdin. |
| -debug *value* | Specify the level of debug messages to be printed.<br>0 – all debug messages are printed.<br>6 – (the default) few debug messages are printed. |

On successful invocation of *vs* it will display the prompt:

```
VPROMPT>
```

# 4. VS COMMANDS AND THEIR PARAMETERS

*Vs* has two levels of commands. The first level commands, called *Workspace Commands*, allow the user to set and query workspaces. The second level commands, called *Version Management Commands*, allow the user to invoke any of the RCS commands. At both levels the vs command interpreter expects its input to come from *stdin* and sends its messages to *stdout*.

## 4.1 Workspace Commands

| COMMAND | EFFECT |
|---|---|
| PROJECT?□ | Display the available workspaces |
| PROJECT:□*workspace* | Set the current working area to *workspace* |
| GAMEOVER□ | Exit gracefully from *vs* |

### 4.1.1 PROJECT? Command

The *PROJECT?* command displays the currently available workspaces. Each workspace is a directory directly below the current directory level. The names returned are proceeded by the *vs* prompt **VMESSAGE vs:**. The name of the workspace varies in length and is followed by the carriage return (0x15) character.

### 4.1.2 PROJECT: Command

The *PROJECT:* command sets the current workspace. The workspace is any valid workspace name.

If the workspace named exists and is directly below the current directory, *vs* will send back a prompt:

**VMESSAGE vs: PROJECT.***workspace*

After sending all the debug messages specified, *vs* will change the current directory to the workspace named and send the prompt for the *Version Management Commands*.

If the workspace named does not exist, *vs* will send back the following informational message.

```
VMESSAGE vs: Working space "workspace" does not
exist.
VMESSAGE vs: Should I create this workspace [Y/N]?
```

---

The □ is really a space (0x20) character and is required by the protocol.

*Vs* will then await input from the user. If the user inputs *Y* then *vs* will go ahead and create the named workspace directly below the current level. *Vs* will then change the current directory to the workspace named. If the user inputs anything other than *Y* as the first character of the input then *vs* will stay in the current directory and stay at the *Workspace Command* level.

In essence, no action will be taken other than to reissue the input prompt:

```
VPROMPT>
```

### 4.1.3 GAMEOVER Command

The *GAMEOVER* command is used to exit from *vs*. Once the command is issued successfully the program will cleanup after itself, exit from the program and return the UNIX prompt.

### 4.1.4 Workspace Command Error

When a command is not understood, *vs* will display the error prompt:

```
VSERROR- vs: Unable to parse command
```

and then indicate allowed the *Workspace Commands*. The response will be:

```
VSERROR- vs: Expected: "PROJECT?", or "PROJECT:", or "GAMEOVER"
ZZZZZZZZ
```

*Vs* will then display the input prompt and wait for more input.

## 4.2 Version Management Commands

| COMMAND | EFFECT |
|---|---|
| CHECKIN-□*options* file | Store a new revision. |
| CHECKOUT□*options* file | Retrieve a revision and store it in the workspace |
| DFDMERGE□*rev* file | Merge branch onto top of trunk. |
| GAMEOVER□ | Exit gracefully from *vs*. |
| RCSDIFF-□*options* file | Compare two revisions. |
| RCSLIST-□ | Display a list of the files under version management. |
| RCSMERGE□-r*rev1* [ -r*rev2* ] [ -p ] file | Add changes between *rev1* and *rev2* into the working file. |
| SENDRLOG□*options* file | Display the revision information. |
| ZZZZZZZZ□ | Return to the previous level. |

---

The □ is really a space (0x20) character and is required by the protocol.

### 4.2.1 CHECKIN- Command

The *checkin* command stores new revisions into revision files. Each revision file has the filename ending of `,v`. In addition, each revision file has a status file accompanying it with the filename ending in `-vlog`. The revision files and status files are stored in a directory named RCS that is directly below the current directory level. Since the program automatically creates a RCS directory for each workspace that it creates, it assumes the RCS directory exists.

Whereas in `ci` RCS files can be specified in three ways (see the `ci(1RCS)` man page), in *vs* they can only be referenced by the working filename. The filename for the revision file and the status file are thus derived from the given filename.

*Vs* always looks in the directory `./RCS` and nowhere else for the revision file and the status file. If these files are not found an error is reported and the command is terminated.

The *checkin* command requires that user have read/write access permission in the directory `./RCS` and on the status file. It requires that the user have read access permission on the working file, and the revision file. The *checkin* command automatically sets these permissions when it is creating these files and the directory.

The *checkin* command provides automatic handling of revisions with optimistic version control. It will check the revision number of the working file against the top-of-trunk revision number and if they are not different, it will proceed to deposit the revision and output the outcome of this execution. On the other hand, if the revision number on top of trunk is greater than the revision number of the working file, the *checkin* command will automatically store this revision on the next available branch based on the revision number of the working file. This assure that no changes will be lost when multiple people are working on the top of trunk and that revisions are deposited on a top of trunk on a first to arrive, first to checkin basis.

All errors encountered by the *checkin* command will be output to stderr. Each error message will be introduced by the error prompt:

```
VSERROR- vs:
```

After outputting the error messages, *vs* will exit this command.

After successful depositing of the revision, *vs* will update the status file. The status file will contain a record of the check in with the date and time, the user and the revision number. This information is never deleted and contains an exhaustive history of the revision file.

Although all the options available with `ci` are available with the *checkin* command, this command is most useful for default case, where all that is specified is:

**CHECKIN-** *filename*

The *checkin* command also features the option *-p*. With this option, the input for the working file is taken from stdin and the *checkin* command will attempt to deposit the revision file as soon as the completion protocol message **ZZZZZZZZ** is received. Thus all data received from stdin until the completion protocol message will be incorporated in the filename given.

### 4.2.2 CHECKOUT Command

The *checkout* command retrieves revisions from revision files. Each revision file has the filename ending to `,v`. In addition, each revision file has a status file accompanying it with the filename ending in `-vlog`. The revision files and status files are stored in a directory named RCS that is directly below the current directory. Since *vs* automatically creates a RCS directory for each workspace that it creates, it assumes the RCS directory exists.

When using the RCS `ci` command, the RCS files can be specified in three ways (see the `co(1RCS)` man page), but with *vs* these files can only be referenced by their working filename. The filename for the revision file and the status file are thus derived from the working filename.

*Vs* always looks in the directory `./RCS` and nowhere else for the revision file and the status file. If these files are not found an error is reported and the command is terminated.

The *checkout* command requires that user have read access permission in the directory `./RCS`, on the revision file and on the status file. It requires that the user have permission to write on the working file. The *checkin* command automatically sets these permissions when it is creating these files and the directory.

The *checkout* command provides automatic handling of revisions with optimistic version control. It does not require that the user check out a file locked to change the file and check it in. *Vs* will record each check out in the status file. This allows *vs* to keep track of all parties who are working on a given revision file. This information can be used by other programs or routines for project tracking purposes.

The actual revision file check out is handled by the RCS `co` command. All the options supported by the `co` command are supported by the *checkout* command. This includes both keyword substitution and command line options within `co`.

All errors encountered by the *checkout* command will be output to stderr. Each error message will be introduced by the error prompt:

```
VSERROR- vs:
```

After outputting the error messages, *vs* will exit this command.

After successful execution of the check out, *vs* will update the status file. The status file will contain a record of the check out with the date and time, the user and the revision number. This information is never deleted and contains an exhaustive history of the revision file.

### 4.2.3 DFDMERGE Command

The *dfdmerge* command provides additional functionality to the RCS `co -jbranchrevno:trunkrevno` command. The *dfdmerge* command takes the branch revision specified and merge the changes between the branch revision and the trunk revision from which it emanated onto the top of trunk.

To do this *vs* will execute a RCS `co` command. It will then execute a `rcsdiff` command. *Vs* will then use its internal decision tree to determine which changes can automatically be applied onto the top of trunk, i.e. the changes meet the project policy.

The decision tree specified can have one of two actions:

1. Make the change specified.
2. Do not make the change specified.

Upon successful completion of the *dfdmerge* command the input prompt will be returned and *vs* will await more input. At this point the user has a new working file that is a modification of the top of trunk. All errors encountered by the *dfdmerge* command will be output to stderr. Each error message will be introduced by the error prompt:

```
VSERROR- vs:
```

After outputting the error messages, *vs* will exit this command.

### 4.2.4 GAMEOVER Command

The *GAMEOVER* command is used to exit from *vs*. Once the command is issued successfully the program will cleanup after itself, exit from the program and return the UNIX prompt.

### 4.2.5 RCSDIFF- Command

The *rcsdiff* command will compare two revisions of the revision file specified. This command is exactly the same as the RCS `rcsdiff` command in UNIX. See the `rcsdiff(1RCS)` man page for more information.

After executing the RCS `rcsdiff` command, *vs* will output the completion protocol message. *Vs* will then output the input prompt and wait for more input.

### 4.2.6 RCSLIST- Command

The *rcslist* command will list the revision files in the RCS directory. Each revision file will be listed by its working filename proceeded by the introductory protocol message **VMESSAGE**.

If no RCS directory exists directly below the current working directory, *vs* will output the error prompt **VSERROR**– followed by the message **`VSERROR vi: unable to open dir "RCS"`**.

After listing the files or sending the error prompt, *vs* will output the completion protocol message. *Vs* will then output the input prompt and will wait for more input.

### 4.2.7 RCSMERGE Command

The *rcsmerge* command will compare two revisions of the revision file specified and then update the working file based on the differences between the two revisions. This command is exactly the same as the RCS `rcsmerge` command in UNIX. See the `rcsmerge (1RCS)` man page for more information.

The *rcsmerge* command after executing the RCS `rcsmerge` command, *vs* will output the completion protocol message. *Vs* will then output the input prompt and will wait for more input.

### 4.2.8 SENDRLOG Command

The *sendrlog* command will display the RCS history and status of the file specified. This command is exactly the same as the RCS `rlog` command in UNIX. See the *rlog(1RCS)* man page for more information.

The *sendrlog* command after executing the RCS `rlog` command, *vs* will output the completion protocol message. *Vs* will then output the input prompt and will wait for more input.

### 4.2.9 ZZZZZZZZ Command

The *zzzzzzzz* command will reset the working space to to the original directory. Upon successful completion the command, *vs* will be placed in *Workspace Commands* mode.

After completing the *zzzzzzzz* command, *vs* will output the completion protocol message. *Vs* will then output the input prompt and will wait for more input.

### 4.2.10 Version Management Command Error

When *vs* fails to understand a command, it will display the error prompt:

```
VSERROR- vs: Unable to parse command
```

and then indicate allowed the *Version Management Commands*. The response will be:

```
VMESSAGE vs: Expected: "CHECKIN- ", "CHECKOUT ",
                       "RCSDIFF- ", "SENDRLOG ",
                       "RCSMERGE ", "ZZZZZZZZ ",
                       "RCSLIST- ", "DFDMERGE" or
                       "GAMEOVER "
ZZZZZZZZ
```

*Vs* will then display the input prompt and wait for more input.

# 5. COMPILING AND MODIFYING THE VS SOURCE CODE

For all of you who are so brave as to attempt to look at this code, there is a Makefile provided.

The source code is divided up into four source modules: definestring.c, definetables.c, dfd_merge.c, and vs.c and four header files: definestring.h, definetables.h, dfd_merge.h and rules.h.

The vs.c file contains the main function.

# 6. FILE FORMATS

## 6.1 Large-Grain Data Flow 2 Diagram Working File Format

### Synopsis

filename

### Description

This ASCII file will contain the data for a particular revision of a Large-Grain Data Flow Diagram revision file. It is formatted as a series of lines of the form:

Protocol_Message Data

The protocol message is a field introducing the data that follows it. The protocol messages are:

```
NODE----,   PROCESS-,   LINE----,   SWITCH--,   CCCCCCCC,
ZZZZZZZZ,   CENTRXY-,   RADIUS--,   NAMESTR-,   ULEFTXY-,
PNUMBER-,   DNUMBER-,   STARTXY-,   PERMISS-,   ENDXY---,
SNUMBER-, WIDTH---, HEIGHT--, TITLE---, #, $,
```

The # introduces any comment line.

The $ introduces any *RCS* marker data.

The CCCCCCCC token is used to separate the LGDF2 data from the *RCS* marker data.

The ZZZZZZZZ introduces the end of the file.

All other protocol messages introduce LGDF2 data. The following commands introduce elements of the LGDF2 file.

```
NODE----
PROCESS-
LINE----
SWITCH--
```

These commands have the following formats.

### 6.1.1 LINE---- Element Format

Each line element will have the following data:

```
DNUMBER- number of the line
ULEFTXY- x_location and y_location of the DNUMBER-
STARTXY- x_location and y_location of line start
```

**LPNUMBR-** *number for the process attached at the start or end*
**PERMISS-** *permission on line start*
**ENDXY---** *x_location and y_location of line end*
**LSNUMBR-** *number for the switch attached at the start or end*

**PERMISS-** *permission on line end*

### 6.1.2 NODE---- Element Format

Each NODE element will have the following data:

**TITLE---** *title of the node*
**ULEFTXY-** *x_location and y_location of the TITLE*

### 6.1.3 PROCESS- Element Format

Each process element will have the following data:

**CENTRXY-** *x_location y_location*
**RADIUS--** *size*
**NAMESTR-** *name of the process element*
**ULEFTXY-** *x_location and y_location of the NAMESTR-*
**PNUMBER-** *number for the process*
**ULEFTXY-** *x_location and y_location of the PNUMBER-*

### 6.1.4 SWITCH-- Element Format

Each switch element will have the following data:

**ULEFTRXY** *x_location y_location*
**WIDTH---** *width of switch*
**HEIGHT--** *height of switch*
**TITLE---** *name of the switch element*
**ULEFTXY-** *x_location and y_location of the TITLE*

## 6.2 -vlog Format

## Synopsis

./RCS/*filename*-vlog

## Description

This file contains the revision history of the associated revision file. It is formatted as a series of lines of the form:

**action weekday month day time year username revision_number**

This is an ASCII file with fixed length fields except for the revision number which is completed with "\n" (the newline character). The action field is either `ci` or `co`. The fields `weekday`, `month`, ..., and `year` are result of output from the Utek `date` command. The username is padded with leading spaces to make it 21 characters wide. All characters in the username that exceed this width will removed.

### 6.3 ,v Format

## Synopsis

./RCS/*filename*,v

## Description

This file will contain the revisions for the working file. It is a standard *RCS rcsfile* with the exception that it contains the revisions of the LGDF2 working file. For more information on the format of the ASCII file, see the *rcsfile(5RCS)* man page.

*INDEX*

# APPENDIX D. UNIX Man Page for VS



**NAME**

vs - A Version Management System for Large-Grain Data Flow Diagrams

**SYNOPSIS**

vs [-echo] [-debug *value*]

**DESCRIPTION**

*Vs* takes a set of inputs from stdin and manages multiple revisions of pseudo text files. This system provides the features of the Revision Control System (RCS) by Walter F. Tichy and additional functionality provided via a semi-intelligent interface program.

**OPTIONS**

**-echo** Display on stdout the commands input from stdin.

**-debug** *value*

Specify the level of debug messages to be printed.
Value 0 - all debug messages are printed.
Value 6 - (the default) no debug messages are printed.

**Getting Started with VS**

Rather than repeat all that is written in the the documentation on the Revision Control System by Walter F. Tichy, I will suggest starting by reading the following documents:

*Design, Implementation, and Evaluation of a Revision Control System* by Walter F. Tichy, in *Proceedings for the 6th International Conference on Software Engineering*, IEEE, Tokyo, Japan, September, 1982. Unix Programmer's Manual documentation *RCSINTRO* and the man pages for each element of rcs used.

Vs provides a superset of the RCS commands with some additional functionality not available in other version management systems. Once vs is invoked it acts as command language interpreter for version management commands. After being invoked successfully vs replies on stdout with the message *vs: started.*

The command interpreter then waits for input from stdin. It parses the input and acts according to it notion of version management.

When no working space has been set vs accepts the workspace commands:

```
PROJECT?
PROJECT:
GAMEOVER
```

If a command is given that vs does not understand vs responds with the message:

```
VMESSAGE vs: Unable to parse command
VMESSAGE vs: Expected: "PROJECT?", or "PROJECT:", or "GAMEOVER"
22222222
```

and then waits for more input.

When a working space has been set successfully vs accepts the version management commands:

```
CHECKIN-
CHECKOUT
DFDMERGE
GAMEOVER
RCSDIFF-
RCSLIST-
RCSMERGE
SENDRLOG
22222222
```

If a command is given that vs does not understand vs responds with the message:

```
VMESSAGE vs: Unable to parse the command input.
VMESSAGE vs: Expected: "CHECKIN- ", "CHECKOUT ", "RCSDIFF- ",
                       "SENDRLOG ", "RCSMERGE ", "ZZZZZZZZ ",
                       "RCSLIST- ", or "GAMEOVER "
ZZZZZZZZ
```

and then waits for more input.

## COMMAND SYNTAX AND SEMANTICS

The workspace commands have the following syntax and semantics.

**PROJECT?**␣
Print out the list of workspaces directly under the current directory.

**PROJECT:**␣*name*
Change to the workspace with the given *name*.

**GAMEOVER**␣
Exit the version control shell.

The version management commands have the following syntax and semantics.

**CHECKIN-**␣*string*
Use the *ci* command to store new revision into version control files.

**CHECKOUT**␣*string*
Use the *co* command to retrieve revisions from version control files.

**DFDMERGE**␣*string*
Use the *dfdmerge* command to incorporate the differences between two revisions of a version control file into the corresponding working file.

**GAMEOVER**␣
Exit the version control shell.

**RCSDIFF-**␣*string*
Use the *rcsdiff* command to compare two revisions of each version control file given.

**RCSLIST-**␣
Print out the list of pseudo-files under version control under the directory ./RCS.

**RCSMERGE**␣*string*
Use the *rcsmerge* command to incorporate the differences between two revisions of a version control file into the corresponding working file.

**SENDRLOG**␣*string*
Use the *rlog* command to print out status information about version control files.

**ZZZZZZZZ**␣
Change back to level and reset the workspace.

The ␣ is really a space (0x20) character and is required by the protocol.

## IDENTIFICATION

*Author:* Charles Adams, Oregon Graduate Institute of Science and Technology, Beaverton, Oregon
Revision Number: 0.2 ; Release Date: 90/11/06.

## SEE ALSO

rcsintro(1L), ci(1L), co(1L), rcs(1L), rcsdiff(1L), rcsmerge(1L), rlog(1L), rcsfile(5L)

## BUGS

Many but unknown at this time.

# BIOGRAPHICAL NOTE

The author was born 28 January 1948, in Madison, Wisconsin. He attended various public schools in Madison until 1964 when he moved to West Covina, California. He graduated from Covina High School in 1966.

In October 1966 the author began military service in the United States Air Force. He was stationed for the most of his duty in West Germany where he attained the rank of Staff Sergeant and was Honorably discharged in August 1970.

In September 1970 the author entered Los Angeles Valley College and transferred to California Polytechnic State University of San Luis Obispo in September 1972. He graduated with a Bachelor of Arts in Speech Communication in June 1975.

In August 1975 the author entered West Virginia University as a teaching assistant and graduated with a Master of Arts in Speech Communication in August 1976. He then attended University of Oregon from September 1976 to June 1978.

In June 1978 the author began a position as Marketing Manager for Northwest Microcomputer Systems. This position lasted until June 1982 when the author began working for Electro Scientific Industries as a Lead Technical Writer.

The author began his present position as Software Engineer with Tektronix, Inc. in January 1987. He has been married for five years to the former Gloria Smith.