Computer Aided Software Configuration Management with KMS

W. Faltenbacher

SIEMENS AG, Munich, West Germany
Communication and Information Systems

Abstract

Software configuration management (SCM) is the discipline of controlling the evolution of complex software systems. This paper surveys the Konfiguration Management System (KMS), a tool that supports the tasks of SCM on a broad range from the early inception of the software system to the end of maintenance with emphasis on configuration control, program generation and system delivery. A distinctive feature is the support to customers for adapting the system according to their needs and to merging adapted systems to new releases of the original system.

Keywords and phrases: configuration management, configuration control, program generation, system delivery, maintenance, open system

Introduction

Background

When developing large software systems, it is essential to control and understand the "static" software product, i.e., its components and their relations, as well as the "dynamic" process of building and modifying this product. Large teams are formed and add to the complexity of the task. The challenges encountered in managing and maintaining such large systems led to a new discipline in computer science: the software configuration management (SCM). Basic concepts and terminology were introduced and standardized in the late 60's [7]. Meanwhile easy-to-read text books have been published on this topic [1], [2], [4].

Tool Support

Early SCM-tools available on a commercial basis were concerned with special aspects of the SCM. Most notable representatives were SCCS [22], RCS [25], CMS [5], and Make [12]. Whereas the first three tools deal with the effective storage of multiple versions of files and provide checkin/checkout procedures, the latter enabled the user to automate the derivation of programs after the modification of source objects. During the last few years tools have been introduced both by universities and industry which attempt to integrate the various aspects of SCM. Examples are Adele [10], Cedar [23], POINTE/PVS [13], DSEE [9] and Gandalf [14]. A discussion of these tools and an overview of the state of the art can be found in [11], [24].

SIEMENS' sub-division AP (Application Programming) - a software producer with over 1200 employees - developed KMS, the Konfiguration Management System running under SIEMENS' mainframe operating system BS2000. KMS was built to provide one tool for all SCM-related activities and to support SCM standardization within the sub-division. Development started in 1983 and, having proved its stability and usefulness within SIEMENS, KMS has now become an official product for external marketing.

Four important aspects of KMS are pointed out in this paper (whenever applicable, the terminology proposed by W. Tichy [24] is followed):

- structure and storage of software systems
- configuration control
- program generation
- delivery and maintenance of open systems i.e. systems which may be changed both by the customer and the producer.

Structure and Storage of Software Systems

Objects

A (software) object is any kind of identifiable document generated during the course of a project which is subject to configuration control. Examples are requirements and specifications documents, user manuals, program sources, object code, test cases, etc. Tichy [24] proposed to consider only machine-readable documents, because only for those control by a configuration management tool can be guaranteed. KMS, however, also administrates objects that cannot be controlled physically by the tool. It can be specified that some documents only available in paper form or on different machines are components of the software system and therefore "complete" systems can be described. Consequently, some of the consistency checks have to be user rather than KMS controlled. KMS supplies system- and user-defined
object types. Each object must have exactly one type. For user-defined object types the form of storage (delta or complete, SAM, ISAM or other access methods, physically controlled by KMS or not) must be specified. Object type names are commonly chosen to indicate the sort of contents of the object. For example, cobol sources could be of type "COBOL", executable programs of the type "PRG". Associated with each object there is a description part describing the derivation of an object (see program generation).

Objects are immutable, i.e. once an object is under KMS control it cannot be changed or deleted (this is true for all objects which are controlled physically by KMS, for all others it should be true!). Since the only thing being constant in a software development is the change, objects will need to be "updated", which implies new objects have to be created. An object and its updates are related via the relation "is version of". Several parallel updates are allowed and used for example for temporary fixes. When temporary fixes are merged with the main line of development the relation "is version of" forms a directed acyclic graph. Every object is uniquely identified by its name, its type and a (KMS-controlled) version number. Therefore objects with the same name but different types may exist. "Version names", e.g. "V 2.0", may be specified and used alternatively to the system controlled version number. When an object is first entered into the system it will have version number 001.00.00. For consecutive updates of the object the first number (the trunk) will be raised, resulting in the version numbers 002.00.00, 003.00.00 and so on. In case there is an update request for an object for which an update already exists the two other places are used (branches). Thus in most cases, given the version number of an object, its predecessor version can be determined immediately through inspection by the user. For textual objects it sensible to specify delta storage in the type definitions thus only storing the difference between two versions. KMS stores forward, interleaved deltas. For computing the delta the Heckel algorithm is used [15].

Structure

KMS stores software systems in product libraries. Type and attribute (see below) definitions are specific to the environment of a product library. Typically one software system with all its versions is stored in one such library. To identify one specific software system out of many versions the concept of configuration lists is used. The idea stems from the "old" hardware tradition to keep stocklists of a system, describing all the parts the system consists of. Within the KMS concept a configuration list is an object of the system controlled type "CONFIG" (a "CONFiGuration") and represents a list. The entries in this list are the identifiers of the objects belonging to the system along with attributes describing them. It should be pointed out that because of this definition configurations do have versions. An implicit consistency constraint on configuration lists is that at most one object of a given name and type is contained in the list, that is, no two "versions of the object" are allowed in one configuration. Entries with object type CONFIG are possible. Thus the relation "consists of" forms a directed acyclic graph. This reflects the reusability of subsystems.

Among the above-mentioned attributes there are system defined attributes like e.g. creation date, lines of code, alias, archive and also nine user-defined attributes to be specified for each product library separately.

Configuration Control

Modification Requests

A modification request (MR) is a change proposal [24]. Without MR's there will be no need to change software systems and thus no need for configuration control. Because MR's to a software system play this integral part in SCM they are stored together with the system itself in one product library. In its present form the functional interface resembles a small subset of PULS [19], a SIEMENS internal product for the administration of MR's. Basically MR's can be identified, classified into trouble reports, change and development requests, attached to affected objects and reported. MR's have the state "open" or "closed". When an MR is first entered into the product library, it has the state open. MR's will be closed by the user when the appropriate changes have been made and the resulting system has been tested.

Activity Packages

To describe the process of building and modifying a software product most commonly models like the "milestone-baseline-process" [3] are used. The corresponding KMS concept is the activity package. Evolutionary software development is considered to be a sequence of system changes, where for such change, based on an already existing system version, a new version is produced. These system changes are triggered by approved change proposals. To plan and control the change process the old (existing) system version and the new (planned) system version are defined as configuration and related via an activity package (see figure 1).

Each activity package represents a certain amount of work (with defined results) necessary to produce a new system version from an old one. For this work milestones, resources, deadlines and responsibilities are planned and documented in the activity package. Thus the activity package forms the interface between the static software-product and the change process (project management). This model is not only valid for the change of existing systems but also for the development of new systems. In this case the configuration on which the change is based is taken to be empty. Via activity packages, teamwork on a common database is supported: 

- every user can get his own activity package
- while working with the activity package every change is local. The "current" configuration with
respect to an activity reflects this view: it contains the base (=old) configuration plus all "local" changes. Outside of the activity packages these changes are not visible as long as they are not released.

- any change of the software-product can be carried out only if it was planned before via an activity package.
- every object planned to be a result of an activity package is listed in a "lock-table". This prevents the element from being planned as a result in another activity package. It will be released from the lock-table as soon as it has been delivered to the product library.
- the owner of an activity package can create "child" activity packages. "Children" can again have children and so on. This is used to structure complex tasks.
- after the changes have been completed, the activity packages serve as documentation for the change history of the system.

Example

Suppose there is a software configuration CAIFB version 4. It contains four (sub)-configurations as shown below plus a central library ("coblib") for COBOL-copies. The type names were chosen to indicate the following contents:

- "lib" hold libraries,
- "asm" hold assembler sources,
- "prg" hold executable programs,
- "mod" hold object moduls,
- "prc","ent" hold dialog/batch-oriented command files,
- "cco","ccp" hold COLUMBUS cobol/copy sources (COLUMBUS is the SIEMENS preprocessor language that supplies "structured programming" constructs for COBOL)

While KMS does not distinguish small and capital letters, caps are used for the names of configuration lists in this example whenever the structural property is to be stressed.

Assume there is an approved change proposal ch1.cr and Mr. Jordan ("Mike") is in charge of carrying out the appropriate changes: here the object "read ccp" (a cobol copy element) is to be changed in the configuration CPY. Suppose further the milestone-baseline-process prescribes tested sources to be delivered at milestone "T26", and integrated systems at milestone "T30".

After starting the program KMS Mike must "identify" himself and the product library (cai) and create a new activity package (w1) to work on a new version of the configuration CAIFB.

`ident cai,w1,jordan, create ,caifb,req = (ch1.cr)`

The milestone T26, T30 and the corresponding results with respect to the configuration CAIFB must be planned:

`defmst t26 (define milestone) defres t26,cpy.config (define result) defmst t30 defres t30,caifb.config`
Now Mike delegates the modification of the configuration element "read ccp" to Bill, who is in charge of all copy elements. Mike creates a child activity package for Bill:

```
defap t26.w1.1,resp = bill,config = (cpy)
```

Bill plans his results and gets his object from the product library:

```
defres t26,read,ccp get read,ccp
```

The following picture visualizes the built process structure:

```
CAIFB 4
  cpy config 8
de config 2
kr config 4
sk config 5
coplib lib 4
```

```
CPY 8
  a-aus ccp 3
mask ccp 8
init ccp 5
read ccp 2
```

To return the changed elements back to the product library, Bill will need to "transfer" his activity package. Then all his planned objects (the changed versions supplied as BS2000-files) will be added to the product library and KMS will create a new version of these elements (here "read ccp 3").

**transfer**

Up to now all the changes made are local to Mike's and Bill's activity packages. Only when a new version of the configuration CPY is produced its changes become "public". This can be achieved by transferring Mike's activity package, i.e. the results of its first milestone. Picture 4 below shows the contents of the current configuration CAIFB with respect to Mike's activity package, after all results for T26 have been delivered and T30 is still open. The points emphasized are:

A) a new version of "read ccp"

B) new version of the configuration CPY

C) identification of the current configuration: "-".

```
figure 3.
```

---

One final remark: the examples above were carried out in KMS' command mode for the sake of clarity and conciseness. KMS also provides a menu-driven shell which - among other things - offers more comfort in regard to choosing parameters already known to the product library.

**Programm Generation**

**Derived objects**

Objects can be classified into **source objects** and **derived objects** [24]. A **source object** is an object created manually by humans, for example via an interactive editor. A **derived object** is an object that is generated fully automatically by a program called **deriver** usually from other objects. Examples for derived objects are programs, object modules, formatted documents, preprocessed sources. To derive an object, the objects it depends on, and the deriver and its parameters must be known. KMS holds this information in the description part of each derived object (object orientation), whereas the widely used UNIX-utility make [12] holds it in separate makefiles.

**Generation steps**

The KMS counterpart of the make command is **generate**. Since generate is integrated in the tool KMS
and made to work with the contents of product-libraries rather than ordinary files the concept differs further from make as shown below.

In order to derive an object, a number of steps have to be carried out. This is somewhat tedious for the first version of an object to be derived but rewarding for later generations. Four generation steps can be distinguished (see also figure 5 below):

- **edit description**: the description part can be viewed as a command file with dummy parameters. Dependencies on other objects are noted version-free. There can be several different "make-instructions" within one description, identified by names. This can be used to generate variants (e.g. for tests, different target machines, ...). The description part can be edited "manually" or via automatic analysis routines.

- **select universe**: an universe is a list that contains the identifiers and date plus time of addition to the library of all objects relevant for a generation. The universe can be selected by choosing the appropriate configuration and/or by manually editing the universe. It can be viewed as a set of actual parameters for the description parts.

- **generate**: input to this function is name and type of the object to be generated and the universe. Let us call this object "G". Generate evaluates the description part of G and also the description parts of the objects G depends on (directly or indirectly). In its most common form generate also evaluates the timestamps of the objects and outputs a command file (DOIENTER) containing only minimal productions, i.e. only commands that produce objects depending on changed objects.

- **run command file**: in order to actually produce the desired object the command file must be executed.

---

**Example**

Suppose the following scenario (see also figure 6):

The program `dex` is produced by linking the object modules `bf01` and `bf10`. The object modules `bf01` and `bf10` are produced by compiling the respective COBOL sources. The copy elements `read` and `init` have to be included in the source as shown in the graph below. The compiler expects the copies in a special library, here "coblib".

![Figure 6](image)

Below the description parts of these configurations elements are listed:

```
ENTRY /* dex.prg */
> MOD BF01 BF10 /* dex depends on "bf01 mod", "bf10.mod" */

ENTRY /* bf01.mod */
> P LIB COPLIB /* previous to compilation coplib must be generated */
> CCO BF01

ENTRY /* bf01.cco */
> D CCP READ INIT Direct dependencies: the description parts of read and init will not be evaluated.
```
After the object "read ccp" has been modified (as shown above) the current configuration CAIFB is used to produce the universe (step 2):

<table>
<thead>
<tr>
<th>Type</th>
<th>name</th>
<th>version</th>
<th>date and time</th>
</tr>
</thead>
<tbody>
<tr>
<td>cco</td>
<td>b01</td>
<td>1</td>
<td>870731102828</td>
</tr>
<tr>
<td>cco</td>
<td>b10</td>
<td>2</td>
<td>87073009030</td>
</tr>
<tr>
<td>ccp</td>
<td>init</td>
<td>5</td>
<td>870701162010</td>
</tr>
<tr>
<td>ccp</td>
<td>read</td>
<td>3</td>
<td>870805113026</td>
</tr>
<tr>
<td>lib</td>
<td>coplib4</td>
<td>4</td>
<td>870725134556</td>
</tr>
<tr>
<td>mod</td>
<td>b01</td>
<td>1</td>
<td>870731144530</td>
</tr>
<tr>
<td>mod</td>
<td>b10</td>
<td>2</td>
<td>870730101711</td>
</tr>
<tr>
<td>prg</td>
<td>dex</td>
<td>2</td>
<td>870731165033</td>
</tr>
</tbody>
</table>

As shown the element "read ccp" is younger than the depending "b01 cco". Thus, if the universe remains unedited, the result of generate "dex prg" (step 3) is:

/PROC
ADD READ TO COPLIB
COMPIL BFO1
LINK BFO1 TO DEX
LINK BF10 TO DEX
/ENDP

Step 4 for is to run that command-file.

**Delivery and Maintenance**

**Open Systems**

An open system is a software system that enables the customer to adapt the system according to his needs. This can be done either by providing defined interfaces as parameter-tables or so-called user exits, where object-modules or dummies are supplied, or by delivering source-code files that can be modified directly. Examples for such systems can be found in the areas CAD, CIM and in banking and accounting applications. Very large open systems within SIEMENS AP environment are the products KORDOBA, SILINE and the SICAD/CADIS family.

With the existence of open systems SCM has to be applied also in costumers' organisations. Additional problems arise:

- Objects and product-structures, and in particular, procedures for generations must also be delivered and administrated consistently.
- Updates released by the producer must be merged with customer modifications.
- In case of warranty claims the cause and the producer of faulty software must be identified.
- To tackle these problems open systems produced by AP will be delivered in KMS product libraries along with functions to support them.

It was very interesting for us to learn that the problem with open systems arise not only between dedicated software manufactures and customers that use this systems but also within large organisations that produce software only for their own purposes. The reason lies in the many regional differences in Europe or even in the Federal Republic of Germany, concerning e.g. holidays, taxes, customs, measures and legal regulations. Thus the regional dp departments have to adapt the "standard" systems and we are in the situation described above.

**Delivery**

The delivery of "regular" software systems, i.e. systems that are not to be modified by the customer, is basically just a matter of correct bookkeeping. In KMS the deliverable objects will be labeled via an attribute signalling its destination. Possible destinations are e.g. test field, software service and customer. There is just one KMS command that - given the desired configuration and destination - produces the appropriate tape. More challenging is the delivery of open systems. When delivering an open system to a customer it should include:

- a list, identified by name and version (= a configuration) that contains the identifiers of all objects ( <name> <type> <version> ) belonging to that version of the system
- the actual (physical) objects described in the list above
- metainformation (dependencies, production-rules ..)

It should not include:

- versions of objects other than those described in the configuration list
- information regarding the change process (activity packages and change proposals)

KMS provides a function to extract exactly one version of a configuration from an existing product library. The result of the extraction is a product library that contains only the version of the extracted configuration list plus the respective objects. For convenience this library is called transport library. It should be pointed out that with this method exactly those objects described in the configuration list will be delivered (consistency!)

KMS also offers the possibility to extract only "delta" transport libraries. In this case only the objects are extracted that have been changed or are new with respect to an older version of the configuration. This is in particular useful for corrections or revisions of the system with little change volume - it should be considered that one version of such an open system can take up 160 megabytes of memory and more.

**Merging**

The delivery of a software system to customers is usually followed up by subsequent updates by the producer. Now customers have to integrate their modifications into the new releases. KMS supports this activity with the update command.
When using the update command the following steps are carried out:
- The new releases' configuration list is copied from the transport library into the product library of the customer;
- all the corresponding objects are copied;
- the configuration will be checked for completeness: all objects described in the configuration list must be contained in the product library of the customer. If not, all actions carried out so far will be taken back (transaction principle). This is useful when a delta transport library was delivered;
- KMS tries to reconcile the customer's and producer's changes and to create a new customer configuration list. For an object there are three possibilities:
  -- neither the producer nor the customer changed: no action is required
  -- either the producer or the customer changed (mutually exclusive "or"): The latest version will be added to the new customer configuration.
  -- the producer and the customer changed. This conflict can not be solved automatically. KMS issues a warning and produces a log-file containing the differences. It is up to the customer to resolve the conflict and add the correct objects to the product library.

A schematic overview of the whole extract/update process is shown in figure 7.

**Summary**

During the last three years KMS has been used to support the configuration management of more than 20 projects, developing software for different application (CAD, banking, telecommunications ...). The project characteristics range from 2-people, 40-modules, 3D-KLOCS systems to 50-people, 5000-modules, 1-MLOCS ones. In general, the software products have become significantly more transparent.

The rigorous planning requirements via activity packages for changes, however, caused acceptance problems in some cases. That is due to the fact that developers prefer to checkout an object whenever an error is detected and to checkin this object as soon as it has been fixed.

Another problem we encountered was that it seemed to be very hard to design a modification request module that is equally adequate for all sorts and sizes of projects. Our conclusion is that some basic functionality has to be worked out and offered. This functionality should be expandable by the user. Examples are user - defined masks, attributes and states.

Before indicating future development and research we would like to mention another motivation for the intended direction. While KMS was intended to be an administration tool with some other software engineering tools somehow separate from it, experience showed that it is desirable that tools like compilers, mask generators, test aids have direct access to the database. This database should include an integrated data dictionary for the various dependencies and relationships between objects, modification requests and activity packages.

Two solutions are currently discussed among the international research community. One is extending traditional file systems to object management systems (OMS) as in the PCTE and CAIS projects [8], [20] and the other is enhancing data base systems to nonstandard database systems as in [7], [16], [21]. SIEMENS is involved in both research directions and working towards standards in ISO. The author feels that the two lines of development show asymptotic behaviour towards a common conceptual basis. Thus it would be technically adequate to work towards one common standardization.

**References**


[17] KMS-Verfahrensbeschreibung SIEMENS AG, D AP, vorläufige Ausgabe April 1985 (in german)

[18] KMS-Manual SIEMENS AG, D AP, Ausgabe Dez.87 (in german)

[19] Meltzer, M., PULS-Bedienungshilfe, SIEMENS AG, D AP, 6.2.84 (in german)


