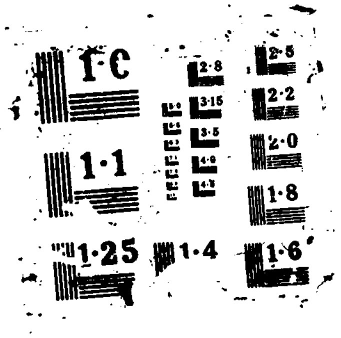
AD-A193 809 A FLEXIBLE BASIS FOR SOFTWARE CONFIGURATION MANAGEMENT 1/1
(U) ROYAL SIGNALS AND RADAR ESTABLISHMENT MALVERN
(ENGLAND) M STANLEY ET AL. FEB 88 RSRE-MEMO-4127
UNCLASSIFIED DRIC-BR-105560 F/G 12/5 NL

END
[illegible]


1·0
2·8
2·5
3·15
2·2
3·5
2·0
4·0
1·1
1·8
1·25
1·4
1·6

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Memorandum 4127

Title:
# A Flexible basis for Software Configuration Management

Authors:
M.Stanley, P.D.Hammond

Date:
February 1988

**Summary:**
Software configuration management is a necessary part of any project support environment. This paper discusses what software configuration management aims to achieve and proposes a basis from which a software c~~nfiguration management system can be developed. The proposed basis is illustrated by reference to the software configuration management features demonstrated in the Flex programming support environment, developed at RSRE Malvern.

# A Flexible basis for Software Configuration Management

M.Stanley and P.D.Hammond

**CONTENTS**

(Continued)

**A Flexible basis for Software Configuration Management**

M.Stanley and P.D.Hammond

**CONTENTS**

(Continued)

# A Flexible basis for Software Configuration Management

M.Stanley and P.D.Hammond

## 1. INTRODUCTION

Configuration management is defined (ref 6) to be management of change. It has two complementary aspects, provision of information about a changing product and controlling changes to the product. Change needs to be controlled and recorded without excessive interference in the development process.

A software product changes whenever any component (compiled unit and the source text from which the compiled unit derived) is changed, or the way in which the components are assembled is changed or any associated item (e.g. requirements, designs, user documents or build files) is changed. A component may be part of more than one product.

This paper discusses a basis for a flexible approach to software configuration management. It sets out the aims of software configuration management and outlines the traditional method of achieving these aims. It goes on to describe the underlying features that provide a basis for configuration management in the Flex programming support environment. (1,2,4,5,10 and 11). It describes the practical application of these features in easing some of the traditional problems of configuration management. Configuration management across a set of machines is considered as well as configuration management on a single machine.

## 2. AIMS OF SOFTWARE CONFIGURATION MANAGEMENT

### 2.1 Product identification

A software product may be instantiated in several places at once, perhaps on different computers. Software suppliers need to be able to identify the components and structure of an existing instantiation of a product (not necessarily the most up-to-date version) to enable them to explain its behaviour, to rectify detected faults and to update an out of date instantiation of the product. Unambiguous identification of each source text used to derive that instantiation of the product, the relationships between the components and identification of associated items such as documentation is essential. It may also be useful to identify the transformations applied in deriving the product, such as the version of the compiler and of the linking process.

### 2.2 Product stability

Change control involves both preventing unauthorised change and assisting authorised change. Prevention of unauthorised change requires access controls that allow use of a component while denying the ability to modify the component. Even authorised change may have limited applicability. A user may wish to prevent an authorised change to a component from being applied to a specific product instantiation. It may be easier to live with an error than to accommodate corrections, particularly corrections that involve changes to the interface between components.

### 2.3 Change evaluation

Any proposed change must be evaluated and its consequences identified. This may involve identifying users of a component to obtain their consent to a proposed change. It may also involve identifying associated components and documentation and tracing the history of a component, identifying date and reason for previous changes.

### 2.4 Testing authorised changes

Before a product is actually altered it must be possible to test an authorised change without impact on other users. Records must be kept of tests, test results and tests that must be repeated after changes have been made.

### 2.5 Propagating tested changes

On completion of testing assistance is required to propagate the corrections to all users of a changed component with a minimum of effort.

### 2.6 Controlling authorised change

When a change is authorised it is important that implementation of the change should not introduce inconsistency in any associated product. Inconsistency may be introduced between a product or component and its documentation or it may take the form of non-matching interfaces within a product. The configuration management system should provide whatever assistance it can to encourage consistency between a product and associated items such as documentation, although this must ultimately be a matter for human judgement. Non-matching interfaces within a product should be prevented. It should not be possible to run a program in which some interface does not match.

## 3. TRADITIONAL SOFTWARE CONFIGURATION MANAGEMENT

Traditionally software configuration management (ref 7) has involved the provision of a library of master copies of items which have been placed under formal change control, access to which is permitted only to a librarian. A new version of any item is regarded as a separate item with a relationship (new version) to the original item.

### 3.1 Product identification

Identification of the components in any product instantiation depends on accurate recording of the issue of components and products to users and on information on how products have been derived from other items, on which version of a component is used by whom and how the various versions of the components are combined to form different versions of the products (for example which source text was used for version 5 of a product). To this end the librarian is responsible for storing, controlling access to and providing copies of all items placed under his control. He is also expected to keep accurate records of the history of controlled items and the relationships between them (such as that two items are different versions of the same thing or that one item interfaces to, or uses, another). Some relationships (such as the relationship between source text and compiled code or the relationship between successive versions of an item) are traditionally indicated by a naming convention. Accurate recording and control of items by the supplier of a product may still be insufficient for completely unambiguous identification of the components in use in a particular instantiation if product maintenance depends, not only on the supplier, but also on trust that any modification issued to a user has actually been correctly implemented on the product.

## 3.2 Product stability

Items from the library are issued only after appropriate formalities have been completed, such as change approval by a designated authority. An authorised change is implemented on copies of the affected master items issued by the librarian and the tested change is entered into the library as a set of new items.

## 3.3 Change evaluation

Change requests are recorded by the librarian who provides status reports on change requests awaiting approval or implementation as well as information (where possible) on the implications of proposed changes, such as a list of other items which will be affected and a list of users. A librarian may or may not be supported by software tools and a database management system holding information about the controlled items. Most database management systems cannot hold the items themselves because they are not designed to hold files, only information about files.

## 3.4 Testing authorised changes

Changes to a component are implemented and tested on an issued copy of the controlled item. Only on completion of testing is the modified item added to the library of controlled items. The set of tests made and the results of the tests are also held in the library. These tests may be rerun when testing a modified item.

## 3.5 Propagating tested changes

When a tested item has been added to the library, the librarian is responsible for propagating the tested change to all affected users. Propagation of completed changes is often achieved by issuing update information to component users listed by the librarian. It is then the responsibility of the recipient to ensure that the change is correctly implemented. There is heavy reliance on manually updated information which is liable to error. Programmers do not always remember to record all relevant information on a changing product.

One of the difficulties in this approach to configuration management lies in the need to have separate copies of software in the library and in use in a product. The librarian is responsible for ensuring that changes to the master copies are reflected in all use of the items in actual programs. There is always the danger that the different instantiations may get out of step.

## 3.6 Controlling authorised change

The responsibility for maintaining consistency between controlled items lies with the librarian. Use of appropriate high level languages assists in enforcing consistency between compiled units in a product but the librarian still needs to identify associated changes resulting from modification of a widely used component. The librarian also has the responsibility for insisting on changes to associated documentation and for requiring proper authorisation before accepting a changed component into the library or propagating a change.

## 4. FLEX FEATURES THAT SUPPORT CONFIGURATION MANAGEMENT

A different approach to software configuration management is demonstrated in the Flex Programming Support Environment (PSE) developed at RSRE, Malvern. Flex is an interactive PSE that has been used for program development since 1978. This section identifies the features of Flex that are relevant to software configuration management and discusses their use as demonstrated in a local network of Flex machines connected on ethernet. One useful feature is the ability to show and to document relationships by using tree-like structures combined with text, on filestore. Another feature is the Flex separate compilation and software construction system which eases propagation of changes to affected users as well as providing binding relationships between a product and the source text from which it was derived. A third feature is the remarkably flexible and powerful access control facilities provided by combining the microcode supported capability mechanism with the treatment of the procedur [illegible] a "first-class" value.

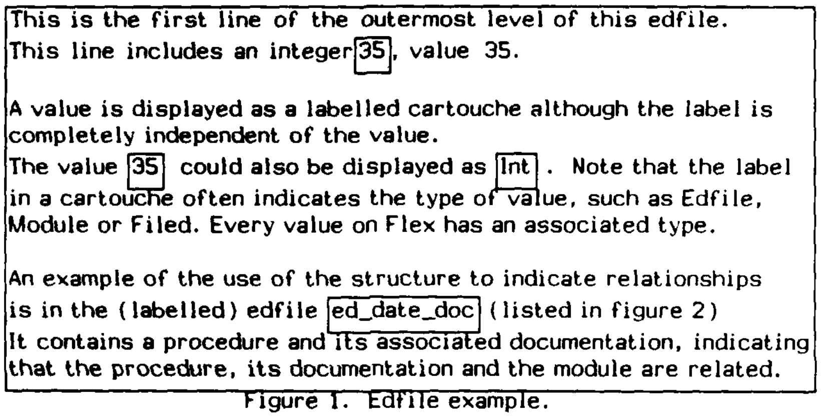
### 4.1 Relationships

#### 4.1.1 Database-like structures

An important aspect of configuration management is the recording of and tracing of relationships between different controlled items.

Flex supports the general expression of relationships in a database-like structure of text and filestore values. The database_like structures are stored on filestore and are called edfiles. The values and associated text held within them are created, updated and searched by the normal Flex editor. This editor is unusual in that it is fully integrated with the command interpreter and handles not just text but a mixture of text and values. It handles any value that can exist on Flex such as other edfiles, compiled units (called modules), integers, programs (called filed procedures) etc. Since any edfile may contain others, edfiles may be a structure of edfiles and other values, with the restriction (enforced by the Flex filestore) that the structure must be acyclic. The values are produced and used by the integrated command interpreter. Many database management systems are restricted in the kinds of data that they can hold. They can hold only textual information about the values such as programs or source text files. They cannot hold the values themselves. The edfile structures are more powerful than these for configuration management because they can hold the actual programs and source text files that are to be controlled.

Figure 1 is a trivial example of the content of a structured edfile as displayed by the Flex editor. The cartouches (boxes) represent values. Anything not enclosed in a cartouche is normal text.

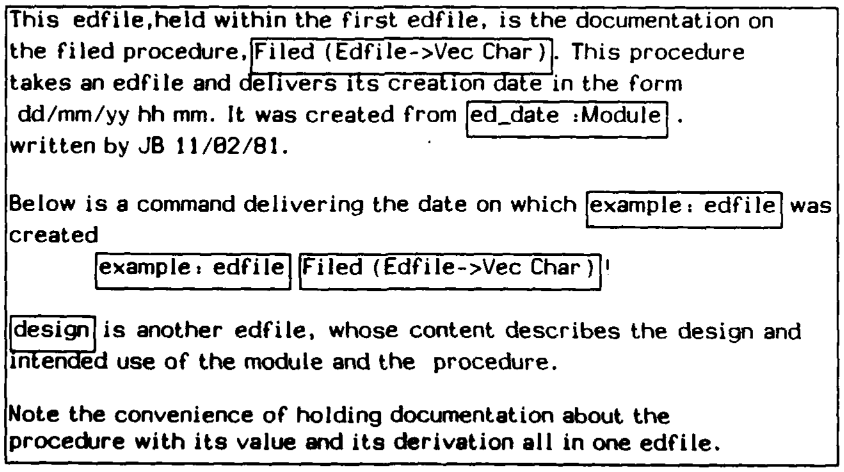


This is the first line of the outermost level of this edfile.
This line includes an integer $\boxed{\text{35}}$, value 35.

A value is displayed as a labelled cartouche although the label is completely independent of the value.
The value $\boxed{\text{35}}$ could also be displayed as $\boxed{\text{Int}}$ . Note that the label in a cartouche often indicates the type of value, such as Edfile, Module or Filed. Every value on Flex has an associated type.

An example of the use of the structure to indicate relationships is in the (labelled) edfile $\boxed{\text{ed\_date\_doc}}$ (listed in figure 2)
It contains a procedure and its associated documentation, indicating that the procedure, its documentation and the module are related.

Figure 1. Edfile example.



This edfile,held within the first edfile, is the documentation on the filed procedure, $\boxed{\text{Filed (Edfile->Vec Char)}}$. This procedure takes an edfile and delivers its creation date in the form dd/mm/yy hh mm. It was created from $\boxed{\text{ed\_date :Module}}$ .
written by JB 11/02/81.

Below is a command delivering the date on which $\boxed{\text{example: edfile}}$ was created

$\boxed{\text{example: edfile}}$ $\boxed{\text{Filed (Edfile->Vec Char)}}$ !

$\boxed{\text{design}}$ is another edfile, whose content describes the design and intended use of the module and the procedure.

Note the convenience of holding documentation about the procedure with its value and its derivation all in one edfile.

Figure 2. Edfile labelled ed_date_doc.

The structure shown in figures 1 and 2 is tree-like. It is not strictly speaking a tree because the same value may appear in more than one place in the acyclic structure. The value of objects such as edfiles and modules are represented by pointers (capabilities, see later) that give access to the object content, so every occurrence of a given value in the tree-like structures gives access to the same object. Only a single copy of the object exists so there need be no concern about keeping different copies in step with each other.

The editor provides facilities for examining the value represented by a cartouche. Every value in Flex has an associated type (ref 12) that defines its structure and the operations that are valid on it. The type is used by the editor to select how the examined value will be displayed. For example examining a value that is an edfile displays its content and examining a value that is a module displays the source text from which it was derived (see later).

An edfile is rather like a combination of a text file and the tree-structured directories available on many other systems, although it is more flexible for expressing relationships than a tree-structured directory. There are two reasons for this. Firstly the ability to hold explanatory text within the structure allows the reliance on navigation and correct use of naming conventions to be abandoned. Meaning can be documented fully in the surrounding text rather than relying on the somewhat inadequate shorthand of appropriately chosen names. Secondly, the same value can appear in any number of places in the structure without extra copies of the object to which it gives access being created. Relationships are expressed by holding related values either adjacent or at different levels in the same edfile. Multiple relationships may be expressed by holding a value in more than one place. For example, the relationships between values such as modules, programs, design documents, user guides etc are shown by holding the related values, with explanatory text, either in the same edfile or in subfiles of an outer level edfile as illustrated in the edfile example, ed_date_doc.

### 4.1.2. Relationships and History

As a software product develops controlled items are changed or superseded. One responsibility of configuration management is keeping a history of the development, retaining copies of superseded items and the relationships between them.

#### 4.1.2.1 Constants

In general Flex filestore values are constants and there is therefore no danger that a change will alter an old version. In particular, edfiles are constants. A new version of an edfile is a totally new value created after modifying a mainstore copy of the edfile using the editor. The old value is unaltered and persists, keeping all the implicit relationship information and old values held in the edfile, as long as a pointer to it (capability for it) is retained. On a system that permits filestore values to be modifed there is a danger that an old version of a file might actually be destroyed when the new version is created.

The Flex architecture ensures that filestore values are normally constants. The filestore is basically non-overwriting. A user can either write something to filestore or he can read or execute something that has already been written. The only operation that can overwrite anything on filestore is an atomic action to update a root pointer to point to a new root value. The fact that only atomic actions can overwrite the filestore prevents the possibility of partially updating filestore. Vital files could be lost and filestore corrputed if a hardware or system crash during file updating could result in partially updated filestore.

**4.1.2.2 Modifiable values**
There are values on Flex which are not constants. They are references to some content. When updated, the reference gives access to new content. Modifiable values do not violate the principle of non-overwriting. Each modifiable value is accessed through a root value which is a constant. Modification is achieved by creating a new constant root value and updating the root pointer to point to the new root value. The old root value, previously pointed to by the root pointer, can be accessed through the new root value. It holds the old content of the modifiable value. Alternatively the old content of a modifiable value can be saved before the value is updated (see discussion on versions). Modules are modifiable values and there are also modifiable edfiles, called edfile-modules.

**4.1.3 Special relationships**
In addition to flexible expression of relationships using the edfile structures there are some special relationships which are relevant to configuration management. The special relationships on Flex are binding relationships which are automatically created and maintained because they have been identified as useful in program development. They are analogous to the relationship between an entity and its attributes in a database such as the entity relationship style of database supported by PCTE (Portable Common Tool Environment) and CAIS (Common APSE Interface set) (refs 15, 16, 17 and 18). These special relationships are expressed on Flex by making the related values accessible through a single object. The related values and the association between them are protected from unauthorised tampering by hiding the object in a procedure or program (see later). Other special relationships could be added using the same technique.

The binding relationship between any value and its type is fundamental to Flex. Otherspecial relationships currently supported on Flex include a relationship between source text, compiled code and log file which are bound together in a single object called a module (discussed later). The relationship between a named value, an information file (an edfile which may be structured) holding information about the value and the date of the association between name and value is expressed by holding the three values together in an object called a Flex dictionary that can be accessed only by calling the appropriate dictionary access programs. (A dictionary is a set of name/value associations associated with a specific user.) The relationship binding an edfile (including source text accessed from a module) to its creation date is maintained by having the creation date as an integral part of the edfile. (Update dates are not relevant on Flex because edfiles are constants.)

**4.1.4 Relationships between source text, compiled code and program**
One problem in product identification is keeping track of the source text from which any software product was derived. The traditional methods of using carefully chosen names has proved difficult to manage. Databases showing relationships between source text, compiled code and product, have been suggested (ref 19).

Flex uses a slightly different approach. Firstly the unit of separate compilation (a module) actually gives access both to source text and to compiled code. The two are indissolubly linked. The usual practice is to rely on this binding to keep the source text. Explicit references to the source text can be deleted as soon as a module has been created. A program or module can be issued to other users with or without access to the source text.

Secondly the source text of Algol68 programs (the programming language most commonly used on Flex) itself contains the actual value of each separately compiled module that it uses. This is possible because both the editor and Flex Algol68 compiler are designed to handle values in the source text rather than referring by name to a compiled unit held in a separate library. The multiple 'used-by' relationship is expressed by the fact that the module value is held in every source text that uses it. This does not imply that each source text holding a module value requires its own copy of the module. A module is a modifiable value and each instance of a given module value accesses the same content, the current version of the source text, compiled code and a log file showing the history of the module. (Log files and versioning are both discussed in more detail later.) Since a source text holds the values of every used module, the user can extract not only the source text from which a module derived but also, by chaining through the used modules, any of the source texts associated with used modules. The user can always be confident that the text extracted from the module is the correct version. There is never any confusion and the programmer is not burdened with the responsibility for keeping appropriate records. Note that a module can be made accessible to other users with the source text hidden. This may be deemed desirable for software that is commercially sensitive.

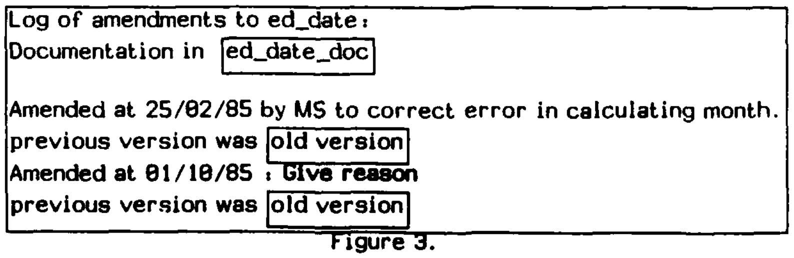
Separate build files are unnecessary because each module always gives access to all the modules that it uses. There is no ambiguity as to the version used because a module is a modifiable value. It is not a sequence of values. A user can also discover from which of the modules to which he already has access a program was derived. However, to prevent a user from gaining unauthorised access to the modules, it is possible only to confirm the derivation. The user must already have access to the module.

When a program has failed, any user can display the source text of the modules from which the failed program derived (provided the text has not been hidden by the supplier, as mentioned above). This is not equivalent to giving access to the module. The text can be accessed but not the module from which the product derived.

It is sometimes considered desirable that not only should the source text of a product be identifiable but the version of the compiler and other programs used to convert source text to an executable program should also be identifiable. Although this is difficult on Flex it could be arranged. At present the date of creation (date of last change) of the source file in each module is recorded but the date of compilation is not recorded. The date of module amendment (which will usually but not necessarily be the date of compilation) is recorded in the log file. Were the date of compilation to be recorded it would be possible, although somewhat tedious, to reconstruct the version of the compiler in use on that date. This would be done by examining every module in the compiler to discover whether it had changed since the date in question. So far the effort required has not seemed justified.

**4.1.5 Relationships to associated items**
Source text and compiled code are not the only elements in a module. A log file that holds records of changes to the module is also bound to the module. It is a structured edfile that can be accessed only through the module. It may hold not only the history of a module, the changes that have been made and the reasons for them, but also the superseded versions of source text, relevant change requests and authorisation and any supporting documentation. Other more permanant information may also be included such as the name and address of the author of the module. It would also be possible to include items that might be affected should the module be changed, such as the design and user documentation.

The log file is automatically updated when a module is changed. The module change programs display the log file, adding the date of the update and the value of the superseded version. They invite the amender to add a comment explaining the change. The amendment is committed only when the user indicates that he has finished updating the log. Figure 3 shows the log file displayed when updating the *ed_date* module shown in figure 2.


```
Log of amendments to ed_date:
Documentation in  ed_date_doc

Amended at 25/02/85 by MS to correct error in calculating month.
previous version was old version
Amended at 01/10/85 : Give reason
previous version was old version
```
Figure 3.

## 4.2 Access control and product stability

Access control is vital to product stability. Items under configuration control should be accessible to users for appropriate use but not for alteration. Two aspects of Flex contribute to access control, capabilities and procedure values.

### 4.2.1 Capabilities

Those values in Flex which require some degree of access control to preserve system integrity are represented by capabilities (refs 13 and 14). This includes all values that would normally be under change control. The capability mechanism ensures that only authorised operations (read/write or execute) can be applied to the value represented by a capability and only the holder of the capability can apply the authorised operations. An important and unusual aspect of Flex capabilities is that they can be treated like other values in that they can be held anywhere on filestore or in mainstore, stored in edfiles or used in programs and can be passed to another user, giving the other user the right to access the controlled value but only in the specified way.

The Flex microcode (which is fixed and inaccessible to users, being used as an extension to the hardware) ensures that capabilities cannot be forged. The only software or user operations involving a capability are:

1. request a new capability from the microcode (any procedure or program can do this; it is not a privileged operation);
2. store a capability for future use;
3. copy the capability to another user;
4. delete a copy of the capability (this does not delete the value to which the capability gives access, another copy of the capability may still exist);
5. use the capability as authorised by the microcode.

Current work at RSRE aims to implement capabilities and the protection that they provide without specific microcode support.

Items under configuration management can be issued freely as capabilities, secure in the knowledge that they can be used only as authorised. Since capabilities cannot be modified, the type of access permission that they allow cannot be changed. A user with a capability to execute a program can execute the program but he cannot do anything else to it, such as reading any internal values or modifying any code. Since capabilities cannot be forged illegal access to the controlled values cannot be obtained.

Capabilities can be split into four general categories :-

1. Mainstore capabilities - exist in only one mainstore and cannot be transmitted elsewhere.
2. Filestore capabilities - exist in only one filestore or in the mainstore of any computer which directly accesses that

filestore. They can be transmitted between mainstore and the filestore.
3. Remote capabilities - can exist in any mainstore and can be transmitted between mainstores. They allow access to other mainstores and filestores across the network.
4. Universal capabilities - can exist anywhere in the Flex world in mainstores, filestores or on networks. They represent commonly used objects like compilers, editors etc.

Items under configuration management are usually filestore capabilities that do not permit alteration of the controlled value. Remote capabilities may be used to give remote access to controlled filestore values.

### 4.2.2 Procedure values

A major factor in the access controls provided on Flex is the fact that Flex treats procedures as first class, context independent values or objects (refs 2,8 and 9). This allows values, including capabilities, to be protected by being hidden inside a procedure. The only way to access such values is to execute the procedure (or program) that hides them. Such programs are known as access programs.

A true procedure value in the sense of Landin (ref 7) has the context or environment in which the procedure runs (its non-locals) bound in as a part of the procedure value. Procedures may be parameters for other procedures or may be delivered by other procedures (provided the language also supports this notion, as does Algol68). Procedure values may be executed either by calling from other procedures or as programs called from the command interpreter. The terms program and procedure are therefore used interchangeably. Values (both local and non-local) and input parameters of the delivering procedure can be bound into a delivered program, and thus hidden from a caller of the delivered program. The facility to hide values in delivered access programs is not privileged but is available to any programmer. It is used to provide the flexible access control necessary for configuration management.

For example, delivered procedures might be used as described below. It should be borne in mind that this is only an example of how the access controls provided by procedure values might be applied. There is complete flexibility as to how the facilities are used. Precise details of the access programs suitable for a full configuration management system have not yet been worked out, although the practical application of access programs is demonstrated in their use to control changes to modules, as discussed later.

Consider a program to place an item under configuration management. The program, *control_item*, takes as parameter a capability for the item to be protected. It creates a history record and an old versions record for that item and delivers a set of four programs for accessing and updating the item and its history. The protected item and its history will then be accessible only through the issued access programs, provided that the programmer who provided the item for configuration control does not retain an unauthorised copy of the capability.

Because Algol68 modes do not include capabilities as a distinct mode, capabilities are represented in Algol68 by integers. (Microcode checks prevent the abuse of a capability represented as an integer.) Program *control_item* therefore has Algol68 mode:

```
PROC control_item = (INT file_capability)
                       STRUCT(PROC INT issue_item,
                              PROC (INT old_item,
                                INT new_item,
                                INT change_request )update_item ,
                              PROC INT show_history ,
                              PROC (INT version)INT old_version);
```

The first delivered program (*issue_item*) takes no parameters and delivers a capability for the controlled item only after it has performed certain checks. These checks might include password checks and a set of questions requesting details of the authorisation to access the item. They might request a capability for the relevant change request. Any checks that the writer of *control_item* wishes to impose can be built into the delivered access program, *issue_item*. Each time *issue_item* is executed it either fails after recording an unsuccessful attempt to satisfy the checks or it records, in the history, the identity of the recipient, the reason for issue and the date and the capability for the authorised change request, before issuing the item. The recorded information is accessible only through another delivered program, *show_history*.

The second delivered program (*update_item*) takes as parameters the capabilities for the issued item, its replacement and the change request against which the item was issued. It checks that the change request and issued item are as recorded in the history, performs any specified compatibility checks between the old and new items, and if all checks are satisfied it replaces the old version by the new one, moving the superseded item to the old_versions record and updating the history record.

The third delivered program (*show_history*) displays the history of the item and delivers a capability to read the history, again after appropriate checks and the fourth, (*old_version*) after appropriate checks will issue an old version.

A user invokes *control_item* to place an item under configuration management. The delivered access programs all give access to the same item. They all have the capabilities for the protected item, its history and the old versions hidden within them as non-local values, bound to the delivered programs when *control_item* is executed. These capabilities need not exist outside these programs. The access programs may be stored on backing store because all the internal values and non-locals can be stored on backing store. The user can use them to access the item and its history and to update the item. He can pass them to other users to give them controlled access to the item and its history. Different programs may be issued to different users, allowing some access only to the history while others have access to *issue_item* and *update_item*. The controlled capability can be shared safely by issuing the capabilities for the access programs. The protected value cannot be reached except by executing an access program because it does not exist outside these programs. The capability to execute an access program does not give the ability to dismember the program to get at the protected value. The access control provided by the capability mechanism combines with the protection provided by procedure values to give an unusually powerful and flexible form of access control that can be used, not only by the operating system, but also by any programmer, to protect values and to provide configuration control. Access programs are ordinary procedures created by a user. No privilege is required.

### 4.2.3 Protection from unauthorised change

The protection of modules from change by an unauthorised user provides a good example of the use of delivered procedures. Each user has a set of procedures or programs peculiar to him, which have his identity bound in. It is impossible to supply the identity of a different user to them. Included in the set of user specific programs are a program *new_module* to create a new module and a pair of programs that can modify only modules created by that *new_module* program. One of these programs *change_spec* permits the external specification of the module to be changed, the other *amend_module* will apply only a change that preserves the external specification. When a change program is invoked on a module, the identity of the owner of the module, bound into the module by the *new_module* program that created it, is compared with the identity bound into the change program. Only if the two user identities match will the module be updated. This ensures that a module can normally be changed only by its owner. Attempts to change a module created by someone elses *new_module* will fail. The owner of a module can still modify a module that has been issued to other users because he holds the change program with the correct user identity. If it is deemed desirable to limit the authority to change an issued module to a librarian this could be done by requiring the librarian to issue only modules created by his *new_module* program. It is easy to create a new module from any existing module submitted to the librarian.

More elaborate configuration management controls could be incorporated in the change programs. They could require confirmation that all associated items (listed in the log file) had been changed, perhaps requesting some form of independent authorisation to confirm consistency before a change is accepted for propagation.

## 4.3 Change evaluation

Evaluating a proposed change may involve considering all items that might be affected by the change, such as documentation, associated products and modules that use components that will change. It is important to consider the problem of maintaining consistency between related items. Consistency between documentation and the rest of a product is a matter of human judgement. However, a configuration management system should provide such assistance as is possible, for example indicating where a change in one item implies a need for a corresponding change in an associated item (e.g. if a compiled unit is changed, the documentation or user guide may need change). If all items likely to be affected by changes to modifiable values such as a module or an edfile module were held in the associated log file, then they would easily be located. The source text of the module gives access all used modules. It is therefore easy to list the modules used in any given module and to access the log files of the used modules to find other associated items.

It is not essential to maintain accurate lists of all users of a module because changes are automatically propagated and interfaces are automatically checked, as discussed below. Provided the external specification and the functionality of a module are not affected by a proposed change then it is probably not necessary to contact all users. Such a change would be acceptable even without prior warning. More care is needed if a change to the external interface of the module or to its functionality is envisaged. It is a human responsibility to consider carefully the consequences of a radical change. It may be preferable to replace the module with a different module in specific products rather than to update the module for all users.

The evaluator of a change that will affect either the functionality or the external specification of a module may also wish to know who uses a module that may be changed. Records are not usually kept on Flex as to where a module is used. Locating all users of a module is difficult because the filestore is acyclic, so there is no explicit route from a value to the values that contain it. It is, however, easy to confirm whether a user has a specific module. Programs exist to search a container (such as an edfile, a dictionary or a module) to confirm whether it gives access to a given module and to search for a module that uses a given module.

## 4.4 Implementing change

### 4.4.1 Interface checking

One task of configuration management is to assure product users that the interfaces between components will always match. A product running with non-matching interfaces will have unpredictable results. Non-matching interfaces within a product are a hazard not only when a product is initially assembled from its components but throughout the life of the product. Corrections to components can give rise to non-matching interfaces so it is necessary to be able to detect a non-matching interface and to correct affected components. Choice of language and of separate compilation facilities affects the clarity of the interface definition and the ability to ensure matching interfaces between components at all times. A language such as Algol68 or Ada that enforces adequate type checking of interfaces at compilation time is an important element in configuration control, particularly where re-use of components is involved. Compilers on Flex check that the interfaces of all used modules match their use.

In addition Flex differentiates between changes that alter an external specification and amendments that leave the external interface inviolate. The amendment program does not permit alteration of the external specification. Because different programs are used, an external specification cannot be altered unwittingly. To prevent the use of modules whose interfaces no longer match their use, instances of a module updated using a *change_spec* program are marked as invalid. They can be revalidated only by recompiling the using module. (It is not necessary to recompile any user of a module whose external specification has not changed.) Invalid instances of modules cannot be loaded or run. Thus any user, even if not formally notified of the change, will know that an interface has been changed when he tries to use the program or procedure involving an invalid instance of the changed module. He will never be subject to the undefined results that can occur if accidentally using a procedure whose specification no longer matches the call.

It is easy to implement all the changes needed to accommodate interface changes. A *recompile* program scans a module and all used modules, locates any use of an invalid instance of a module and recompiles the user, validating the used module. Where non-matching interfaces are involved it calls the editor to invite change to the using module.

Modifying a using module will fail if the using module was created by another user because a module can be updated only by its creator. For this reason approval for changes that will alter an external specification of a widely used module should be granted only in exceptional circumstances.

### 4.4.2 Change propagation

A common problem in software configuration management is to ensure that, when an error has been corrected in a component, the correction is propagated to all users. Some systems provide automatic propagation of changes in compiled units to all dependent units but the propagation is not always extended to cover programs or executable images that include the changed units. Manual propagation of changes, aided by tools such as a program to list where a compiled unit has been used, may depend on a programmer remembering to insert the necessary information in a database. The administrative burden of keeping track of changes can be reduced by such tools, but reliance on systematic use of the naming system and on separate systematic recording of all changes and all uses of a compiled unit is dangerous and error prone.

Changes to Flex modules have immediate system wide effect. This is unusual. It means that a change is automatically propagated to all users of the module, to all programs or filed procedures that involve the module and across all Ada libraries that contain the module. Changes are so propagated because Flex links its programs dynamically. The user of the module (be it another module or a program) does not incorporate the code of a module until it is loaded at run time. There are no executable images with distinct copies of the compiled code bound to them. In addition a module is effectively a reference to its content. When a module is updated the same module capability references the new content. There is only one copy of the compiled code of any module regardless of the number of references to it held in different using modules. The module capability gives access to the same content whether issued to another user or not.

Sometimes a user may wish to prevent changes being propagated to his use of a module that was supplied by someone else. Flex has facilities for this. The module user can call his own *new_module* program to create a new module from the old one. The new module will not receive any of the changes propagated to the module from which it was derived. Obviously to prevent any changes to anything would require creating a whole new set of modules. However, in practice this rarely seems necessary. The programmer can still hold the module that he has replaced (perhaps in the log file of the replacing module) thus having access to the log file and corresponding changes in the replaced module in case they prove to be of interest. It will be the responsibility of the owner of the new module to check occasionally whether any significant change has been made to the replaced module.

### 4.4.3 Testing an authorised change

Any proposed change can be tested and approved prior to committing the change to all users. Although a program is usually created from a module it can be derived directly from the compiled code without affecting the module. The directly derived program can be tested by calling it from the command interpreter without changing any module. Alternatively, because programs are linked dynamically, a new program can be created for testing purposes by replacing selected modules used within a program. The replacements are linked into the program instead of the original modules. Neither the replaced modules nor the original program is affected and no change is made to any source text. This powerful facility can be used to test proposed changes before propagation of the change or to construct multiple versions of a piece of software by merely replacing certain modules. The remainder of the modules will be common to the different programs. Module replacement is possible only if the replacement has the same external interface as the replaced module. Replacement of modules with altered external specification is achieved by replacing the using module, altered as necessary to match the altered used module specification.

**4.4.4 Propagating changes in a distributed system**
In a distributed system or in situations where separate machines have their own copy of the software, changes must be propagated to other machines. If Flex machines are connected on a network then software on one machine may invoke software residing on another machine on the network using remote capabilities. The configuration management facilities described above apply both to software on a single Flex machine and to use of software shared using remote capabilities. In the situation where separate machines each have their own copy of the software is is necessary to be able to keep the copies in step, or at least to know when copies differ. Conventionally this is done by keeping records of updates and of transfers of updates to the different machines, usually keeping a master copy of any shared software in a library from which updates may be taken. Transfers may be of source text, of compiled units or executable programs or of amendment instructions.

At present any Flex software may be copied from a host machine to other (target) machines. Users of the target sofware periodically update it from the host. A program, *update,* is run on the host machine. *Update* automatically extracts the source texts from all modules in a given set which have been changed since a given date. These are transferred to the target machine where another program is run to update the target software. This recompiles the transferred texts and amends the modules on the target machine. It creates new modules to cater for those modules created in the host since the given date (probably the last update date). In some cases this process can be speeded up by transferring the compiled code of the module to avoid recompilation on the target machine. Changing the modules on the target automatically results in changes to the programs derived from them, as described above. These facilities allow software to be taken from any host to any target. While they do not rely on manual recording of who has what software nor when host software has been changed, the control is not as rigorous as that which can be applied on a single Flex machine. It relies on correct use of *update* and on records to indicate which hosts have supplied software to a target. Each individual machine is however protected to a certain extent by its own internal configuration management.

A higher level of distributed configuration management will be achieved by the use of universal capabilities (ref 14) and an ethernet linking all the Flex machines. Work on universal capabilities is currently under way. A universal capability represents a system value which has a counterpart on each machine e.g. the Algol68 compiler. On the network this value will be represented by a simple unique token. On receiving this token a machine will know that it should use its own copy of that universal capability. Each machine will also keep the version number of its copy of any universal capability and will transmit this with the universal capability across the network. On receiving a universal capability with a higher version number than that of its own copy a machine will set in motion an update of its copy of that universal.

## 4.5 Versions

One responsibility of configuration management is maintaining an accurate record of the different versions of controlled items and the relationships between them. Versioning is a partial ordering, in which any object is superseded by its successor, but parallel versions may also exist that do not affect the currency of other versions.

### 4.5.1 Names

Conventionally a sequence of versions of an item share a common name, optionally qualified by a version identifier. Flex does not have this concept (ref 20). A Flex name in a given dictionary references a unique value, (which may be a structured edfile containing many other values) rather than a sequence of values. When a new value is associated with a name a new dictionary is created. The old dictionary, holding the value formerly associated with the name can be accessed through the new dictionary. A user who wishes to retain a sequence of values may keep them all in a single edfile, with suitable text or labels to indicate status, as is done automatically in the log files for modules (fig 3).

### 4.5.2 Old versions

As mentioned earlier, most values on Flex machines are constants and constant values cannot be altered, although they can be superseded. Replacement of constants with new values can make it difficult to maintain consistency if the same constant appears in several different places (for example the same edfile may appear in different branches of a structured edfile). One instance may be superseded by a new version but the other instances may still point to the old version. It such cases modifiable values are used.

Modules are not constants, they are modifiable values. Instances of a module may appear in many places and changes to it are automatically propagated to all instances. A modifiable edfile (called an edfile-module) is another example of a modifiable value, created using the module mechanism. An edfile-module is like a module whose content contains the edfile but no compiled code. It is updated in the same way as a module, and superseded versions of the edfile and its history can be stored in the log file if required. This is particularly useful because it allows references to structured edfiles. Edfile-modules are used for those edfiles which need to be referenced from more than one place, for example to allow cross-referencing in documentation.

Old versions of constants can be accessed as long as a capability for them is retained. Successive versions of constants may be held together in a single structured edfile. Old versions of the content of an modifiable value can be written automatically to the log file when the value is changed. Alternatively old values can be accessed through old dictionaries which can themselves be accessed through the current dictionary.

**4.5.3 Archives**
Traditionally the configuration management librarian has responsibility for archiving controlled items to ensure that old versions can be recovered or recreated and that, when recovered, consistency between related items is maintained. The user of a conventional system cannot expect to recover a precise copy of a prematurely discarded file. He may retrieve an earlier version by use of the system back-up facilities, but the earlier version (if accessible) may not be the same as the discarded file and may be inconsistent with other files or dictionaries in the current filestore. On Flex a discarded value previously associated with a module, an edfile-module or a name may be retrieved through old dictionaries, which are automatically retained on Flex until the next disc garbage collection. A value held in a previously named value can be retrieved through the value formerly associated with the name. The old version, recovered by access to the old dictionaries, is precisely the value that was lost and not some arbitrarily selected earlier version. Because the root pointers are the only values on Flex that can be over-written, each root pointer gives access (indirectly) to a completely consistent snapshot of filestore. This eases the problem of maintaining consistency when old versions are retrieved. A user worried about losing access to old versions at garbage collection can arrange to keep old versions automatically, as in the log file.

**4.5.4 Parallel versions**
Parallel versions of a program differing from each other in only a few modules are easy to create and to keep in step on Flex. For example, a compiler may exist with two back ends A and B, for two different target machines. The two versions exist in parallel and are composed from the same modules except for those peculiar to the specific target. Conventionally separate executable images for the parallel versions are created using separate build files. The only indication that both programs are essentially the same is in the manually maintained records held about them. On Flex this artificial separation is unnecessary. Version A of the compiler is created in the usual way from its modules. Version B is then created from version A by replacing only the target specific modules with different, equivalent compiled units for target B. Replacement of modules in an existing program ensures that that the differences are only those specifically indicated by the replacements. No accidental differences can creep in due to different build files or different modules being used in error or due to changes in shared modules not being incorporated in both versions. The command showing the replacement of modules from the parent program is kept with other relationship information in an edfile.

Corrections to modules that have not been replaced will be propagated to both versions in the usual way. The propagation of changes to the module replacements is not as neat as the propagation of changes to modules, and a balance needs to be struck to decide when a new module would be more appropriate than a replacement. If, however, a replacement is used in only one modified program, as usually happens, then the inability to propagate changes automatically to other units is irrelevant.

## 5. SUMMARY

### 5.1 Product identification

The identity of the source text from which a product instantiation derived is never in doubt on Flex. This certainty derives from a combination of binding source text to compiled code in a module and the use of actual values of the modules within the source text of the using modules. It removes the potential confusion of working with the wrong version of source text and enables the diagnostic program to deliver the actual text in which an error has been detected.

It has not in practice been found necessary to reconstruct the version of the compiler or linker used to derive a product from its source text. Although difficult, this would be possible.

### 5.2 Product stability

Procedure values provide the means to protect a module from unauthorised change. The actual protection provided prevents change by an unauthorised user, prevents unwitting change of external specification and requests an update to the log file containing a history of changes. Other protection could be incorporated by placing additional constraints such as password protection or a set of queries to be answered within the program that amends a module. Similar protection could be provided for other values if deemed useful.

A user who wishes to avoid all future change to his version of a module must create his own version of the module which will then be completely independent of its parent. He can still gain information about changes to the parent by keeping the parent and calling a program to display its log file, but he will not automatically receive notification of changes.

### 5.3 Change evaluation

The log file of a modifiable value such as a module or an edfile module, accessed through the value itself, need not be restricted to holding the history of the value. The use of structured edfiles to hold related items together with their documentation helps those assessing the impact of a proposed change to identify associated items such as documentation that may need change. The log file may be just such a source of information. The modules used by a module are easily identified by examining the source text extracted from the module. Although no records are kept on Flex as to where a value such as a module is used, procedures can be written to search a dictionary or container for a specific value if the information is needed. Should it prove necessary to discover where an item is used each user can search his own dictionaries to check if they give access to the item. This cumbersome task has not yet proved necessary in the ten years of use of the Flex system.

A radical change to a module, that would alter either its functionality or its external interface should be authorised only with extreme caution. Users of modules will reasonably expect stability in the external appearance of the module. It is incumbent upon those evaluating a proposed change not to authorise radical change. Any change radical enough to involve a change to the external interface of a module could result in non-matching interfaces. Although users of the modified module would automatically discover that the change had occurred when the product with non-matching interfaces was loaded, notification at run-time is obviously unsatisfactory. It is better to reject such radical changes, insisting on a new module in place of the previous version rather than the same module with a changed specification. The propagation of change would then be limited to those products where it is known that the change is needed.

The relationships between compiled code, source text and log file have proved invaluable for configuration management. If other explicit relationships are deemed useful for change evaluation they could be added to Flex using a data structure similar to a module in which related capabilities would be accessible through the same object.

## 5.4 Testing authorised changes

Testing an authorised change without actually modifying any component is done by running a program with selected modules replaced just for the purposes of the test or by creating a program directly from the compiled code and calling it from the command interpreter.

Test data, results and any textual explanation can if desired be kept in the associated log file or with the module in any other structured edfile. Change requests and authorisations may be also held in these edfiles. A procedure could be written to check that all the items identified as affected by an authorised change request had actually been modified before the change request status was marked completed.

## 5.5 Propagating tested changes

On completion of testing a module must actually be updated. Propagation of completed changes is automatic on a single Flex machine because linking of a module to a product occurs only when the product is loaded. Any change to the product then comes into force automatically. Changes must still be propagated explicitly to other machines.

## 5.6 Controlling authorised change

Module amendment will change an external specification only if this is explicitly requested, and will then invalidate all other instances of the changed module on that machine. Revalidatation involves calling the compilers on each using module. The compilers will check against the use of modules with non-matching interfaces. Automatic means are provided to locate instances of modules with changed external specification and to recompile only as necessary.

Although human intervention is still the most important factor in maintaining consistency between a component and associated documentation, the use of structured edfiles can assist by keeping associated items together. If desired all associated items can be held in the log file. This does not waste space since it is only the capability for an item that is actually kept in several places. The item itself occurs only once.

## 6. CONCLUSIONS

We have shown that an alternative to the traditional approach to software configuration management is possible. Although the actual procedures for a full software configuration management system have not been worked out, the facilities described here, and demonstrated in the Flex PSE, provide a powerful and adaptable basis on which to build a software configuration management system. These facilities support a form of software configuration management that allows for greater flexibility than can be permitted when relying on a library of master copies coupled with manually maintained records. The facilities can prevent related items from getting out of step, as demonstrated in the binding of compiled code to source text and log files, they can enforce proper use of controlled items, as demonstrated in the procedures controlling change to modules and they can express relationships in a particularly flexible way in the structured edfiles. The main characteristics that enable this support to be provided are the distinction between constants and modifiable values, capabilities, context independent procedure values, dynamic linking, the module system and an editor that handles values as well as text, thus supporting the database-like edfiles.

Additional tools for configuration management of large software projects could be envisaged which make use of the powerful facilities demonstrated. Some tools such as a file comparison program already exist and an equivalent of the source code control system (Glasser (8) and Rochkind(9)) system could be developed.

Procedure values combine with capabilities to provide a remarkably flexible and powerful system for access control. Although we have not investigated the control facilities needed for full configuration management, some indication of the power of access procedures to provide the necessary controls is demonstrated by the change controls provided for modules.

An editor that handles any type of value as well as text leads to the tree-like structured edfiles that make it easy to express relationships, to hold related objects together and to document values in the edfile that holds them. Instances of the same value can appear in any number of places in the structures without the value of the item to which the instance gives access actually being duplicated.

The module is a structure giving access to source text, compiled code and log file. The fact that these are linked in a single item gives confidence that they cannot get out of step. Similar structures may be desirable to support other aspects of configuration managment. The supplier of a module (and its user if permitted) will always know what source text was used to create the current version of a module and all the modules it uses. Use of structured log files, accessible only through the item whose history is being recorded, gives an unusual approach to the retention not only of old versions but also to the location of associated items.

Although most values on a system with non-overwriting filestore are constants it is possible to support modifiable values as is shown with modules. Non-overwriting filestore, designed to provide high integrity, gives confidence that values will not change. Old versions only need protection from being discarded. They cannot be corrupted. Although the non-overwriting filestore leads to difficulty in establishing who has access to a value held in several parts of an acyclic structure, this has not so far proved to be an important problem. Individual parts of the filestore can always be searched for a specific value when necessary.

Dynamic linking of procedures at run time plus the fact that modules are modifiable values with instances of the used modules actually held in the source text gives automatic propagation of completed amendments to all users of a module on a single machine. Programmers need not therefore worry about keeping accurate records of the recipients of modules. The ability to test a change without actually updating the affected module makes it safe to have a module amendment system that automatically propagates change to all users. A potential change to a module can be tested without propagation of the change. The change control facilities still give the user the opportunity to cut himself off from module updates by creating his own copy of the module.

The detailed specification of tools that could be developed to provide automated support for configuration management remains to be done. A PSE such as Flex demonstrates that it is possible to support such tools in a flexible way.

## 7. ACKNOWLEDGEMENTS

The authors wish to acknowledge the help of their colleagues at RSRE who developed the ideas embodied in the Flex PSE and demonstrated their use.

## 8. REFERENCES

1. Currie, I.F., Edwards, P.W., and Foster, J.M., 1985, "PerqFlex Firmware", RSRE Report 85015.
2. Currie, I.F., Edwards, P.W., and Foster, J.M., 1982, "Flex: A working computer with an architecture based on procedure values", RSRE Memorandum 3500.
3. Landin, P.J., 1964, "The mechanical evaluation of expressions", Computer Journal, Vol 6, No 4, pp308-320.
4. Currie, I.F., 1982, "In praise of procedures", RSRE Memorandum 3499.
5. Stanley, M., 1986, "Using true procedure values in a programming support environment", RSRE Memorandum 3916.
6. Dunn, R. and Ullman, R. "Quality Assurance for computer software", McGraw Hill 1982
7. Bersoff,E.H., Henderson,V.D.and Siegal,S.G. "Software configuration management- an investment in product integrity" Prentice Hall 1980.
8. Glasser,A.I.,"The evolution of a source code control system" Proc. of the Software Quality and Assurance workshop in Software Engineering notes , Vol 3 no 3 July 1978
9. Rochkind,M."The source code control system" IEEE Trans. on Software Engineering, Vol SE-1 no 4 Dec 75.
10. Stanley.M. 1986 "An evaluation of the Flex PSE" RSRE Report 86003
11. Stanley.M and Goodenough S.J, "Some practical aspects of software re-use" Proceedings of Alvey conference on software engineering environments, Lancaster 1986; ed, I. Sommerville, Peter Peregrinus
12. Stanley M., "Extending data typing beyond the bounds of programming languages." RSRE Memorandum 3878
13. Foster J.M and Currie I.F, "Remote capabilities" Computer Journal, Vol 30 No 5, Oct 1987
14. Foster J.M and Currie I.F, "The varieties of capabilities in Flex" RSRE Memorandum 4042 1987
15. Chen P.P "The entity relationship model: Towards a unified view of data" ACM Transactions on Database Systems, 1976 Vol1 No 1.
16 . PCTE Functional Specifications 4th edition
17. Lyons T and Tedd M.D "Recent developments in tool support interfaces. CAIS and PCTE" 6th Ada UK Conference, York Jan 1987.
18. Lyons T and Tedd M.D, "Technical overview of PCTE and CAIS" 6th Ada UK Conference, York Jan 1987.
19. Bernard. Y, Lacroix M, Lavency P and Vanhoedenaghe M "Configuration management in an open environment" 1st European Software Engineering Conference, France, Sept 1987
20. Stanley, M., 1986, "Using values without names in a programming support environment", RSRE Memorandum 3901.

DOCUMENT CONTROL SHEET

Overall security classification of sheet ........ UNCLASSIFIED ........

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S) )

| | | | |
|---|---|---|---|
| 1. DRIC Reference (if known) | 2. Originator's Reference<br>Memo 4127 | 3. Agency Reference | 4. Report Security Classification<br>Unclassified |

| | |
|---|---|
| 5. Originator's Code (if known)<br>778400 | 6. Originator (Corporate Author) Name and Location<br>RSRE Saint Andrews Road, Malvern, Worcs WR14 3PS |
| 5a. Sponsoring Agency's Code (if known) | 6a. Sponsoring Agency (Contract Authority) Name and Location |

7. Title

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers) Title, place and date of conference

Alvey Conference on Software Engineering Environments - Durham, 21-24 Mar 1988

| | | | | |
|---|---|---|---|---|
| 8. Author 1 Surname, initials<br>Stanley M | 9(a) Author 2<br>Hammond P D | 9(b) Authors 3,4... | 10. Date<br>1988.2 | pp. ref.<br>24 |

| | | | |
|---|---|---|---|
| 11. Contract Number | 12. Period | 13. Project | 14. Other Reference |

15. Distribution statement

Descriptors (or keywords)

continue on separate piece of paper

Abstract

Software configuration management is a necessary part of any project support environment. This paper discusses what software configuration management aims to achieve and proposes a basis from which a software configuration management system can be developed. The proposed basis is illustrated by reference to the software configuration management features demonstrated in the Flex programming support environment, developed at RSRE Malvern.

S80/48

END

DATE
FILMED

8 8[illegible]