PRELIMINARY EXPERIENCE WITH A CONFIGURATION CONTROL SYSTEM FOR MODULAR PROGRAMS

-o-o-o-o-

J.Estublier, S.Ghoul, S.Krakowiak
Laboratoire de Génie Informatique
IMAG, Grenoble, France

1. INTRODUCTION

This paper describes some preliminary experience gathered during the implementation and early use of a program composition and version control system. This system has been designed and implemented as a part of the Adele research project, a programming environment for the production of modular programs (Estublier 83). This project has four main components: a) a program editor, interpreter and debugger; b) a parameterized code generator; c) a user interface; d) a program base, the subject of this paper. The current version of this environment has been developped on a Multics system.

The program base, including the system composition and version control mechanisms, has been used for six months, notably for its own development and maintenance.

This part of the effort in the Adele project has been directed towards the problems of the development and evolution of large experimental systems. Its main objectives are:

1) to provide a data base for the long-term storage of the components of a software system,

2) to provide a language for the description of system composition, including a provision for the description of multiple versions and of user-specified constraints,

3) to automate such operations as changing of versions and propagating the effects of a local modification, in a safe and efficient way.

The following guidelines have been adopted for the design:

- the design is independent of the programming language, as long as the notions of separately defined interfaces and implementations are provided,

- the dependency on the underlying operating system is confined to a low-level layer which implements a set of file-handling primitives,

Experience in this area is still limited; related work is described in (Cristofor 80, Kaiser 82, Lampson 83, Schmidt 82, Tichy 82a,b).

The paper is organized as follows:

- Section 2 presents the overall design and implementation principles of system composition and version control: definition of the components and relations, naming scheme, expression of consistency constraints, data structures.

- Section 3 is a description of the two main algorithms used in the current operation of the version control system: the composition of a system, and the effect of a modification on a part of a compound system.

- Section 4 contains an account of the early experience gained in the use of the version control system, and some indications for improvements in its functions and internal structure.

2. DESIGN OF THE VERSION CONTROL SYSTEM

2.1 System components and relations

Let us first briefly recall the principles of modular program composition; these principles are common to the current languages which offer the module construct, such as Mesa, Modula-2 or Ada.

A system is defined as the association of an interface, which is a description of the resources provided by the system, and an implementation, which actually provides these resources. An implementation may consist of a single "self-contained" module body, which provides by itself all the resources. However, a module body usually relies on external resources. These are in turn described in other interfaces and provided by other implementations. Thus the implementation of the original system consists of a set of module bodies, which we call a configuration. The user of a system only relies on the interface specifications; whether the interface is implemented by a single module body or by a configuration is irrelevant for the importer of the interface, as long as the specifications are met.

In summary, the modular decomposition defines two classes of objects: interfaces and configurations (with "self-contained" module body as a special case of a configuration). Between these objects, two relations are defined:

- a configuration implements an interface if it provides all resources described in that interface,

- a module body requires an interface if it uses a resource described in that interface; the interfaces of the component module bodies of a configuration are said to be internal to the configuration; any other interface required by one of the component bodies is said to be required by the configuration.

Let us assume, for the time being, that the module body which implements an interface is uniquely defined. The relation "requires" defines a directed acyclic graph; all the module bodies which make up a configuration may be obtained, starting from the interface, by constructing the transitive closure of the "requires" relation.

We now introduce multiple versions for interfaces and configurations. For module bodies, we define the usual 2-level (version-revision) scheme (Cristofor 80, Kaiser 82). In addition, we allow a third level of evolution for interfaces (the notion of an interface family).

The following notions are defined:

a) a family is a set of related interfaces. Usually, a family consists of different subsets of a given set of facilities. Thus the resources provided by a file system may be described by a family; the different interfaces would describe subsets of this family (e.g. read only, or (open, close, read, write), or (create, delete)).

b) a version defines a specific instance of a module body.

c) any version may undergo a sequence of revisions. These revisions are numbered by successive integers.

The distinction between version and revision is somewhat arbitrary. In principle, versions correspond to significant changes (e.g. different operating environment, or different time-space tradeoff, etc), whereas revisions result from error corrections, enhancements, etc. We make this distinction more formal by specifying that all revisions of a version have the same required facilities. Any revision that involves a change in the resources required by a version implies the creation of a new, different version.

The naming scheme for objects within a system is a hierarchical one; it is derived from the graph of the relation "requires" (also called dependency graph), as explained in the following example. At each level of naming, a default is specified. Thus frequently used options have short names and they are efficiently retrieved.

Example. The notions presented in this section are illustrated by an example, which we shall use throughout the paper. The system described on figure 1 is a very simplified version of the program base manager.

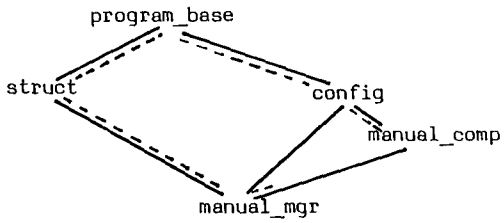

Figure 1. The structure of a simple system

The system is composed of 5 modules; its dependency graph is shown in full lines. The identifiers (program_base, struct, etc) are family names.

Let us make the following assumptions:

- each family contains a single interface, except for manual_mgr, which has two interface versions int1 (the default) and int2.

- each interface is implemented by a single module body in a unique version, except for struct and for manual_mgr-int1, each of which may be implemented by two versions v1 and v2. In both cases, version v1 is defined as the default.

In this example, the naming scheme may be described as follows:

1) A spanning tree is defined for the dependency graph; it is shown in dotted lines on figure 1. The choice of this tree is left to the user.

2) Each family is denoted by a full name derived from the tree (e.g. program_base>struct>manual_mgr.

3) Interface and body names are defined by the following syntax

<interface name>::=
<family name>- <interface identifier>
<body name> ::= <interface name>-
<version identifier>.<revision number>

The star convention may be used in names with its usual meaning (a star matches any identifier).

4) If any of the optional elements is missing, the default is assumed.

5) In addition, any node of the tree may be chosen as the root of a working environment, similar to a working directory in a file system. Family names are interpreted in the working environment.

For instance, if struct is the current working environment:

-v2 denotes the last revision of version v2 of the body which implements the (unique) interface of family struct

manual_mgr-- denotes the last revision of version v1 of the body which implements interface intl of manual_mgr (recall that intl and v1 were defined as the default)

The analogy with a file system is extended to include two notions:

- Visibility: the naming tree also defines a pattern for visibility. Visibility along the arcs of the dependency graph which are not spanned by the tree must be explicitly specified. This provides some protection against uncontrolled evolution,

- Access rights: access lists are attached to each object and checked at each access against user rights.

In a configuration which contains multi-version bodies, the relations "implements" and "requires" do not uniquely define a configuration any longer. The choice of the right component at each level of the hierarchy must be directed by some specification. This problem is the subject of the next section.

2.2 Overall program structure and consistency constraints

2.2.1 Principles

A well-known source of errors in multi-version systems is the internal inconsistency due to the coexistence of "incompatible" versions of some modules. We therefore need a means to define what a "consistent" system is. In order to assist the designer in the expression of consistency constraints, we define a set of attributes for each object. An attribute is usually a name-value pair. Some of the attributes are user-defined; others are automatically derived by the program base.

The constraints on a program component express restrictions on the objects that this component may (transitively) require. These constraints take the form of logical expressions involving conditions on attribute values. Thus one may express implications (e.g. version x of module A requires, or excludes, version y of module B). Default conditions are also attached to each object; they are in effect if no other constraint is specified.

A more detailed description of attributes and constraints, with examples, is given at the end of this section.

The following principles have been adopted for the representation of the attributes and constraints:

- uniformity: the structure of the representation is uniform for all classes of objects (interfaces, module bodies, configurations),

- locality: all the information relevant to the use of an object is attached to that object,

- implicit or automatic derivation: the system makes use of whatever information that can be automatically collected in or derived from the data base; the information provided by the user is restricted to its minimum.

The uniform structure for an object consists of two segments: a text and a manual. For a module body or an interface, the text segment contains the source text of the body or interface; for a configuration, the text contains a configuration specification and a composition list. This list contains the names of all the bodies that compose the configuration; it is constructed by the system when the configuration is created, using the configuration specification provided by the user. This specification may be explicit (all components are specified by their full name) or implicit (some components are specified by relations on their attributes). The specification may even be empty, in which case all default options are chosen. Configuration specifications are described in section 3.1.

The manual of an object contains two parts: user-specified attributes and constraints (as defined above), and system-derived informations. This latter part contains the dependency list of the object, which gives the names of the interfaces required by this object. The contents of the manual is uniform for all classes of objects.

2.2.2 Attributes and constraints

We now give a more precise description of the form and meaning of the attributes and constraints, as they appear in a manual.

1) Attributes. The attributes of a module body may be classified as follows:

- Descriptive attributes may specify any property defined by the designer; in addition, a small number of attributes (e.g. date, status) are automatically maintained by the system. Descriptive attributes are prefixed by the keyword attribute. They have the form (attribute name = value).

- Visibility attributes are prefixed by the keyword visible. They specify the list of environments (i.e. family names) from which the specified component is visible. They may be used for security reasons.

2) Constraints. A constraint is the expression of the presence or absence of specified program components in the dependency list of the constrained object. The components are specified either explicitly (by their name), or implicitly (by a condition on their attributes). Three types of constraints may be specified.

- Imperative constraints: the specified program components must be part of the dependency list of the object.

- Exclusive constraints: the specified program component must not be part of the dependency list of the object.

- Conditional constraints: these only apply to program components specified by conditions on their attributes. A conditional constraint only applies if these attributes exist; if not, the constraint has no effect.

3) In addition, default conditions may be defined; they are used only if they do not conflict with a constraint.

Example. We shall describe a typical contents for a manual, using the example system introduced in section 2.1. The manuals of some of the module bodies of this system are displayed on figure 2.

Most of the information on figure 2 is self-explanatory. The constraints on module body struct--v1 specify that any program component transitively required by this body must have the attributes concur and version, with values "true" and "83", respectively. Attribute alloc is not required; but if it is present, it must have the value "dynamic" The module bodies which implement interfaces of the family manual_mgr do not have any constraints since their dependency list is empty; they have a visibility attribute, which specifies that they are visible from any family in the program_base environment. All revisions of a version have the same manual, except for two attributes, status and date, which are specific to a revision. The meaning and use of the status attribute is discussed in section 3.2.

3. ALGORITHMS FOR SYSTEM COMPOSITION AND CONSISTENCY ENFORCEMENT

In this section, we give an outline of the main algorithms used in the program base. A detailed description is given in (Ghoul 83).

3.1 Configuration specification

A configuration specification has the same form as the constraints part in a manual. Thus the program components which make up a configuration may be specified either explicitly, by name, or implicitly, by imperative, exclusive, conditional or default selections. Configuration specifications are illustrated with the example of section 2.1.

```
config program_base--ex1
   imperative
   * (version = 83)
   struct>manual_mgr-intl-v1.02
   conditional
   * (concur = true)
end

config program_base--ex2
   imperative
   *((date > 6.20.83 )
      or (status = approved))
   exclusive
   (status = incoherent)
end
```

The specification of program_base--ex1 constrains all components of this configuration to have the attribute version, with value 83; the module manual_mgr must be used with interface intl and body v1.02; in addition, if any component has the attribute concur, its value must be true.

The specification of program_base--ex2 constrains all components to have either a date later than 6.20.83 or to have the status "approved"; it excludes any component with status "incoherent".

3.2 System composition

The following algorithm constructs a configuration which implements a given interface, starting from a configuration specification. The system attempts to construct the transitive closure of the dependency relation by a breadth-first search. For each interface, it must select a single implementation. This is done as follows:

1) Process the consistency constraints that apply to the interface (these constraints are found either in the specification of the configuration being built or in the manual of already selected implementations). This processing is done in three steps.

- (process conditional constraints): if a module body has an attribute name specified in a conditional constraint, this

```
manual  struct--v1

   <user-defined>

   attribute
       (implem = tree)
       (concur = true)
       (version = 83)

   imperative
       *(concur = true)
       *(version = 83)

   conditional
       *(alloc = dynamic)


   <system-derived>
       (dep_list = manual_mgr-int1)
       (including_conf = )
       ....

   <revisions>
       01 : (date = 6.20.83)
            (status = approved)

end
```

```
manual  struct--v2

   <user-defined>

   attribute
       (implem = linear)
       (concur = false)
       (version = 84)

   default
       *(concur = false)
       *(version = 84)


   <system-derived>
       (dep_list = manual_mgr-int2)
       (including_conf = )
       ....

   <revisions>
       01 : (date = 12.8.83)
            (status = experimental)

end
```

```
manual  manual_mgr-int1-v1

  <user-defined>

  attribute
      (alloc = static)
      (concur = false)
      (version = 83)

  visible
      prog_base>*


  <system-derived>
      (dep_list = )
      (including_conf = )
      ....

  <revisions>
  01 : (date = 08.03.83
       (status = experimental)

  02 : (date = 09.05.83)
       (status = approved )

end
```

```
manual  manual_mgr-int1-v2

   <user-defined>

   attribute
      (alloc = dynamic)
      (concur = true)
      (version = 83)

   visible
      prog_base>*


   <system-derived>
       (dep_list = )
       (including_conf = )
       ....

   <revisions>
   01 : (date = 11.02.83)
        (status = experimental)

   02 : (date = 12.01.83)
        (status = approved)

end
```

```
manual  manual_mgr-int2-v1

   <user-defined>

   attribute
       (alloc = dynamic)
       (concur = true)
       (version = 84)

   visible
       prog_base>*


   <system-derived>
       (dep_list = )
       (including_conf = )
       ....

   <revisions>
   01 : (date = 11.30.83)
        (status = approved)

end
```

Figure 2. Examples of manuals for module bodies

constraint is added to the list of imperative constraints,

- (process imperative constraints): construct the list of all components (i.e. module body revisions) that match the imperatives constraints (any module body matches an empty list of constraints),

- (process exclusive constraints): delete from the above list all components that match the exclusive constraints.

In this process, any conflicting (i.e. incompatible) constraints are detected and marked.

2) Scan the component list constructed in step 1.

- if a conflict was detected, the configuration is marked as "inconsistent"; a warning is issued to the user.

- if the list contains a single component, select it; if several choices are possible, use the default rules (e.g. most recent revision or user defined default rule),

- if the list is empty, the configuration is marked as "incomplete"; a warning is issued to the user,

During this process, the composition list and the system-defined part of the configuration manual are constructed. If the algorithm succeeds, the configuration is attached to its interface; it can now in turn be selected (according to the contents of its manual) if the interface is used in the construction of an enclosing configuration.

3.3 Propagation of changes

We now specify the effect of a modification of a part of a configuration.

1) Modification of a module body: the effect is to create a new revision for this body.

2) Modification of a manual: the modification may affect the consistency of the configuration in two ways (upwards or downwards in the dependency graph).

3) Modification of an interface: in this case, nothing can be done automatically except inconsistency detection.

A reconstruction algorithm proceeds in two steps. In the first step, the components which are involved by the modification (e.g. because they depend on a modified object) are marked. In the second step, the marked components are processed in bottom-up order (with respect to the dependency graph). Typical reconstruction algorithms (e.g. (Feldman 79)) use timestamps to detect the modified objects.

We have chosen to introduce a more elaborate status information, and to maintain this status up to date, including the automatic propagation of changes; however, the reconstruction itself, which is a fairly expensive operation, is only carried out on user request. After a modification has been prepared, but before its execution, its effect on the status of the system is displayed to the user, who may then decide either to cancel the modification or to have it carried out. The user may now request the reconstruction of the system.

The status information of a component is described in a status attribute, which is attached to each object. It may take the following values:

- incoherent, if the object is subject to conflicting constraints, or if its text is syntactically incorrect,
- incomplete (for configurations only) if the composition algorithm was unable to select a revision for the implementation of some family,
- mod_int, if an interface required by the object has been modified,
- obsolete (for configurations only), if none of the above applies, and if, in addition, the reconstruction of the configuration would change its composition list.
- experimental, if none of the above conditions apply.

In addition, the user may modify the value of the status attribute. These modifications preempt those made by the system. At user request, the system may list, for each objet, the modifications which changed its status.

The status information may be used to characterize a consistent system. We define two forms of consistency:

- a system is strongly consistent if all its components have the "experimental" status,

- a system is weakly consistent if all its components have either the "experimental" or the "obsolete" status.

A weakly consistent system may not conform to the latest changes in the constraints, but it still may be used for testing or debugging, without having to carry out a reconstruction.

3.4 The user's view

In order to illustrate the functions of the program base, we describe a typical user session for the construction of the simple system described as an example in section 2.1. The source text of the interfaces and module bodies is assumed to be initially stored in a set of files. System prompts are preceded by --; texts between <> are comments or abstracts of a sequence of actions.

1) (Define tree) Define a spanning tree for the dependency graph; the tree defined by the user was shown in dotted lines on fig. 1. It is essentially used for naming the modules.

2) (Initialize) For each component (interface or body), execute the command:

```
create <object name> <file name>
```

This creates program base objects for the system; the dependency lists in the manuals are automatically constructed from the import declarations.

3) (Create configurations)

```
createconf <conf.name> (<file name>)
```

The file specified by the optional <file name> contains the configuration specification expressed by a set of constraints.

```
createconf program_base--ex1 ex1.conf

--unable to find a body in manual_mgr-intl:
--constraints: -intl.v1.02 concur = true
--program_base--ex1 will be incomplete.
--Do you confirm (y/n)? y
```

4) (Execute)

```
exec program_base--ex1 "initiate"

--program_base--ex1 is incomplete:
--a body for manual_mgr is missing.
--Do you confirm (y/n)? y
```

5) (Modify module)

```
reserve config-- <locks the module>
   <modification sequence under editor>
release_and_store <unlock and store>

--this operation will make obsolete:
--program_base--ex1.
--Do you confirm (y/n)? y
```

6) (Modify manual)

```
reserve struc--v1.man
   <modification sequence under editor>
release_and_store

--your operation will make obsolete:
--program_base--ex1 program_base--ex2
--Do you confirm (y/n)? y
```

4. EXPERIENCE AND CONCLUSIONS

4.1 Context of the experience

The program base has been used for six months, mostly by the members of its development group. The main program supported by the base has been its own program, which consists of about 20 000 lines of source code, divided in 43 modules. The depth of the composition tree is 10. Three versions of this system are currently supported; some of the modules have revision numbers as high as 20. Two other programs are also currently developped; each of them consists of about 15 modules and 5000 lines of source code. A total of 1100 segments is maintained by the program base.

In the current version (january 1984), the management of documentation and the facility for concurrent use have not yet been implemented.

4.2 Evaluation of the system

The early experience with the system can be summarized under two headings

1) Methodological impacts

It was noted that the availability of the configuration management tools had a definite influence on the process of system design and construction. Since it was possible to rebuild rapidly and efficiently a system after a component was changed, experiments with alternate versions of modules were made easy; this in turn had a positive influence on the design process, as the designers were encouraged to draw module boundaries so as to allow such variations, and thus to isolate significant design decisions.

Another discovery was the general applicability of the tools. The configuration and version control system was initially designed for languages with separately defined interfaces and implementations. The language used in the Adele programming environment is a modular extension of Pascal. However, the program base has also been used for PL/I programs (about 10% of the total code); in that case, the interface description must be explicitly provided. The program base, which was initially closely integrated in the Adele programming environment, is currently being redesigned as an autonomous system with a wider applicability.

2) Expression and enforcement of consistency constraints.

Configuration consistency is a key concept in a multi-version programming environment. However, due to lack of experience, this concept is not yet stabilized. In our approach, consistency is defined as conformity to two sets of constraints: user-defined specifications, such as attribute selection, and structural constraints, such as conformity of exported and imported resources. As a configuration evolves by a succession of modifications, its consistency must be preserved. Our experience has led us to define two forms of consistency. 1) "strong" consistency: at any time t, the configuration, as results from its evolution, is exactly in the same state as if it were reconstructed from its components, at time t, by the algorithm described in 3.2. 2) "weak" consistency: the configuration conforms to structural constraints, but some of its components are obsolete, in the sense that some user-defined

constraints may not be satisfied any longer after modifications. The notion of weak consistency allows us to limit the "ripple effect" of modifications during the debugging and integration phase.

Another concept whose usefulness was discovered by experience is that of a downwards inheritance mechanism for constraints. A constraint defined as "heritable" would propagate down the dependency graph. This would both simplify the modification algorithms and help prevent conflicts between constraints. The inclusion of this mechanism is currently contemplated.

4.3 Conclusion

Our approach may be compared to other efforts in the same direction. The philosophy of the Unix-based tools (Feldman 79) has provided the general inspiration for the automatic reconstruction; the notion of composition list is from (Cristofor 80). Another, more ambitious, system based on similar ideas is the Cedar Modeller (Lampson 83, Schmidt 82); multi-version systems are also supported by Gandalf (Kaiser 82). The main original points in our approach lie in the expression of multi-version system composition by constraints on attributes rather than by component names (the attributes being locally attached to any component or configuration), and in the extended notion of status. We believe that such an implicit definition if often a more natural specification means for the designer than the explicit naming of components (which is still possible in our system). The price of this increased generality is paid in the complexity of the reconstruction algorithms. Our approach may also be regarded as an extension to that described in (Tichy 82b), where the expression of constraints is restricted to and/or conditions on the nodes of the dependency graph. Our preliminary experience is leading us to take a step backwards in the generality of constraint expression, while preserving our main design. We hope that the current experimentation shall mark some progress in the search of a suitable means of expression for "programming in the large".

Acknowledgments. The work described in this paper has been supported by the Centre National d'Etudes des Télécommunications (projet Concerto). The initial phase of the Adele project has been partly supported by Agence de l'Informatique.

REFERENCES

(Cristofor 80)
Cristofor E., Wendt T.A., Wonsiewicz B.C., Source control + Tools = Stable systems, Proc. Compsac 80 (IEEE Computer Soc. Press) (oct. 1980)

(Estublier 83)
Estublier J., Krakowiak S., Mossière J., Rouzaud Y., Design principles of the Adèle programming environment, Proc. International Computing Symposium on Application Systems Development (ACM), Nuremberg (march 1983)

(Feldman 79)
Feldman S.I., Make - a program for maintaining computer programs, Software - Practice and Experience, vol.9, 3 (march 1979), pp.255-265

(Ghoul 83)
Ghoul S., Base de données et gestion de configurations dans un atelier de génie logiciel, (in French), Thèse de Docteur-Ingénieur, Institut National Polytechnique de Grenoble (déc. 1983)

(Kaiser 82)
Kaiser G.E., Habermann A.N., A description of the correct version control supported by the Gandalf environment, in The 2-nd Compendium of Gandalf Documentation, Carnegie-Mellon Univ. (1982)

(Lampson 83)
Lampson B.W., Schmidt E.E., Organizing software in a distributed environment, in Proc. SIGPLAN '83 Symposium on programming language issues in software systems, vol.18, 6 (1983) pp.1-13

(Schmidt 82)
Schmidt E.E., Controlling large software development in a distributed environment (Ph.D. thesis, Univ. of California, Berkeley), CSL 82-7, Xerox PARC (dec. 1982)

(Tichy 82a)
Tichy W.F., Design, implementation and evaluation of a revision control system, Proc. 6th International Conf. on Software Engineering (ACM-IEEE), Tokyo (sept. 1982)

(Tichy 82b)
Tichy W.F., A data model for programming support environments and its application, Proc. IFIP W.G. 8.1 Workshop on Automated tools for information system design and development, New-Orleans (jan. 1982), North-Holland