# Software Configuration Management for Medium-Size Systems

*W. Reck, H. Härtig*

German National Research Center For Computer Science (GMD)
Postfach 1240
5205 Sankt Augustin 1, West Germany

## 1 Abstract

Software configuration management for large real systems is an ugly task. For such systems large–scale software configuration management systems are necessary. Small systems don't need software configuration management except a way to backup. This paper describes a simple software configuration management system (called SCMB) for real medium–size systems.
SCMB obtains its simplicity from clearly separating revision control (i.e.maintaining sequences) from variant control (i.e. maintaining alternatives) and instrument elimination.
Center of SCMB is the variant control system VCS. VCS maintains variants and sets of variants using a *context free grammar.* The terminals and nonterminals of the grammar reemerge as names of *variant subsystems* of the hierarchally structured system.

Based on VCS, a common revision control system for modules, and a compiler, which is able to analyze the dependency structure of modules, the SCMB is used

- to check the *variant consistency* of the system.
- to generate configurations of the system according to a triple (revision, variant, instrumentation).
- to manage revisions of the *system*

This paper describes design, implementation, experiences and limitations of SCMB, which has been used for the development and maintenance of the BirliX operating system, consisting of 300 Modula-2 modules containing about 200000 lines of code.

KEYWORDS: software configuration management, revisions, variants, instruments, subsystem creation, conditional compilation, physical separation

## 2 Introduction

Software configuration management for large real systems is an ugly task. For such systems large–scale software configuration management systems are necessary. Small systems don't need software configuration management except a way to backup. This paper describes a simple software configuration management system (called SCMB) for real medium–size systems.
SCMB obtains its simplicity from clearly separating revision control (i.e.maintaining sequences) from variant control (i.e. maintaining alternatives) and instrument elimination.

As a system is continuously developed further, continuously new *revisions* of the system arise. Storing old revisions of a system is necessary to fall back on these if the further development becomes faulty, or if faults are reported for old revisions. The function of the software configuration management concerning revisions is to register and store changes as well as to designate (e.g. date, number) and to *identify* revisions. Essentially, revision management is done by managing sequences.
*Variant* management is done by managing alternative realizations of the same concept. Variants of a system appear if the system is designed for different target systems or/and for different user interfaces. Since some parts of a system – the *common base* – are valid for all variants, some parts are valid only for some variants and some parts are valid only for one variant, the software configuration management has to enable the creation of any set of variants. Like revisions, variants need to be identified. At the time of system design often not all target systems or user interfaces are known in advance. That means some variants are created though they are not expected to occur. Therefore the software configuration management should supply ways to easily add one or more variants.
*Instruments* support debugging and tuning a system. As mistakes are made during the development of a system, instruments are necessary to find them. The function of the software configuration management concerning instruments is to eliminate (parts of) them if they are not needed, e.g. for efficiency reasons.
To avoid making manual mistakes the software configuration management must be able, to generate *automatically* an executable system for one *configuration.* A configuration of a system designates the software for one revision in one variant and with certain instruments.

## 3 Related Systems

Generally revisions of modules (or, more general, documents) are managed using text differences. Examples are RCS [Tic85] and Jasmine [MW86].
Variant management in known systems is mixed up usually with either revision control or automatic recompilation issues. Both approaches have inherent disadvantages:

Revision control systems support the appearance of variants by allowing alternative sequences of revisions. Whenever a change is affecting one variant only, an additional revision is created for the respective sequence. Thus, if a change affects the common base of some variants, all alternative sequences of revisions must be changed. This finally leads to the concept of 'merge'. Since merges in systems based on text differences can be handled by text differences only, i.e. without semantic knowledge, they can hardly (or not at all) be done automatically.

Automatic recompilation systems such as Make [Fel83] use dependency descriptions of a set of modules to produce object code. Make supports variants by allowing multiple dependency descriptions with several entry points ('targets'). The dependency descriptions are also used to identify the common base of variants. The problems caused by this approach are result from the fact, that the variant structure is hidden somewhere in a much more powerful and dedicated data structure. Manual or automatic checks of any sort of variant consistency is very difficult. Changes in the variant structure may cause a large number of changes in the remaining dependency descriptions, which could be omitted in appropriate high level variant management systems. In this sense, Make can be considered as 'assembly language for variant management'.

SCMB avoids these disadvantages by clearly separating management of sequences, alternatives and instruments. To manage variants, a simple and efficient variant control system – called VCS – has been built. Together with RCS, which is used for revision control only, and conditional compilation for instrument elimination, an executable system can be generated according to a triple (revision, variant, instrumentation).

## 4 Variant Management

The essential function of variant management is to *identify* a variant of a system. To identify a variant several decisions have to be made on different levels of abstraction. In the case of an operating system, a top level decision can be: 'it is for a single processor machine, not for a multi processor machine'. A decision of the next lower level of abstraction can be: 'it is for a Motorola processor, not for an Intel or a VAX processor'. Finally a decision on the lowest level can be: 'it is for a SUN machine, not for a PCS or a Macintosh machine'.
That hierarchy of decisions can be expressed by the productions of a *context free grammar*. To this end variants are represented by terminals, decisions by productions. The *language* described by the grammar is the set of variants. In the example above,

- terminals, i.e. variants, may be denoted

SUN3.T
PCS.T
MAC2.T
IBM.T
SIEMENS.T
SEQUENT.T

- nonterminals may be defined as

START == Single.N | Multi.N
Single.N == SingleMC680x0.N | SingleIntelx86.N
SingleMC680x0.N == SUN3.T | PCS.T | MAC2.T
SingleIntelx86.N == IMB.T | SIEMENS.T
Multi.N == SEQUENT.T

Describing the variant structure of a system by a context free grammar is independent of how different variants are textually represented. Different variants can be represented inside one module (conditional compilation) or in different modules (physical separation) or by text differences.
Since SCMB strictly separates different software configuration management problems and because conditional compilation must be used for instrument elimination (see chapter 5), different variants are represented in different modules. The decision where an alternative of a variant dependent module is laid down in the system, had been influenced by the hierarchally structure of the BirliX operating system [HKK*90]. BirliX has been built using a 'stepwise refinement procedure', where modules are laid down in different subsystems. Each subsystem consists of a specification and an implementation. A specification consists of a single module, while an implementation consists of a single module or several modules in a subsystem of the next lower level of hierarchy. While a specification is variant independent, an implementation may be variant dependent. Now different variants of an implementation are laid down in different subsystems too. The name of a such a subsystem, called *variant subsystem*, is a terminal or nonterminal.
For example, the topmost subsystems of the BirliX operating system are Nucleus, MemoryManagement, InnerKernel and OuterKernel. The Nucleus consists of subsystems Processes, Synchronization, Scheduler, and so on. Since scheduling on a single processor machine is partially different from scheduling on a multi processor machine, the subsystem Scheduler consists of variant subsystems Single.N and Multi.N. That is one implementation of the scheduler is valid for variants SUN3.T, PCS.T, MAC2.T, IBM.T and SIEMENS.T (derived from Single.N), the other is valid for variant SEQUENT.T (derived from Multi.N).
Figure 1 shows a cut of the *source tree* from the just mentioned example. The source tree is that data structure which contains the modules of a system in all variants with all revisions and with all instruments.

Variant dependent implementations and their specification form a *variant switch*. A variant switch represents a production of the context free grammar. The specification represents the left side of the production, because the specification is valid for all variants (terminals) which can be derived from the nonterminal on the left side of the production. Each implementation represents a terminal or nonterminal on the right side of the production, because the name of the corresponding variant subsystem is such a terminal or nonterminal. In figure 1, the single variant switch represents the production
START == Single.N | Multi.N .

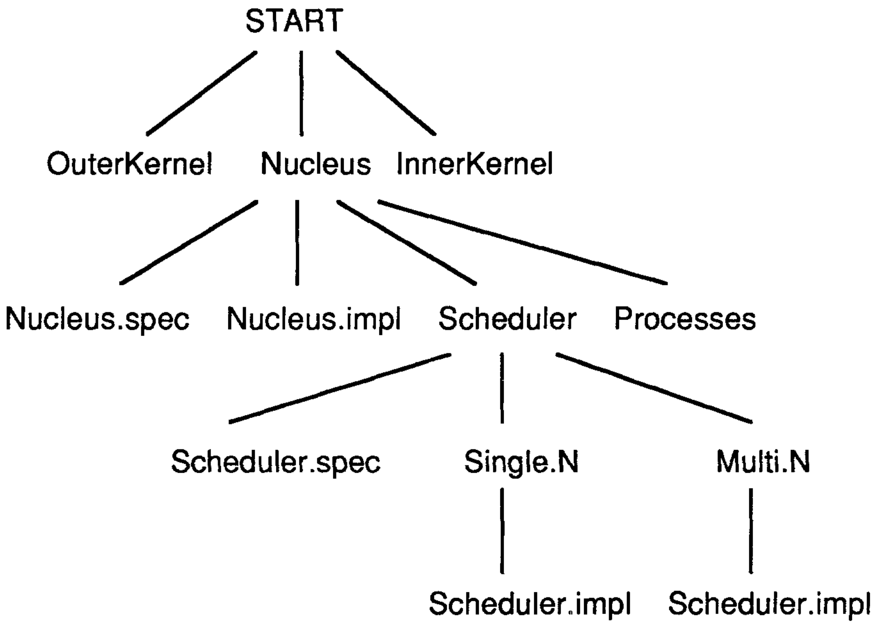

Figure 1: Hierarchally structured source tree with variant subsystems

A consequence of the described concept is, that variant switches are as important as 'normal' subsystems. If only a small part of an implementation is variant dependent, the variant dependencies receive an own variant switch. The specification of the variant switch defines the interface of the variant dependencies. For example, if only a small part of the subsystem Scheduler is variant dependent, variant switch SchedulerV is created. Then the modules Scheduler.spec, Scheduler.impl and SchedulerV.spec are laid down in the subsystem Scheduler, and one implementation SchedulerV.impl is laid down in the variant subsystem Single.N, the other in Multi.N.
The alternative to the creation of a new variant switch is the existence of several very similar implementations. This alternative has the disadvantage that changes at the variant independent part may be neglected in some implementations. This disadvantage exists in systems like RCS which control variants with the aid of text differences.

As mentioned in chapter 2, the software configuration management has to enable the creation of any set of variants, because the common base of the systems is valid for all variants, some parts are valid only for some variants and some parts are valid only for one variant. Naturally, all terminals derived from one nonterminal form a set of variants.
New subsets of variants are defined by defining new nonterminals using existing nonterminals and terminals.
A new variant arises, if e.g. an operating system is ported to another target system. With respect to the grammar, adding a new variant means adding a new terminal to the grammar. Some existing productions (nonterminals) have to be redefined, and/or new productions have to be added. For example, the above mentioned operating system is ported to the multi processor machine Firefly from DEC. Then the terminal FIREFLY.T is added to the grammar. Because Firefly is a multi processor machine, the nonterminal Multi.N is extended by the new terminal. Now all modules contained in variant subsystems with name Multi.N are valid for the new variant. Certainly this doesn't correspond with the reality for some modules, i.e. there are modules which are correct for SEQUENT.T (the old value of Multi.N), but not for FIREFLY.T. For example, if scheduling on Firefly is partially different from scheduling on Sequent, the variant switch SchedulerV must be created in variant subsystem Multi.N, representing the production
Multi.N == SEQUENT.T | FIREFLY.T

Caused by mistakes in the definition of nonterminals or on the creation of variant switches, some modules may be missing in some variants or some modules may exist several times in some variants.
A system is called *variant consistent* if for each variant each specification exists at most once. Additionally, for each variant in which a specification exists, exactly one appropriate implementation must exist.
Since for each module the set of variants in which it exists can be determined by traversing the source tree, the variant consistency can be checked automatically.

The automatic generation of an executable system according to a triple (variant, revision, instruments) is done in the following way. The source tree is traversed and the specified revision of each module, which is valid for the specified variant, is moved into a so called *compilation library.* Finally the Modula–2 compiler is required to compile (with the specified instrumentation) and link all modules. Because the Modula–2 compiler analyzes the dependencies between modules, no extra 'make' is necessary.
Note that VCS doesn't presume that modules containing a specification are compilation units as in Modula–2 [Wir83]. A specification may be any kind of document.

## 5 Instrument Elimination and Revision Management

As solution for the instrument elimination problem the conditional compilation suggests itself. A manual instrument elimination comes not into question, because most systems are developed further and serviced continuously and the instruments are used again and again. If the used compiler doesn't know conditional compilation, the use of a preprocessor is an equivalent alternative.
When generating an executable system, the compiler is called with the specified instrumentation and compiles all modules with that instrumentation. Of course, instruments which are not specified are not compiled but eliminated.
That conditional compilation must be used for instrument elimination is a strong reason for not using conditional compilation to solve the variant mangement problem. Otherwise instrument elimination and variant management would be interlocked, and the problem of nested conditional compilation would arise.

Each module may exist in any number of revisions independently of the variants. Thus all revisions of a module are valid for the same set of variants.
SCMB uses RCS to maintain revisions of single *modules*. But as import as maintaining the revisions of single modules, is maintaining revisions of the *system*. When identifying a configuration of the system according to a triple (variant, revision, instrumentation), the revision component must be applicable to each module. Revision numbers of modules can't be used because different modules are checked in different often and thus their revision numbers diverge too much.
To solve the problem, SCMB maintains a 'reference tree' and supports *complex updates* of the 'reference tree'. A complex update consists of several modules, changed in 'private trees' of possibly several software engineers. Each update of the 'reference tree' is assumed to be tested in a compilation library. Then the *date* and a short description of each update is stored in a special file. Each date in the special file defines a revision of the system, because it is applicable to each module. In RCS, specifying a date denotes the first revision which is as old as or older than the specified date.
When generating an executable system according to a triple (variant, revision, instrumentation), the revision component is always a date, in general the current date. When falling back on an old revision of the system, a date is selected which is stored in the special file. The short descriptions in the special file can be used to select the right date.

## 6 Experiences

The development of the BirliX software configuration management system began with VCS, because early in the development of BirliX multiple variants were needed. The mistake, which occurred most frequently at this time, was that after changes modules were copied to the wrong place in the reference tree ('update mistake').

Thus some modules existed in an old and in a new revision (at this time no revision management was in use) which led to compilation or runtime errors. To recognize the update mistake early, the variant consistency of the system was defined and checked periodically. Since this checking had sometimes been neglected, it was included into the automatic generation of an executable system. Since RCS is used, each software engineer has to use an uniform update tool. A module is checked in only if it just exists. Otherwise the user of the tool is asked whether he didn't make a mistake. In this way the update mistake is recognized at the earliest possible time.

The automatic generation of an executable BirliX system requires 40 minutes on a SUN3/260. 10 minutes of that are used to check the variant consistency, to eliminate the old compilation library, to check out the needed modules and to build up the compilation library. The remaining 30 minutes are required for compilation (and linking). The Modula–2 compiler MOCKA (developed at GMD Karlsruhe) compiles up to 15000 lines per minute, but slows down with the number of IMPORT statements. If the 'reference tree' is not on the same machine as the compilation library, the automatic generation requires 50 minutes.
Consistency checking (written in Modula–2) requires 1 minute.

The described software configuration management system provides advantages taking into account a number of limitations.
The advantages include an efficient way to maintain variants. Its simple and clear conceptual model makes it acceptable for the users. The structure of a source tree is easy to read. Problems arising with 'merging' are avoided.
The limitations are

- VCS can't be adapted to existing systems. If variants have not been built according to the described model, the system has to be adapted to VCS.
- maintaining various revisions concurrently, as needed for large and extensive systems, is not supported. It's an unanswered question, whether this function can be incorporated into SCMB without giving up its simplicity.
- a hierarchally structured system design is presumed. That implies, that unimportant but variant independent modules (e.g. string operations) are laid down in high level subsystems of a source tree (i.e near the root), because low level subsystems are often variant dependent.

## 7 Future Work

The next tasks concerning SCMB are

- to analyze how the existing theoretical knowledge about formal languages can be used to prove and extend its power.
- to analyze whether it is applicable for general document management.

- to compare the power of VCS with systems like shape [AM88], which use *attributes* to deal with variants. At a glance, both models seem to have the same power. Since in 'attributive systems' a set of attributes is used to define a variant, a set of attributes corresponds to a terminal (i.e. variant) in VCS. Single attributes seem to correspond to nonterminals.

## References

[AM88] Andreas Lampen Axel Mahler. shape - a software configuration management tool. In *International Workshop on Software Version and Configuration Control*, pages 228–243, Grassau, January 1988.

[Fel83] Stuart I. Feldman. MAKE - A Program for Maintaining Computer Programs. *Proc. of the SIGPLAN 83 Symposium on Programming Language Issues in Software Systems*, 18(6):1–13, June 1983.

[HKK+90] H. Härtig, W. Kühnhauser, O. Kowalski, W. Lux, W. Reck, H. Streich, and G. Goos. The Architecture of the BIRLIX Operating System. In *11. ITG/GI Fachtagung Architektur von Rechensystemen*, Munic, March 1990. VDE Verlag.

[LS79] B. Lampson and E. Schmidt. Organizing Software in a Distributed Environment. *Software - Practice and Experience*, 9(3):255–265, March 1979.

[MW86] Keith Marzullo and Douglas Wiebe. Jasmine: A Software System Modelling Facility. *Communications of the ACM*, 22(1):121–130, December 1986.

[Tic85] Walter F. Tichy. RCS - A System for Version Control. *Software - Practice and Experience*, 15(7):637–654, July 1985.

[Wir83] N. Wirth. *Programming in Modula-2*. Springer Verlag, 1983.