HiCoV: Managing the Version Space

Bjørn P. Munch*
Telenor R&D
N-7005 Trondheim, Norway
e-mail: Bjorn.Munch@fou.telenor.no
Phone: +47 73 54 44 28  Fax: +47 73 52 85 90

November 1, 1995

Abstract

This paper presents HiCoV, high-level extensions to our Change Oriented Versioning model (CoV), which will reduce the inherent complexity of selecting versions based on this model. Thus, we hope to eliminate the largest obstacle to making CoV practically usable.

1 The Complexities of Version Selection

A central part of software configuration management (SCM) is the selection of a configuration. By configuration, we mean a complete and consistent set of software objects. Since most SCM systems can manage also different versions of the software objects (henceforth called components), the selection process actually consist of two parts:

- Selecting the correct software objects which will make up the configuration. This is related to the product space.
- Selecting appropriate versions of each component. This is related to the version space.

For a large configuration where each component (or at least many of them) exists in many versions, this selection is not trivial. There are usually a number of more or less well-defined restrictions for which component versions can be selected together as part of a consistent configuration.

Those restrictions may be represented in several ways:

- Explicit relationships between software objects, stating that a version of one component requires a certain version (or one within a range of versions) of another component. Simple systems based on e.g. RCS [Tic85] may use this method.

* This work is done at Department of Computer Systems and Telematics, Norwegian Institute of Technology.
Hierarchical version representations: configuration and sub-configurations are versioned, and each version of a (sub)configuration is defined to include specific versions of its constituent parts. An example of this is CVS [Ber90].

Attribute based selection, where attribute values on the versions decide which get chosen. Attribute values on already selected versions, and rules associated with them, may also be used to restrict further choices. An example of this is Adele [BEM93].

Intensional selection, where some logical expression defined over the whole database defines the version of the configuration, and which version to pick of each component can be deduced from this. An example of this is Aide-de-Camp [ade90].

Whichever way the configuration restrictions are represented, the SCM system must give appropriate support to the developers. This includes:

- Manually specifying CRs.
- Automatically maintaining CRs, based on work units.
- Using CRs for guiding the user to a valid selection.
- Disallowing user selections which are in conflict with the CRs.
- Presenting the CRs to the user.

An overview of the state-of-the-art in selecting configurations of a versioned software product can be found in [CW95].

The rest of this paper is organized as follows: Section 2 briefly describes our versioning model CoV and explains the particular problems this model introduces. Section 3 then describes a set of version rules which are intended to alleviate some of the problems. We take this further into a description of planned support tools in Section 4. Section 5 describes some initial experience with CoV. We give a conclusion and indicates further work in Section 6.

2 Our Background: CoV

2.1 Definitions

Change Oriented Versioning (CoV) falls into the “intensional selection” category of representing restrictions. Versions are defined via global, logical variables called **options**. Each option corresponds to some property of the product. The user may create a new option whenever he adds a new property to the product or part of it.

Options will be identified by their name. Examples could be:

**GUI** : Set to **TRUE** for versions of programs which supply a graphical user interface. Can also be used for the documentation.
SunOS: Set to true for program versions specially adapted for being run under the SunOS operating system.

OptSpeed: Set to true for versions specifically optimized for speed.

In order to select a version of a configuration, the user will specify a version choice, in CoV simply called choice. This is a list of bindings of options to either true or false. The choice will define which version of every object is included in the configuration, and also which objects are part of it.

When a set of changes are applied to the product, the user must set an ambition for the changes. An ambition is also a set of option bindings (usually binding fewer options than the choice) which defines the “scope” of the changes. This ambition limits which versions are being affected by the changes. Once the set of changes (which may touch a single component, or span the whole product) have been committed, they will be included in any future version selection based on a choice which includes all the option bindings of the ambition.

For example, if the ambition GUI \& OptSpeed is used on a set of changes, those will later be included whenever the user specifies a choice binding both these options to true, irrespective of which other bindings the choice includes.

In the above example, the user will introduce changes which are specific to this combination of two options, and which will only take effect when both are set to true. Thus, one can make the combined version quite different from what is produced by the automatic merge. This will be completely transparent to the user.

For a more thorough description of CoV, see e.g. [Lie90, MLG+93].

2.2 Current Implementation

Our prototype SE Database, the EPosDB [Mun93] implements the basic CoV for objects and relationships (according to an extended ER data model) and for “longfields”, i.e. files stored under DBMS control.

On top of the EPosDB, we have built a higher-level user interface which we have simply called ECM (= EPOS Configuration Management). ECM gives the user access to the configuration through names rather than OIDs. This is currently in Prolog; we are building a graphical UI on top of this again. ECM also manages the user's workspace directory and takes care of checking files into and out of the EPosDB.

2.3 Problems

While CoV completely solves the problem of deciding which component versions go together, the selection problem is far from being solved; it has simply been transformed into another dimension. Instead of having a selection problem in the product space, we now have a problem in the version space. We have to manage rules about what options go together, rather than what objects.
This has traditionally been the major criticism against CoV and similar systems, like those based on change sets. We also believe it is the greatest obstacle in the way of making CoV practically usable. Even our (so far) simple experience of using it on our own development has shown these problems. The problems facing the CoV user can be classified into a number of sub-problems:

- What options are available, and what do they mean?
- Which options are relevant for this part of my product, and which ones can I ignore?
- What are the restriction on which options go together? Having selected an option:
  - Which other options are then not selectable?
  - Does that option depend on some other option also being selected?
- Which versions have passed through various stages of testing, QA etc., and can thus be used for a certain purpose?
- Which versions have been “frozen” and should not be modified in the future, e.g. because they have been delivered to customers?

### 2.4 Moving the Selection Problem

In comparison with traditional, “version-oriented” approaches, we can see from the above that we have moved the selection problem into the *version space*. Our problem is not to select the product given a version description, but to select the version description in the first place; the product then “falls out” automatically. The purpose of HiCoV is to ease the first phase of the configuration selection process for the user.

One very simple change which is being introduced is to add a mandatory descriptive text to every option. The creator of the option should use this to describe its purpose. Currently, only the name itself may give any hint of its use.

In the rest of the paper, we will look at various techniques to reduce the selection problem in this version space.

### 3 Adding Version Rules

The rules described in this section are similar to the high level version descriptions that were introduced in [GKY91]. Examples will be jointly presented in Section 3.7.

In addition to all the version rules, all options which are not bound through any other mechanism in a version choice, are interpreted as though bound to **false**. This ensures that a choice is complete and identifies a unique version.
3.1 Option Constraints

Not all option combinations make sense. Some options may be mutually exclusive, some may depend on others, and these constraints restrict how options can be combined. Typical examples of the former include options for different hardware or different language in user interface text. An example of the latter can be options for deciding layout of a graphical user interface, which make no sense unless GUI is selected in the first place.

3.1.1 Mutual Exclusion

Mutual exclusion means that at most one of a set of options may be bound to TRUE. We cannot model this by relationships directly between the involved options. Not only would that be inconvenient when we have many options, but more fundamentally, we cannot correctly model an option which is member of more than one mutually exclusive set.

Instead, we introduce a concept of option set. An option set represents a set of (at least two) mutually exclusive options; no more than one of the options in a set may be bound to TRUE in any ambition or choice. An option may be the member of any number of sets. The members of an option set are not fixed; the set may expand over time by more options being incorporated.

3.1.2 Option Dependency

Dependency of one option on another means that the option cannot be bound explicitly to either TRUE or FALSE in an ambition or choice unless another option is bound to TRUE. This is naturally a relationship between the two options, and from this one can construct a dependency graph.

We should point out that an option dependency is not simply a matter of stating “this feature requires that feature”, but rather “this feature does not even make sense to talk about, unless we have that other feature”.

Since this dependency is not a standard logical implication, we need some other syntax to describe it. One possibility is shown below, indicating that \texttt{fix3\_unix} depends on \texttt{UNIX}:
\[
\text{Dep} : \text{fix3\_unix} \rightarrow \text{UNIX}
\]  

We will also allow an option to be dependent on an option set (as defined above). Such a dependency means the option requires one (and only one) of those in the option set to be bound to TRUE. More informally, “this feature also requires one of these other features to be selected”. A possible use is e.g. to force the user to select among interface languages.

3.1.3 Sub-Options

Another typical dependency can be:
\[
\text{Dep} : \text{SunOS} \rightarrow \text{UNIX}
\]  

5
This example is actually a case of what we call sub-options, since SunOS can be thought of as one of several possible specializations of UNIX. All sub-options of UNIX will naturally form an option set, since we can only choose one of them.

If an option set is created with the same name as an already existing option, the set (and hence indirectly all its members) is automatically made dependent on the option. The members of this option set will be called "sub-options". Note that the dependency does not go the other way: you don’t have to select any of the sub-options after having selected UNIX.

3.2 Product Specific Options

Just as some option combinations do not make sense, not all option/product combinations are useful either. For example, it would not matter what the option GUI is bound to when you are making changes to a database kernel. On the other hand, some options may apply to the kernel only, such as alternative B-tree algorithms.

By making some options local to a specific (sub-)product, we can achieve two things:

- The user will not have to consider all options for other (sub-)products than the one he is working with.
- We do not have to worry about whether the combination of two local options for non-overlapping sub-products are compatible. Since their changes cannot touch the same components, they will not be in conflict.

The product local option is only allowed to be bound if the configuration includes parts of the product to which the option belongs. Thus, the decision about using the option is not a pure CoV decision but has to be done by a tool which also knows the product structure.

Note that the option is still globally applicable, thus not breaking with the basic philosophy of CoV. The product locality only restricts the scope within which is is visible, and thus selectable, by the user. This also implies that a local option may be given a larger scope later, if the user finds out it has a more global perspective than originally assumed.

3.3 Aggregates

It is a quite common situation that a developer or team performs a number of sequential bug-fixes or correction to a product. This will often happen e.g. after the first version of a new or enhanced product has entered testing.

In these situations, there will often be little need for picking only some of those corrections in a choice. One would normally select none (to get the original, non-corrected version) or all (to get a version with as few errors left as possible). Instead of having to name each of the bugfix options, we can make things simpler by introducing the concept of aggregate options.

An aggregate option is simply a single entity representing a number of options; it is not a real option in itself. The user can select it as he selects a normal option. When the aggregate is chosen, the system will expand it and actually set all the part options to TRUE.
3.4 Validities

A validity is, simply stated, a disjunction ("or") of not necessarily complete version choices (see 2.1 for the definition of version choice):

\[ Val = VC_1 \lor VC_2 \lor \ldots \lor VC_i \]  

(3)

These choices, or validity terms, are built up like choices and ambitions: each is a conjunction ("and") of option bindings, as was described in Section 2.1.

A validity indicates that the included versions have some common property. It is generally up to the user or application to interpret the semantics of these properties, and to decide when to add new terms to them. HiCoV provides support for maintaining the validities.

A typical use is to assign some form of status value to different versions depending on how "good" they are conceived to be. Examples could include "compiled", "module tested", "integration tested", "accepted" etc. There will be one validity for each such value.

Note that we are here talking about global properties of the versions, that will be used to guide choice selection. If status values on (versions of) particular objects are desired, attributes or other data values individual to each object can of course be used in addition.

The validities do not take part in the versioning itself, so all the existing rules and algorithms for handling choices and ambitions remain the same. Validities will not cause any overhead in normal use. What they will affect, is the user's selection of ambitions and version choices.

3.4.1 Stability

The stability is built up just like other validities, but it is used in another way. As the name suggests, all versions covered by the stability are stable, or immutable. We want to prevent any update to them. A simple way to achieve that goal, is to prevent the user from ever working with an ambition that overlaps with the stability. Another important distinction is that this restriction will always take place "behind the scenes" with no need for user interaction.

In general terms, we can say that the normal validities are meant to limit which versions we can read, while the purpose of the stability is, within the same formalism, to limit what version we can write to.

The checking of ambition/choice against stability/validity is done by the evaluator, which logically is an integral part of the CoV service. It cannot be a stand-alone tool, since we must ensure that the stability is always enforced.

3.5 Overview of Option Relationships

Figure 1 shows an example of some options and the logical relationships between them. Note that only the dependency relationships are actually represented this way; the mutex and sub-option relationships are internally represented in other ways\(^1\).

\(^1\)For a discussion of some of the representation issues, see [Mun93].
This (global) option graph is in some ways similar to the (local) version graphs of Tichy [Tic88]. The dependency relation corresponds to Tichy’s `revision of`, while the mutex corresponds to his `variant of`.

Of course, our option “graph” is not a version graph, as there are many more possible versions than options. And the existence of independent and freely combinable options have no equivalent in Tichy’s version graph model, nor do the aggregates.

![Option Graph](image)

**Figure 1: Example of relationships between options.**

*Reviewer note: I just realized I ought to make this figure consistent with the example below; this will be done if the paper is accepted.*

The figure is shown here to illustrate a point; it does not necessarily reflect how a similar graph might be shown buy a visualization tool. See [Gu93] for a description of such a tool.

### 3.6 Preferences

Preferences were described in [GKY91] as a means for the user to indicate desirability of having certain options bindings included in the choice, without insisting. These would be used by the heuristics to select among different alternatives, or for overriding the defaults.

We have chosen not to support preferences at this time. We feel that the described set of rules are already quite challenging to implement and support, and preferences possibly the most complicated of all. We are also rather unsure about how these would be used in practice; some experience with the rest of the HiCoV rules may give us a better understanding.
3.7 Examples of Use

The rules described in this section serve two purposes, namely to restrict what choices are legal and to automate some of the selection process. In order to better illustrate the usefulness of these rules, we will construct a simple example and show how the user is helped by them in setting his ambition/choice.

3.7.1 Option Constraints

In the following description\(^2\), \texttt{mutex <setname>}: means the following options are mutually exclusive, \texttt{dep}: means the first option depends on the second, \texttt{subopt <name>}: and \texttt{aggr <name>}: means the following options are sub-options, resp. aggregate members.

mutex opsys: unix msdos os2
subopt unix: aix ultrix sunos
subopt sunos: sunos4 solaris
mutex language: english deutsch norsk
dep: menus gui
dep: scrollbar gui
dep: help language
dep: helpmenu help menus
dep: msdos intel
aggr scrollfix: bug12 bug14 bug19
dep: #scrollfix scrollbar

Given the above version rules, here are some examples of how user specified version choice will be expanded\(^3\) (! is a negated option, # names an aggregate):

<table>
<thead>
<tr>
<th>Explicit</th>
<th>After Expansion</th>
</tr>
</thead>
<tbody>
<tr>
<td>help msdos</td>
<td>help english !deutsch !norsk intel !unix !os2</td>
</tr>
<tr>
<td>helpmenu solaris</td>
<td>helpmenu help norsk menus gui sunos unix !english !deutsch !sunos4</td>
</tr>
<tr>
<td>bug19 os2</td>
<td>bug19 scrollbar gui os2 !unix !msdos</td>
</tr>
<tr>
<td>#scrollfix deutsch aix</td>
<td>bug12 bug14 bug19 scrollbar gui deutsch !english !norsk unix !ultrix !sunos !msdos !os2</td>
</tr>
<tr>
<td>unix !gui</td>
<td>unix !msdos !os2 !menus !helpmenu !scrollbar !bug12     !bug14 !bug19</td>
</tr>
<tr>
<td>aix english norsk</td>
<td>ILLEGAL</td>
</tr>
<tr>
<td>ultrix bug14 !gui</td>
<td>ILLEGAL</td>
</tr>
<tr>
<td>!intel help</td>
<td>!intel !msdos help deutsch</td>
</tr>
</tbody>
</table>

Note that the explicit choice on the left is also the one that will be presented as the “official” choice when the user asks to see it. Just as he will not have to enter all the extra binding, he should not have to be bothered with them later as part of his work specification. The effective choice on the right can of course also be seen, but must be asked for explicitly.

\(^2\)This is not the syntax we will use; it’s just been made for this example

\(^3\)More precisely, how they would have had to be specified, if no versions rules were in place.

\(^4\)The user is forced to choose a language; may be another one.
3.7.2 Validities

Let us assume that we have the following validities:

stability: (os2 english !gui !help) v
         (sunos4 norsk !gui !help) v
         (sunos4 norsk gui help !helpmenu)

tested:  (os2 english !gui) v
         (os2 english gui !menus !scrollbar !help) v
         (os2 deutsch !gui !help) v
         (unix norsk !gui !help) v
         (sunos norsk !gui) v
         (sunos4 norsk gui !scrollbar)

Ambitions have to be restricted so that they do not overlap the stability. Some examples of ambitions and additional option binding that the system may add:\footnote{Note that this is non-deterministic; there may be more than one possibility}

\begin{verbatim}
  os2 !gui       # !english
  os2 english   # help
  os2 english !gui !help intel # ILLEGAL
  os2 deutsch   # OK (!english is implicit)
  ultrix menus  # OK
  sunos4 norsk gui # !help
\end{verbatim}

Choices have to be restricted so they overlap completely with the validity the user want to comply with. If he has requested tested:

\begin{verbatim}
  os2 gui !help     # deutsch
  os2 !gui help     # english
  os2 gui help      # ILLEGAL
  solaris norsk help # !gui (sunos is implicit)
  sunos norsk gui   # sunos4 !scrollbar
  aix help          # ILLEGAL
  msdos             # ILLEGAL
\end{verbatim}

4 The HiCoV Tools

COVST and VRMAN, the two tools described here, will become an integrated part of our environment. They will be implemented on top of our graphical environment CHAT, which in turn is based on ECM which was mentioned in Section 2.2. We will first make a textual-interface prototype of both in order to get experience with the basic functionality. This work is now in progress.

4.1 VRMAN: Version Rule Manager

The main task of VRMAN is to give the user an interface for creating and manipulating the version rules. Since most of the operations supported by VRMAN are rather trivial, we will
not go into detail about them in this paper, but only mention a few non-trivial ones.

4.1.1 Creating a new Option

The central part of this operation is for the user to provide a name (we will not give any help in choosing names), which will be checked for uniqueness, and a descriptive text.

Calls to this part of the VRMAN tool may come from COVST. In that case, it will already be established into what relationships the new option belongs. In other cases, the user will be prompted for:

- What option(s) the new option depends on, if any.
- Which option set(s) it should be a member of, if any.
- Whether it is a sub-option of another option.
- Whether it’s part of an aggregate option.
- Whether it’s local to a specific (sub-)product.

In order to help the user find options and option sets, rather than having to remember the names, an option space browser will be provided. This should work similarly to the COVST menu system, except that there is no concept of selecting or un-selecting items.

4.1.2 Manipulating Validities

To create a new validity is trivial: just supply a name and a description. By default, validities start out empty (meaning FALSE).

When adding a term to a validity (except for the stability, which we will manage separately), it will usually be, in fact it should in principle always be, the current choice of a transaction in which we have checked that the version in question indeed has the property defined by the validity. Thus, VRMAN supplies an operation which just adds the current choice to a named validity and makes no more fuss about it.

Adding a term to the stability is another matter. Here we may start off with the current, or last, ambition and let the user modify this if possible. He will also be prompted with a list of options he might want to add negated to the term in order not to block off the possibilities of further development too much.

4.2 COVST: the version Selection Tool

The main purpose of the COVST tool is to help the user select ambition and choice for his work. This will be done through a menu-based interface, which lets the user see what options are currently available for selection. COVST will also relieve him from the error-prone task of typing in the option names manually, as it has to be done now.
We use the term **selection** to refer to the ambition or choice which is in the process of being specified by the COVST user. We assume that the user already knows (this will be the task of other parts of the EPOS environment) which product he is going to work on, i.e. the software object representing the root of his product hierarchy.

### 4.2.1 Selection Menus

Selection of options (at least when done from scratch) is based on a hierarchical selection menu which at any time lists all **items** ready for selection. These items can be independent options, options on which others depend, options sets or aggregates.

The different kinds of items are marked with different fonts, colors or additional symbols; see Figure 2 for an example. Note that an option set may be an option with sub-options; this fact will of course also be noted.

If an independent option is selected, it is simply removed. If another item is selected, a **submenu** appears in which the user may select one or (except in the case of option sets) more other items. This operation may recurse, if the sub-menu contains non-simple options.

If the rest of the items on the sub-menu are put back on the parent menu, they will be indented to to indicate the fact that they are not “top level” items, or they may be “hidden” inside the parent item, which must be marked as “opened”. An option having sub-options is also marked as “opened” if it has itself been selected, but none of the sub-options have been.

Any item may be **unselected** at any time, in which case it is deleted from the menu and any selection already done “inside” it (see below) are deleted as well. The exception to this is the ambition, which is frozen when we are selecting the choice, and the parent ambition when selecting a sub-transaction ambition; these cannot be unselected.

The user can at any time request information (in practice, that means the descriptive text we plan to add) for any item on the selection menu.

COVST does not care about validities, only option constraints, and so its operation is the same for selecting ambition as for selecting choice. The natural way to proceed, is to first select the ambition, then freeze that selection and proceed to bind more options for the choice. The latter may be repeated if the user wants to change his current choice. A similar operation will be done when setting the ambition for a sub-transaction (sub-WS); the settings of the parent are fixed.

### 4.2.2 Menu Example

Figure 2 shows how the menu might look like at a few stages during the selection process, based on the options and rules form the example in Section 3.7.1. On the left is the menu at start-up, with only the “top-level” options available. In the middle is a fully expanded sub-menu structure, with no selections yet made. On the right, the options **sunos** and **menus** have been selected, **help** and **intel** unselected, and the **language** sub-menu closed.
Figure 2: Examples of a COVST menu during selection.
4.2.3 Checking against Validities and Stability

Before starting the actual selection process, the user will be allowed to specify which (if any) validities to evaluate the choice against. Before “submitting” the version selection, COVST calls the evaluator to check whether the selection is legal related to the stability or validity, and if it is, what additional option bindings, if any, have to be added.

If the selection is restricted by additional option bindings (at least for the ambition; may not be so important for the choice), the user will then be prompted with the result and asked if it’s OK. If it is, the transaction (workspace) will be started. If not, we may be able to provide alternatives.

The ambition and choice which will be registered as the one we used, and which may in turn be the basis for a future selection, are the ones explicitly requested by the user, before validities and stability were taken into account.

4.2.4 Re-using Previous Version Selections

COVST should remember the last selection the user made, since he is quite likely to want the same or something quite close to it for the next job. At start-up, he will be allowed to pick this directly, or to enter the selection tool with the last selection pre-made so he can make a few changes to it.

The user may also define his “base ambition” and “base choice” containing the most used options. By default, COVST will then startup with this selections pre-made.

In this base selection the user may refer to an aggregate option as such, so that the base choice will include all the part options of the aggregate at any given time, not just the ones that existed when the base choice was defined.

4.2.5 Product Dependent Options

At start-up, COVST will ask the user to specify the product root of his coming workspace (or may get that information from the work environment). Based on this, COVST can eliminate all options local to any product or sub-product outside this workspace root.

Note that when starting a workspace with product root $R$, an option $O$ depending on product $P$ can be:

- Used in the *ambition* only if $R$ is $P$ or a sub-product of $P$.
- Used in the *choice* also if $P$ is a sub-product of $R$. 

14
5 Implementation and Initial Experience

Implementation

Some of the version rules described in this paper have previously been implemented [GKY91] independently of the EPosDB implementation. Much of this will be reused in our new tools, especially the validity evaluation which is non-trivial.

The version rules will have to be stored into the EPosDB. We have written complete specifications for additions to the Prolog (ECM) interface for manipulating and operating all the different kinds of rules. The HiCoV tools will be built upon these.

After ECM became stable enough, we have started to use it for managing some own development, which was previously done with RCS. That includes this paper, where some parts (the description of version rules and tools) exist in two main versions: a short one for this paper and a longer one for an internal design document. Common changes to both can be applied with a ambition not binding the paper option which distinguishes the two.

We begun by using ECM for some stages of development and debugging of itself. Apart from some initial chicken-and-egg problems of this particular use of the system, it it has worked fine. However, we began running into some of the problems mentioned in Section 2.3. As part of our development process, we wrote down a list of how we would have used the HiCoV functionality if it had been available, and even from this rather small example we can see that it would have helped us.

CoV in Real Use

ECM consists of about 20 components. We performed six change jobs and introduced the same number of options. The first option "usable" identifies the upgrades needed to make the system stable enough that we could use it for installing itself and running the first change job. Four of the other options depend on this, while the sixth is independent.

Those four options all identify bug fixes, so it was natural to want to select all in the choice when doing later updates. Each one is also specific to one of the two sub-products of ECM.

Finally, ECM seems fairly stable after we did these fixes, so we would like to define the version choice including all as a validity. Thus, we can later check out according to this validity to retrieve a version to use, without getting not-yet-tested further developments.

If all the above had been modeled using HiCoV, the developer would have avoided these problems (which were real problems that actually occurred):

- Remembering and correctly typing in the option names; these had to be looked up elsewhere.
- Forgetting to add usable to the ambition when doing the changes that depended on it. With HiCoV, this would automatically be added to the choice and may in fact be optimized away from the ambition.
• While doing a bug fix to one of the sub-product, identified by a supposedly product-local option, a small error was discovered and fixed in the other sub-product. This was possible because the whole ECM had been specified as the product. The extra fix had to be redone later with an appropriate ambition. HiCoV would have prevented this from happening by not allowing the combination of option and product root.

• Forgetting to add all previous bugfix options to the choice. The result was that a buggy version was checked out, and testing of the new bugfix could not be done properly. Again, HiCoV would have helped us by use of the aggregate mechanism.

As the number of options increase, and we start introducing variability and the need for stability, we expect it to become increasingly difficult to enter the full ambitions and choices manually and correctly.

6 Conclusion and Further Work

6.1 The benefits of HiCoV

Though our experience so far is rather limited, we have already seen ourselves that we run into problems that HiCoV could have solved for us. We would of course have to specify the version rules (in our case, option dependencies and aggregates). This could be just a few extra parameters to the “create option” command. It would have been much less work than having to redo some changes because we had checked out a buggy version and thus could not test the system; this happened to us on one occasion.

From the examples in section 3.7 we see that HiCoV will be able to reduce considerably the number of options the user have to explicitly specify.

We can conclude that HiCoV will give a number of benefits to the CoV user:

• Navigation through the set of available options.

• Much fewer options (ideally only those the user are really interested in) have to be specified.

• Illegal or nonsensical combinations are prevented.

• Aggregates help the user getting “all those changes” without having to list them all.

• Stability will prevent accidental changes to versions which were supposedly fixed.

In addition, knowledge about option constraints may be utilized for internal optimization of CoV, further reducing the internal complexity⁶.

The only disadvantage we can see, is the added work for the user in establishing the version rules in the first place. In most cases, the corresponding rules will be known by the user who creates a new option, so it should not be much added burden. This is especially true if we

⁶Discussion of this is outside the scope of this paper.
allow an option to be created from within a sub-menu in COVST; then the appropriate rules
are given by the context.

6.2 Future Work

Our future work in this field consists of these main parts:

- Implementing the HiCoV as specified is of course our main task now. This consists of
two main parts: the support functions in the EPosDB which actually enforce the rules,
and the user interface tools described here.

- An initial test system, where the rules are not stored in the database, and not managed
by any HiCoV tools. The rules will be manually edited as Prolog statements in an
external file, and evaluated outside the database. This can be done quite quickly, and
may give us some further insight before we start the “real” implementation.

- Continued use of CoV for our own development, including a planned major upgrade of
the EPosDB.

- Evaluating previous history of the EPosDB source code, documentation etc. to see how
CoV and HiCoV could have been used.

- We hope also to be able to acquire some change history of real products from industry.
This may be even better input that our own code, which is mostly just sequential
revisions of one canonical variant.

We feel confident that the HiCoV extensions will make CoV much easier to use, both for
ourselves and in the future hopefully also for other users. We look forward to getting more
hand-on experience with CoV with help of these tools.

References

[adc90] Software Maintenance & Development Systems, Incorporated, Concord Mas-

[BEM93] Noureddine Belkhatir, Jacky Estublier, and Walcelio Melo. Software Process
Model and Work Space Control in the Adele System. In [Ost93], pages 2–11,
1993.

[Ber90] Brian Berliner. CVS II: Parallelizing software development. In Proceedings of

[CW95] Reidar Conradi and Bernhard Westfechtel. Configuring versioned software prod-
ucts. Submitted to SCM6, October 1995.

Software Configuration Management (SCM-4), Baltimore, Maryland, May 21–22,
1993.


