AVCS: The APL Version Control System

Nikolai I. Puntikov
Russian Academy of Sciences
Institute of Linguistics
9 Tuchkov per.
St.Petersburg 199053
Russian Federation
Telephone&Facsimile: +7-812-213 5127
Nick@iha.spb.su

Maxim A. Volodin
Alexei A. Kolesnikov
ISM Partnership
52 1st Line V.O.
St.Petersburg 199053
Russian Federation
Telephone&Facsimile: +7-812-213 5127
Maxas@iha.spb.su

Abstract

This paper describes AVCS, which is an APL-oriented version control system devised as a tool to track the history of software projects and to control concurrent access to project components. The basics of version control systems are explained, and specific aspects of applying a version control system methodology to project development in APL environments are considered. Particular attention is given to features which differentiate the approach accepted in AVCS from that of available version control and project management systems. Specification of AVCS's data maintenance, programming and user interface is presented to the extent required in order to explain how the system works. In conclusion, the possible application of AVCS to solving problems of porting APL projects across different environments is outlined.

Keywords Version control, project management

1 Introduction

The need for version control systems is widely recognized today as a means to track the complete history of each component of an application and to protect against uncontrolled changes. Modern software development is often performed by a team of programmers working in a distributed environment. If two programmers attempt to change the same file simultaneously without any mechanism to prevent one from overwriting the other's changes, the result can easily be a disaster. The larger the system the more important proper version control becomes.

There are other applications of version control not limited to team development. Consider, for example, a search for the best solution to a certain problem by trying out different algorithms. This may be handled by developing several versions of a function and selecting the one that best satisfies the chosen criteria.

Another example is program maintenance, where version control also plays an important role. Bertrand Meyer [Meyer 1988] estimates that 70% of the cost of software is devoted to maintenance. Once a software system has been released to customers, it often becomes necessary to make changes, due to new specifications suggested by the customer, or to fix bugs revealed by the customer. At the same time the designers of the system may be working on a new version. In order to deal with the problem of having to work on old and new versions simultaneously, software engineers need to use version control systems.

In this paper we observe the problems of using conventional file-oriented version control systems in APL-based project development. At the same time we consider the bottlenecks of existing APL-oriented project management systems with regard to the needs of development and maintenance of a general software project.

In order to address these problems, we decided to develop the APL-oriented Version Control System (**AVCS**) as a tool built around the notion of a *software project* which could consist of one or several workspaces, documentation files, or even non-APL files.

The implementation of the system prototype has been done on a PC platform with Dyadic Dyalog APL for Windows. By the time we present this paper we plan to port AVCS to IBM APL2/PC and Manugistics APL*PLUS.

We do not pretend to cover the subject of version control completely, but we believe that the approach explored in this research might prove helpful to the developers of future APL systems, and our software might prove useful to the programmers of contemporary APL-based applications.

APL '95, San Antonio, Texas, USA

2 Overview of Available Systems

Since version control systems are not widely used by APL developers, this section, along with the survey of available version control and project management systems, provides a short introduction to the basic version control system concepts and terminology used throughout this paper. In the subsequent sections we will examine how these concepts are transformed in AVCS.

Most available tools can be divided into two groups:

- file-oriented systems
- repositories of APL objects (variables, functions and operators)

File-oriented systems (Microsoft Delta and PVCS are good examples) define version control as a file management system which tracks the history of changes to components of a software system [INTERSOLV 1993].

A *project* in a version control system is a group of related files to which access can be restricted and in which changes can be archived [Microsoft 1993]. Users who have access to the project files and their version histories are *project members*.

The version archive is called the *master project*. Archives contain the changes made to a file, descriptions of the changes, information about who made the changes, and the dates and times when the changes were made. The master project serves as a library from which project members can retrieve the current version of the file as well as any previous version back to its original copy.

A snapshot of the project state reflecting certain milestones in the course of development is a *project release*. The release points to fixed versions of project files.

The master project files typically are stored on a network server or in another central location. A project member owns a *local copy* of the project. To gain access to the changes made by other members the local copy needs to be *synchronized* with the master copy. This means the outdated files will be replaced in the local copy by the new revisions coming from the master copy.

After synchronization, local files are available in read-only mode. In order to edit a file the user needs to *check* it *out* (that is, request it) from the version control system. When a file is checked out by one user, other users cannot alter the master copy until the first user finishes editing and *checks* the file back *in* to the version control system, which would create a new version of the file.

Delta is differences between two versions of a file. The display of delta would show lines which were added, deleted or changed. (Some version control systems distinguish between *text* and *binary* files; display of delta is available for text files only.)

Reporting features are used to query the master project for the list of components, histories of edits and releases, and so on.

Project administration is supported by tools to create or delete projects, define the location of master projects, indicate when projects are ready to release, and manage project membership.

The problem of applying a file-oriented approach to APL projects is that defined procedures, which are the principal components of APL applications, are not stored in native system files. One might suggest treating the entire workspace as a single application component, but this is not effective. APL programmers need to relate to each component in a workspace to be able to manage the environment in a reasonable way.

Some attempts have been made to make use of conventional version control systems with APL. For example, John Mizel [Mizel 1992] describes how the UNIX Source Code Control System (SCCS) tool can be utilized to save versions of APL transfer files. Although the approach is effective, it appears to be a bit awkward. Steps such as using transfer files or physically dividing workspaces between developers do not look natural.

Two examples of APL-based version tracking tools are: Workspace Manager (**WSM**), which acts as a repository of APL objects [Swain&Jonusz 1993], and **LOGOS**, a SHARP APL software development environment [Reuter 1990]. (In fact, LOGOS is not just a version control tool, but a powerful system addressing many of the issues of APL-hosted development which are beyond the scope of this paper.)

In our opinion, these two systems have still another drawback when compared with file-oriented systems; they are far too APL-specific.

WSM treats the end environment exclusively as an APL workspace, which is not necessarily true. The real software project often relies on additional native system files, which may be either static data files, or documents, or a general-purpose software library. Sometimes it is convenient to treat even an APL workspace as a single binary component of a project. This might be useful, for example, when a developer wants to fix the version of a third-party workspace on which the developed software depends and which contains locked functions. There is no other way to save data of this kind, since it is not possible to gain text representation of locked functions.

LOGOS accepts the importance of files in applications and provides a set of utilities to bring files into the end environment. However, files are not considered to be

objects in the LOGOS database. Consequently, they are not subject to version tracking.

Another problem with LOGOS is its complexity. Many LOGOS features are very attractive, but are not directly related to version control. We believe that in an APL integrated environment, version control should be distinctly separated from other aspects of software engineering.

WSM and LOGOS both place a primary emphasis on maintaining a shared repository of objects from which an end environment can be built. On the contrary, we wanted to have a system in which the attention from the very beginning was shifted toward the target application, albeit the issues of redundancy and reusability would not be addressed explicitly.

In fact, these two ideas complement each other nicely. The software project may benefit from any particular company's repository of reusable components. However, the intention of project management and version control systems is to help ensure project integrity, robustness and extendibility. A version control system should be a handy tool, which is easy to operate and which provides access to a database storing versions of components of the whole project.

3 AVCS Design

3.1 Object database

An important question in the design of a version control system is how the database of objects should relate to the application (the end environment). In any such system, two methods may be adopted. According to the first method (LOGOS and WSM use it), the database contains a pool of objects that may be combined to produce an end environment. LOGOS provides a rich set of tools to introduce a structure into this pool, which makes the method rather efficient, especially for large team of developers. However, such an approach is excessive for small and medium-sized applications. Besides, as already noted, we think that the database of general-purpose reusable components should be separated from that of the particular software project.

A simpler method suggests associating an object database with the end environment. AVCS takes this approach. An *AVCS master project* is defined as a repository for everything necessary to build a software system. Therefore, an AVCS project corresponds to an application. Strictly speaking, this is not enforced by AVCS, except that there are no obvious reasons why someone should mix irrelevant components into one project.

In the case of team development, a project could be stored on a network server. The members of the development team would then have copies of the project at their local workstations. These copies could be partial. They would be created when a developer *joins* the project.

Developers can join a project only if they are already *registered* for that project. Registration of a developer is carried out by the *project administrator*. By convention, the project administrator is the developer who creates the project. The administrator establishes the *user identification record* within the project. Initially it contains the user's password only. After the user goes through the process of joining, that record will also contain a path to the user's *local root directory* at the user's workstation where the local copies of the project files reside.

Currently we have decided against sharing data between AVCS projects. AVCS projects store the evolutionary history of components of one application, and at the same time, components of an application subject to version control tracking are entirely stored in one AVCS project. We have already expressed our attitude toward sharing in terms of reusable utility components. We think they should be simply copied into the AVCS project.[1]

AVCS distinguishes between two types of components: workspace-related and external files.

A *workspace component* of an AVCS project is a variable, function, or operator.

A *file component* represents a file of any type.

Note that WSM and LOGOS refer to database entities as *objects*. For AVCS database entities, we have chosen the term *component* deliberately. The term *object* is rather loosely used nowadays in a variety of contexts. Many APLers, for example, may understand it as a reference to a workspace object. We wanted to emphasize that in the AVCS project, one can save application components of any type.

Functions and operators (defined operations) are *text components*. Variables and files are *binary components*.[2]

A *component version* is a distinct state of the component stored in the master project. Versions are identified with a subsequent *version number*.

3.2 Accessing the AVCS project

A component version may not be altered directly. In order to change a component, a developer needs to run through a process known as *check-out*. This process yields a copy of a component for the developer's workspace (workspace

[1] A very attractive feature of version control would be sharing in the sense of depending on the existing project or subproject. We hope to implement this feature in future versions of AVCS.

[2] A user would certainly benefit if differences in versions of text files (e.g., documents) and variables were available. (If the structure of the variable were changed during its evolution, then the system might at least report the changes in the rank or shape.) We also hope to implement this in the future.

components) or local root directory (file components) and marks the component in the master project as *checked-out*. A component which is already checked out cannot be requested again. This is a safeguard against parallel changes to the same component.

The reverse of check-out is *check-in*. This process is run in order to save a modified component. After check-in is completed, a new version of the component becomes available to other developers.

Of course, one can edit every component which resides in the local workspace. However, unless the developer has this component checked out, the AVCS software will not allow the creation of a new version of the edited component. This convention (adopted by almost all file-oriented version control systems) is safe, but may be inconvenient. The problem is that an APL-oriented version control system cannot prevent accidental changes in the same way that traditional systems do.

Conventional systems set the read-only mode for those files which cannot be modified. AVCS uses this method for file components too. With workspace components, however, this approach does not work. Neither variables nor functions nor operators can be made read-only.

AVCS works around this problem by providing a facility to query the status of a component. Of course, this facility cannot prevent unauthorized editing, but it can at least help in detecting the 'read-only' components. In addition to that, the AVCS check-in algorithm provides a feature which can, under certain circumstances, rescue an absent-minded developer. A modified component can be checked in even if it was not checked out previously, only if the component is not already checked out by another user and if the version of the component in the master copy matches the one in the local workspace. Since these conditions can by no means be guaranteed, it is strictly not recommended to rely on this AVCS feature.

We also considered other approaches. The most obvious solution is to lock the function or the operator. However, this seems to be too restrictive, because it prevents a programmer from examining the code.

A better idea is to insert a warning message right after the operation header. The warning would say: "This function is not checked out. Changes will not be recorded in the master project." Of course, such a comment would never appear in the production code, because this code would indeed never get checked in. Unfortunately, this solution does not work either. Inserting comments into the function code may destroy its integrity if it contains number-based (instead of label-based) branching. Besides, such an approach is suitable only for functions and operators, but not for variables.

AVCS includes reporting features to show the revision chain. These reports include information about the number of revisions kept and the identifier of the user who has introduced a particular version. If a user supplied an explanatory text for a version, this text would be shown too.

Using this information, a user can deduce whether or not the master project contains a new version of a certain component. If it does, users may decide to update their local copy of the component. This process is known as *synchronization* with the master project. Synchronization can be performed both for a single component and for the entire local copy. In the latter case, not only all components already existing in the local copy will be synchronized, but also new components created by other developers will be copied.

At a certain stage of the project life-cycle the project administrator may decide that the project is ready to be released. When the administrator *defines* the *release*, AVCS verifies that there are no project components checked out by any user, and links the latest versions of all components to the *release record* in the master project. Starting from this point, the entire release could be referred to by its name.

3.3 Coordinated Interaction with the Master Project

The previous section presented a more or less common scheme of interaction with the object database accepted by many systems. In our opinion, this scheme can be improved.

Suppose that a developer wants to introduce a new feature into a system. Suppose that several components (say, two functions) need to be changed to achieve this. If so, then the developer has to check out the required components and modify them. After that the developer must check the components back in to the master project and supply notations describing the changes.

The problem with this approach is that the result of this activity is spread over the project database. The changes made to functions have no connection with each other, although they were made during the same activity: the process of introduction of a new feature. To roll back the changes (suppose that the new feature was poorly implemented), one needs to hunt down both functions, and this is not an easy task. We know from experience that developers tend to record *what* changes they made to a component, but not *why* they were made. The notations for any changes often give little or no help for figuring out the reason for the changes and which other components were involved.

To resolve this problem, AVCS introduces the notion of *activity*, which is the means to combine several logically related procedures into one. With this tool at hand, the above example may be reorganized as follows:

- First, the developer starts an activity, specifying what the activity is intended to accomplish (introduction of a new feature, in our case).
- Next, the developer checks out the required components. The AVCS software associates these check-outs with the opened activity.
- Then the developer edits the components and describes the changes, as the editing progresses.
- After finishing the coding and debugging, the developer closes the activity. At this point, AVCS checks in all components which had been checked out since the activity began.

Note that the notations provided during the second step ("Why am I doing this?") remain in the database and are available whenever needed. Another thing stored in the database is a time stamp: when the activity was opened and when it was closed. This introduces a development tracking facility into AVCS. With coordinated activities, the whole history of changes to the project is available to the viewer.[3]

4 AVCS Implementation

What follows is an informal description of the AVCS implementation. The intent is to provide an idea of how AVCS information is maintained and used. Insignificant details are deliberately omitted.

4.1 Data Storage

AVCS maintains the master project in several files stored under the same directory. The actual number of files depends on the number of registered users and the number of components in the master project. The principal structure is as follows:

- The master project file stores the list of user identification records, the list of component identification records, the list of release records, and the list of opened activities.
- Component files (one file per component) store the revision chain of each component; each version stores the contents itself together with version-related data: date and time of creation, owner ID and notations.[4]
- Check-out tables (one table per user) identify components requested for editing.
- Activity files (one file per opened activity) list components involved in the activity.
- Release tables (one table per release) identify versions of components included in the release.

AVCS does not provide shared access to the master project. This is because project members would usually require a short time to access a database in order to read or write components and/or retrieve certain information. Any AVCS interface function would open the archive and then close it after requested transactions have been performed.

4.2 Programming Interface

Developers work with the master project database by means of a package of APL functions. That package can be brought into the workspace with the `)COPY` command. The package also contains the `VCS_WipeOut` function which wipes all AVCS functions and variables out of the workspace when they are no longer needed.

We realize that such a method risks name conflicts, but we have not found another way which would be portable across different platforms. Currently, the safeguard against name conflicts is the uniform prefix `VCS_` given to all AVCS functions and variables. Developers should name their own objects with this fact in mind.

The AVCS programming interface defines a few basic functions to access an AVCS master project. Here we give some examples of their usage. These examples are provided in order to give a feeling for how AVCS works, and they are not very formal. In particular, the error signal mechanism is not considered. Examples are presented in the order that one might imagine one's work with AVCS.

The first task a user has to accomplish before any other calls are executed is the initialization of the user's ID. AVCS uses this global variable to distinguish which user accesses the project:

```
VCS_UserId←UserName Password
```

To create a new project a user calls the following function:

```
[Comment] VCS_CreateProject Path Name
```

This function creates project files and initializes project data. The user defined in `VCS_UserId` becomes the only project member and project administrator.

The function also initializes the workspace global variable `VCS_ProjectHome` with a character vector containing a full path to the newly created project. This variable is used by AVCS functions to identify the current project.

[3]Currently AVCS does not support nested activities. They are subject to further development.

[4]Currently the system stores the entire contents of the component; the alternative is to save delta, the way many version control systems do. However, the benefit of this approach is questionable, since APL defined operations are usually much smaller than non-APL program modules.

AVCS does not provide a designated function in order to open a project. Instead `VCS_ProjectHome` is used to refer to the existing project.

Adding a component to the project is performed with the `VCS_AddComponent` function:

```
[Comment] VCS_AddComponent Name
```

This function checks that the `Name` is unique in the project, creates a component file in the master project directory, and stores component contents as version 1. Note that starting from this point the component becomes available to other members for editing. The read-only mode will be set for the file component.

The `VCS_AddComponent` function should not be confused with `VCS_AddVersion`:

```
[Comment] VCS_AddVersion Name
```

In spite of the similar syntax, these operations have different semantics. Adding a component means creating a new entry in the master project (a task often assigned to the project administrator in traditional version control systems). Adding a new version corresponds to checking a component back in to AVCS. This results in saving the new state of the component with an incremented version number.

The check-in can only be performed if the component was previously requested from AVCS for editing.[5] The ambivalent `VCS_GetVersion` function is used for this purpose. The monadic application of the function checks the component out and, if necessary, copies in the local workspace (or root directory) the latest version of the component:

```
VCS_GetVersion Name [Comment]
```

The system notes that the component was checked out and rejects any requests to edit this component from other users until it is checked back in.[6] The `Comment` notes why the check-out was requested.

With dyadic usage a user can request from AVCS any particular version of the component:

```
Version VCS_GetVersion Name
```

On this call AVCS would not lock the component in the master project.

Synchronization means retrieving the latest version of a component for the local workspace or root directory. This feature plays an especially important role in team development when new versions of components appear beyond one's control:

```
(Owner Date Comment)←VCS_Synchronize Name
```

The user receives information about the last version of the component identified by `Name`. Both `VCS_GetVersion` and `VCS_Synchronize` replace the current local version (if any) in the active workspace.

The problem of batch synchronizing the whole project can easily be resolved by means of APL. AVCS provides a function to get a list of either all component names or component names of specified types defined in the master project:

```
Component_Names←VCS_GetNames Name_Type
```

where `Name_Type` set to 0 means the complete list of all components, including files.

The following expression performs a global synchronization:

```
VCS_Synchronize ¨VCS_GetNames 0
```

All outdated components will be replaced by those in the master project.

There is a set of informational functions in AVCS. Worth mentioning is a function informing the current member about components that have been checked out:

```
CheckedOutList←VCS_CheckedOut Name_Class
```

Again the same `#each` technique may be applied to check in all components that have been checked out:

```
VCS_AddVersion ¨VCS_CheckedOut 0
```

When the project develops to a certain stable state, the administrator defines a release by using the following function:

```
[Names] VCS_Release Release_Id [Comment]
```

The optional `Names` parameter allows the administrator to specify explicitly the names of components that must be included in the release. By default, the release includes all components.

There are other functions available from the AVCS programming interface. Since knowing their syntax would not help much in understanding AVCS concepts, we have not considered them in detail.

4.3 Graphical User Interface

Although the programming interface is sufficient to have AVCS perform real-world version tracking, access to the main facilities through a graphical user interface (GUI) is desirable because it can simplify the usage of the system.

We have developed an experimental GUI in Dyalog APL for Windows. It is implemented as a set of windows which provide interactive access to AVCS services. For example, to perform basic system initialization (user name, project path, release number), a user supplies appropriate data in text-entry fields in the window. The component-related services are available via push buttons and list boxes filled with the names of components.

[5]Overriding this convention, as described above, is achieved by the `VCS_RescueVersion` function.

[6]In an emergency, the project administrator can override the checked-out state of the component by the `VCS_DiscardCheckOut` function.

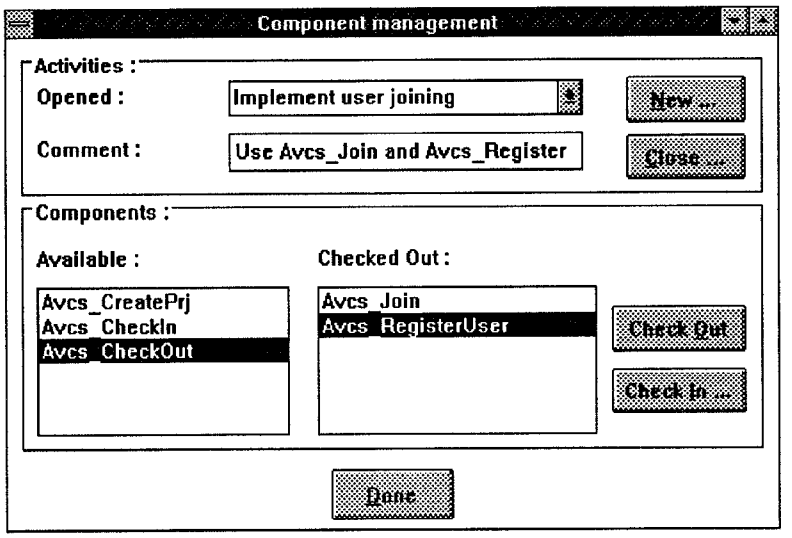

Figure 1. AVCS Component Management Window

The sample window (Figure 1) illustrates how the user can proceed with check-in/check-out of components within a selected activity.

In the *Activities* combo box the developer either selects the existing activity (*Opened* drop-down list box), or defines a new one (*New...* button), or closes the current activity (*Close...* button).

The information specified in the *Comment* text field relates to all components involved in the specified activity.

To close the activity, the user presses the *Close...* button and a new dialog appears displaying the list and the state of the components related to this activity. The activity cannot be closed if there are components linked to it that are checked out by other users.

In the *Components* combo box, the *Available* list box contains project components which can be checked out (that is, are not currently checked out by any other user).

The *Checked Out* list box presents components checked out by this user. These are the components which the user is allowed to check in.

When the *Check In...* button is pressed, a *Check-In* dialog helps the user to select components for check-in and to provide notations for selected components.

5 Implementation Notes: The Issue of Portability

Since AVCS had been targeted for implementation in three APL systems (Dyalog APL, APL*PLUS and APL2/PC) from the earliest stage of design, we decided to store project and component data in a uniform format. Thus, the same repository can be accessed in different APL systems. This allows a somewhat unexpected application of AVCS: porting across APL systems, providing AVCS is implemented on both the source and target platforms.

There are problems with this approach. The obvious one is that each different system uses a different character mapping. The decision was made to implement translation modules as an internal part of AVCS. This hurts performance, but not excessively. Actually, AVCS was never intended for use in real-time applications, since it is a programmer tool. In the long run, the system may be cross-customized so that minimum translations are performed in the host system. Perhaps the AVCS master project files need to contain a "host system" identification record, so that each AVCS read/write operation could compare the value stored in this record against an internal value; if they match the translation can be skipped.

Another problem is storing APL arrays. Each environment defines an individual format for the internal representation of variables. Our method is to store all arrays as character arrays. We are fully aware of the impact on performance which this formatting might make, but we based our choice on the assumption that updates of project variables (whether they are tax rates or report headings) stored in the version control system should not often be necessary. If these data items change very often it would be better to put them in an application file or production workspace.

Of course, the issue of portability involves more complicated problems concerning language differences. To some limited extent AVCS can help to deal with this too. Consider the following simple examples:

- The IBM APL2 interpreter recognizes overbar as a part of an APL name; Dyalog APL does not allow this.
- The *#first* primitive is implemented differently in IBM APL2 and Dyalog APL. The former uses ↑ while the latter uses ⊃ (default migration level).

It is quite possible to develop a function which would scan through all functions and operators of the project to perform global search and replace. With this function, accommodating language differences (as in the above examples) would be much easier.

The *rename* feature could also help with other development bottlenecks. Consider the tedious business of renaming a function. Each dependent workspace component needs to be found and manually patched. With AVCS, the task of the programmer would be to confirm or reject suggested changes. Afterward, AVCS will automatically create new versions of updated components in the master project, making them available to the whole development team.[7]

6 Conclusions

The AVCS version control system is a tool to help with team development of software in distributed environments. Version control is widely utilized in the software industry. Although AVCS is developed in a PC environment, its approach can be accommodated on any other platform. We believe that the idea might be considered by APL vendors as a possible extension to the APL development environment.

7 Acknowledgments

Robert G. Brown made several suggestions which we used in designing AVCS.

Robert Bernecky helped us to obtain LOGOS manuals.

Reviewers' comments greatly improved the presentation.

We are especially grateful to Larry Schwink and Robert Bernecky for their editorial assistance.

8 References

[INTERSOLV 1993] PVCS for DOS and OS/2, PVCS Manager, Getting Started. INTERSOLV, Inc., 1993.

[Meyer 1988] Meyer, B. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[Microsoft 1993] Microsoft Delta, Version Control System for Windows, User's Guide. Microsoft Corporation, 1993.

[Mizel 1992] Mizel, J. "Using SCCS to manage APL2 development projects," *APL92 Conference Proceedings, APL Quote Quad*, vol. 23, no. 1, July 1992.

[Reuter 1990] LOGOS, User's Guide. Reuter:file Limited, 1990.

[Swain&Jonusz 1993] Swain, R., and D. Jonusz. "The Workspace Manager: A Change Control System for APL," *APL93 Conference Proceedings, APL Quote Quad*, vol. 24, no. 1, August 1993.

[7]LOGOS offers an advanced rename facility which involves analyzing the source code. We are also considering such a possibility for future versions of AVCS.