Change-oriented version descriptions in EPOS

Bjørn Gulla, Even-André Karlsson and Dashing Yeh*

Abstract

In EPOS,[1] software configuration management is based on the *change-oriented versioning* model. As part of a configured system specification, a *version description* is used to select the desired versions of different components that constitute the product. Furthermore version descriptions are used to specify which versions are affected by a certain change. For large software systems comprising many variants, building a consistent configuration meeting specified requirements is not an easy task. In this paper we propose a set of version description mechanisms and exemplify their use. An abstract description specifying desired properties and functional requirements is expanded using *validities* characterizing properties of versions, preferences and defaults. We believe that the proposed mechanisms provide simpler, more intuitive and more compact descriptions of versions of large-scale software systems than conventional tools.

1 Introduction

The problems of controlling the evolution of a large and complex software system are complex. One of the main problems is the selection, from a pool of versioned components, of those that are to be assembled into a consistent system with the required features, i.e. version control, or configuration management (CM). We describe some features that we believe a version control system should support, and illustrate how these are addressed in standard UNIX development environments (SCCS [1] and conditional compilation):

- selection based on property: in UNIX a specific version of a component is usually selected by a combination of a SCCS revision number and switches for conditional compilation. Usually, the compilation switches are consistently named across different components, and their names usually reflect a functional property. The revision number in SCCS is quite arbitrary and poorly reflects the properties of the version, as it is a combination of many changes. The selection of a consistent set of "latest" versions, on which to test ones own changes within an SCCS based system (in particular, one with branches), is problematic. The selection of versions on the basis of property is especially useful for people outside the development team, i.e. people running tests, sales people and customers.

- modular descriptions: compilation switches are named independently, with poor provision for grouping them together in hierarchies. For SCCS naming, it is the contrary; revisions are grouped after their time and branch of introduction, and rudimentary support is given to any other grouping. In addition, there is no support for recording or enforcing which revisions of different components and which compilation switches go together.

- support for parallel work: it is unrealistic to suppose that all work on a large software product is done sequentially, and the version control system should support parallel work on the same components. This is supported in SCCS by branches in the revision tree. The problem is selective propagation of changes between these branches. Users need to control when they will see parallel changes. The possibility to gradually including changes makes the difficult task of finding interference errors simpler.

*Bjørn Gulla and Even-André Karlsson are with the Division of Computer Systems and Telematics, Norwegian Institute of Technology, N-7034 Trondheim, Norway; and Dashing Yeh is with International Software Systems, 9430 Research Blvd., Echelon IV, Suite 250, Austin, TX 78759, USA. The paper was first received on 30th May 1990 and in revised form 27th March 1991.

[1] EPOS (Expert System for Program and System Development) is supported by the Royal Norwegian Council for Scientific and Industrial Research (NTNF) through grant ED0224.18457.

- specify scope of change: when a change is implemented, it may be understood that it affects certain versions, i.e. it is orthogonal to some changes and depends on other changes; whereas for some changes, it should not appear at all. In SCCS, a change will only appear on the leaf of the version branch where it is implemented, and it will be invisible in all other branches. A change implemented by a conditional compilation switch will be orthogonal to all other switches. Thus, changes are split into parallel variants (conditional compilation) and sequential revisions (SCCS).

- specify immutable versions: when a product is delivered, it must be possible to regenerate the delivered version to reproduce errors that the user encountered. This is currently done by freezing revisions in the SCCS tree. The exact revision of each components which made up the delivered system must be recorded separately.

In EPOS, we provide the user with CM facilities to control and maintain evolving versioned products or configurations. In the EPOS CM system we try to cover all the aspects of version control listed above. This is accomplished by the *change-oriented versioning* (COV) model [2, 3] COV can be most easily understood as an extension and generalization of conditional compilation. COV can be considered to be dual to conventional component-based versioning models, which we will call *version-oriented versioning* (VOV). COV unifies conditional compilation and revision changes, and keeps the versioning largely orthogonal to the module structure.

2 Change-oriented versioning

In this Section, we provide a short account of COV. Readers should see Reference [4] for a more complete description. Note that most COV concepts may be more easily understood if they are considered as *sets*, although they are represented as Boolean expressions.

In COV, the variation of a product is described by a set of *options*. Each option is a user-defined Boolean variable and is associated with a specific change in the external property of a product (i.e. a *functional change*). Options are global in a twofold sense; first, an option may be used by different users; and secondly, an option may involve changes in more than one component.

The basic unit of a versioned component is called a *fragment*[2]. Each fragment is tagged by a Boolean expression of options, called a *visibility*. A *version* of a component (or indeed the entire database) consists of all fragments with `true` visibitities. Such a version is determined by a value setting for all existing options, which is called a *choice*.

A choice does not need to specify the value of each existing option. A choice is *complete* if it contains enough settings that a unique version can be determined for the accessed part of the database. Usually, an incomplete choice corresponds to a set of versions. In what follows, the use of an unqualified *choice* always refers to a complete choice.

The set of all possible choices is called the *choice space*. Structurally, the choice space is an n-dimension space with two values in each dimension. A choice corresponds to a point in the choice space. An incomplete choice corresponds to a subspace with the dimension equal to the number of unset options. We say that two Boolean expressions overlap if the conjunction is satisfiable, i.e. the subspaces they define overlap. It is also useful to draw the Boolean expressions as Venn diagrams. When a new option is introduced, the choice space is expanded by adding a new dimension corresponding to the new option.

When changes are made to the database, the user must specify a 'scope' for the changes, i.e. which choices should be affected by the changes. This is specified as an incomplete choice called the *ambition*. By using the ambition in updating the visibilities of the fragments affected by the change, the changes appear in the desired versions. Choices outside the ambition will be unaffected by the changes (Figure 1.a). Explicitly, changes can be the insertion of a new fragment, the deletion or the modification of existing fragments. Deleting a fragment while working with ambition a will change the visibility v of the fragment to $v \wedge \neg a$ (Figure 1.b). Inserting a fragment gives a visibility a (Figure 1.c), and updating a fragment changes the visibility of the original fragment to $v \wedge \neg a$, and the changed fragment becomes a new fragment with visibility $v \wedge a$ (Fig-

[2]Depending on the data model, the components could be text files, relation tables or E-R entries, for example. The corresponding fragment sizes could then be a text line or a character, a relation tuple, and an E-R attribute.

ure 1.d).

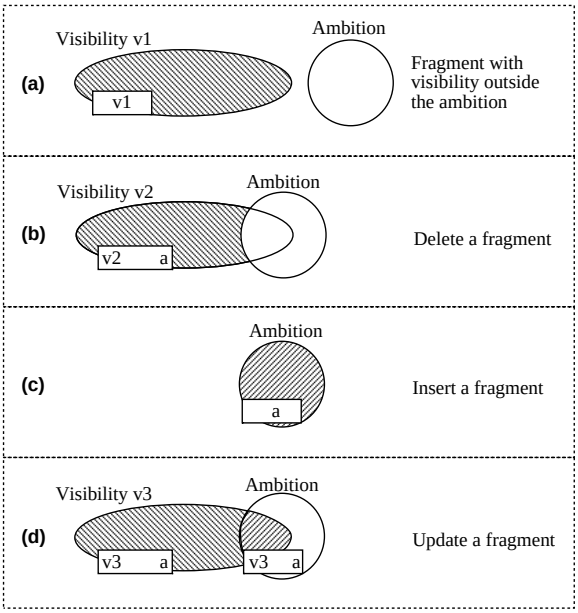

Figure 1: Updating of visibilities

Figure 2 shows how version descriptions are used in the EPOS software engineering database. When the user wants to retrieve some information, the choice defines a single version view of the database. A product description may further restrict this view, based on a selection of components. The ambition *a* is used when updating the database.

3 Version description mechanisms

In this Section, we describe the mechanisms for specifying choices and ambitions. It also introduces *validities* that are used to characterize certain sets of choices. One important use of validities is to make certain choices immutable (reproducible) or stable, i.e. assuring that no future changes affect these versions. In the following we use VeD as an abbreviation for version description.

3.1 Explicit setting of options

For small systems or during start up of a project, there is a limited number of options. The only mechanism needed is to be able to include or exclude certain functional changes, represented by setting the associated options to `true` or `false`,

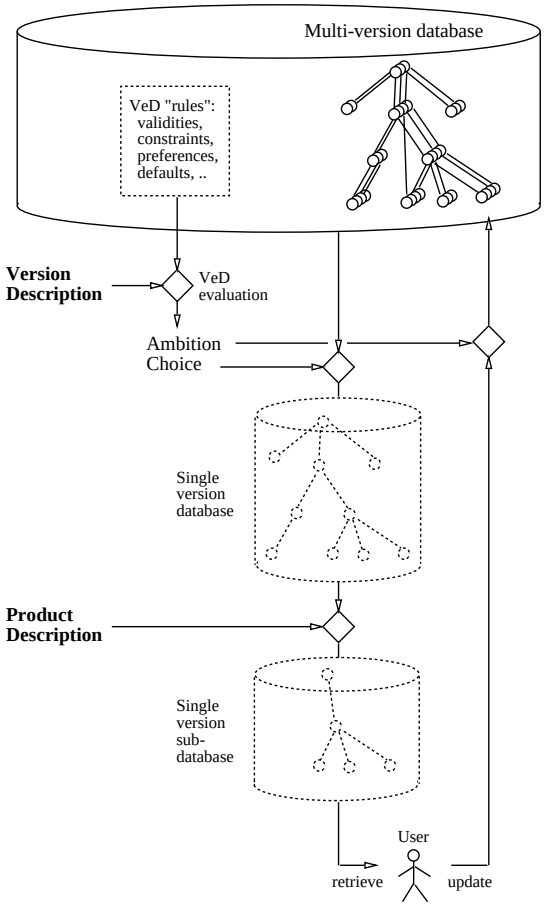

Figure 2: Use of configuration descriptions in the EPOS database system

respectively. In a VeD, this is accomplished by naming the option or its negation, for example

$$vms \wedge \neg bugfix3$$

This selects the *vms* version without *bugfix3*. By giving meaningful names to options, these descriptions can be quite intuitive.

The explicit setting of options can be used both to specify choices and ambitions. Aggregates (see Section 3.5) are provided to split up and name version descriptions.

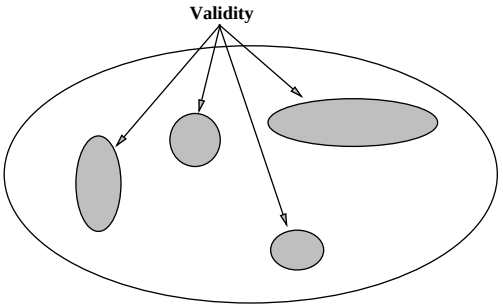

Figure 3: Example of a validity

3.2 Validities

Validities are introduced to specify that certain choices possess a particular property. In software engineering, validities are typically used to express status properties of versions of software components. The user may define a validity for each interesting status, e.g. experimental, compilable, component-tested, module-tested, integration-tested, and immutable. Validities can also be used to control access rights and define responsibilities in larger projects. A choice can belong to several validities.

3.2.1 Evolution of validities

When a user identifies a property that later may be used as a selection criterion, he may define a validity for that property. A meaningful name and optionally an initial (possibly incomplete) choice must be entered.

As work proceeds and a choice (complete or incomplete) is observed to have the property in question (e.g. satisfying some acceptance test), the choice is included in the validity. In this way, the validity increases to cover more choices. Thus the validity is a disjunction of choices (conjunctions), which is illustrated in Figure 3.

Whenever a new option (o) is introduced, all validities are automatically updated by conjuncting them with the negation of the new option. This means that only the original choices within the validity (i.e. with $o =$ `false`) are still valid. Corresponding choices with $o =$ `true` are *not* valid.

3.2.2 Use of validities

Validities can be used in version descriptions in two ways, positively or negatively. Using a validity positively means that the resulting choice must satisfy the validity. Using it negatively means that the resulting choice must not overlap with the validity.

An common example of a validity used in the negated form, is *stable*. All immutable versions (e.g. delivered to customers) must be included in *stable*. When initiating a change job to modify some part of the database, the VeD for the ambition must include the negation of *stable*. Otherwise, the new changes could also possibly be visible in versions intended to be immutable. Thus, validities can be used for access control supplementing the component-based access control of the DBMS in question, i.e. it imposes access control on the choice space of each component.

Positive validities are normally used to complete a choice. After setting the ambition, a choice *within* that ambition must be selected to retrieve a version. The validity is included to ensure that the selected choice has some specified properties. An example would be a validity called *module-tested*. This could be included if we wish to perform some modifications and want to start with a combination of changes that has reached a level of module-tested.

Ignore list

When using a validity in a VeD, an *ignore list* may also be included. An ignore list is a set of option bindings. During the evaluation of the validity, all occurrences of options mentioned in the ignore list are deleted.

The ignore list is interpreted as follows: try to fulfill the validity, but ignore the dimensions in the option space denoted by the options in the ignore list (Figure 4). The ignore list is frequently used to complete a choice from an ambition. To obtain a reasonable starting point, some validi-

ties expressing wanted properties can be included in the VeD. However, if the ambition binds some new options to `true`, just concatenating the ambition and the validities would not make a satisfiable VeD. Listing the new options in the ignore list would solve the problem and compute a combination of bindings for the *other* options that fulfill the requirements.

Unset options

We often start by specifying a maximum ambition and some negative validities. We let the evaluation algorithm reduce the maximum ambition by binding more options, so that it does not overlap with the specified validities. During this evaluation, bindings for options not mentioned in the VeD might be introduced. In some cases, it is useful to state that some options must be left unspecified by the evaluation process. A prefixing question mark (?) is used for an unset option.

This expresses the idea of a minimum ambition (Figure 5). If the resulting ambition does not include the minimum ambition, the result is not interesting.

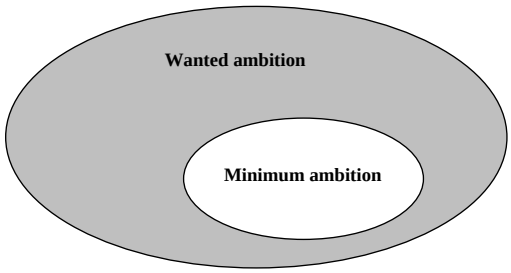

Figure 5: Minimum ambition

3.3 Constraints

Constraints are used to express static restrictions on combinations of options. We could have allowed arbitrary Boolean expressions as constraints, but we have found three types of constraints that seems intuitive and cover most practical cases. These are:

- at most one: this indicates that the options are mutually exclusive, i.e. at most one of them can be chosen. An example is the choice of operating system version, such as:

  $$vms \otimes unix \otimes pcdos$$

  (where $\otimes$ represents a mutually exclusive operator. Note, however, that all options may be `false`.)

- implication: an option requires the presence of other options and precludes the presence of others. A typical example is a feature or a bugfix that is dependent on some other feature(s), for example

  $$bugfix3 \Rightarrow unix \wedge \neg\ x11$$

  This constraint defines that *bugfix3* is only applicable together with *unix*, but not with *x11*.

- incompatible options: some combinations of options are not supposed to work together, i.e. we cannot simultaneously select all the options. An example is

  $$not\ (unix \wedge x11 \wedge bugfix8)$$

  which indicates that *unix* and *x11* can not be selected together with *bugfix8*.

Other types of constraints may emerge as we gain more experience with these mechanisms.

Constraints and validities differ, in that validities are automatically updated by conjuncting them with the negation of the new option, whereas, constraints remains invariant under introduction of new options.

3.4 Preferences

As mentioned in Section 2, evaluating a VeD is a search in the choice space. Heuristics are needed to guide the search. *Preferences* are essentially such heuristics, which allow the user to indicate where to search for solutions first. Preferences allow the user to give a looser version description.

Preferences are defined as small floating-point numbers, where positive values denote desired features and negative values denote unwanted features. The absolute value expresses relatively the strength of the desire; i.e. how much you want or do not want a given feature.

The default value for a preference is 0, i.e. neither preferred or unwanted. Usually, preferences are in the range between -1 and 1. Such preferences are called *sound*. In some contexts, it may also be legal to use *unsound* preferences, whose absolute value is greater or equal to 1. *Unsound* preferences express a desire or dislike strong enough to relax some global objectives

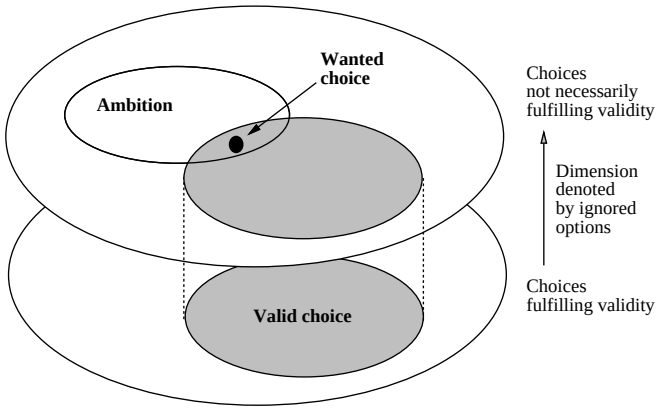

Figure 4: Choice completion with ignore list

(e.g. minimizing the number of introduced bindings).

The preference values are summarized below.

Range			*Interpretation*
	p	≤ -1	*unsound* unwanted
$-1 <$	p	< 0	*sound* unwanted
	p	$= 0$	default
$0 <$	p	< 1	*sound* preferred
$1 \leq$	p		*unsound* preferred

Both the preferred and the unwanted range may be associated to any single option: $0.5[\neg a]$ is a preference to bind a to `false`. $-1.5[a]$ means that you want to avoid setting a to `true`, although it may result in a larger number of bindings. $0.3[?a]$ means to mildly prefer that a is not bound to any value.

Sound preferences may also be combined with validities and aggregates.

3.5 Aggregates

Aggregates are named version descriptions. They allow modularization and reuse of existing descriptions. Aggregates are normally stored symbolically and are not evaluated before use (lazy evaluation). To obtain immediate evaluation, the *eval* operator can be applied to the VeD. Am example of an aggregate is

$$my\_unix = unix \wedge inher\_constr \wedge 0.5[tested]$$

Aggregates may include non-circular references to other named aggregates. When attaching preferences to aggregates, the semantics of this is defined as multiplying each term in the aggregate by the preference value. Only sound preferences are allowed.

The identifiers used in aggregates may also include wildcards and ranges to make the descriptions even more compact.

3.6 Defaults

To enforce project policies, e.g. to automatically include $\neg stable$ in all ambitions, defaults may be supplied in certain situations and contexts. In the EPOS environment there are mechanisms to support different kinds of defaulting. In our case, default values should be dynamically inherited along the project and task structure. These mechanisms are not discussed further, as they are regarded as a functionality of the underlying environment.

4 Examples

We provide below examples of the use of COV and VeDs.

4.1 Starting out

We presume that the product (p) initially consists of a collection of non-versioned components f_j (files). In our example, we can think of the components as ordinary files found in a file-based development environment.

The product is delivered to some customers, who all have the initial version. Thus we need to make this version *stable*, i.e. no ambition should overlap with the stable versions of p. Stable is defined as a validity which is given initial value

`true`, i.e. the initial version is stable. Stability is important to reproduce delivered versions for reproducing reported bugs.

The product is constructed as a hierarchy of subsystems, with modules at the bottom. Testing of p proceed bottom up. We associate different validities with each level of testing, i.e. module-tested v_m, subsystem-tested v_s and product-tested v_p. As the initial product has passed all these tests, all validities v_* are set to `true`.

We describe an abstract evolution of p, which consists of correcting a number of bugs b_i, and making a number of enhancements e_i. The correction of bugs will only affect existing components (possibly more than one); whereas the introduction of enhancements will possibly also create new components. These changes are functional changes, and each is denoted by a Boolean option with the same name as the bug (b_i) or enhancement (e_i).

4.2 Fixing the first bug

The first bug b_1 is discovered. Since we have delivered p to the customers, we do not want to overwrite the original text, and so we create a new option b_1 to identify the changes made when fixing this bug. A *choice* setting b_1 to `false`, i.e. $\neg b_1$, will exclude these changes and reproduce the original version delivered to the customers.

By introducing the new option, the validities are automatically extended with $\neg b_1$. This means that *stable* will change to $\neg b_1$. Thus, only the choice $\neg b_1$, which reproduce the initial version, is stable. The same applies for the tested validities v_*, which will be $\neg b_1$.

To actually fix the bug, we set an *ambition* for the changes. The ambition in this case is b_1 set to `true`, i.e. the VeD is b_1. Thus, the changes should only be *visible* when b_1 is set to `true` in a choice. This is achieved by updating the attached *visibility* of each fragment we change. Originally, all fragments have visibility `true`.

Thus, the ambition determines the scope of the changes made by this change job. Before making the changes, we also have to determine a *choice*, i.e. which version of the product do we want to see when we make the changes. Ideally this should be the whole ambition, i.e. we want to see everything that we affect, and for this ambition that is possible. When the choice space is larger, we will generally have to select one suitable choice where we start implementing the changes.

Thus, in our example the initial visibility of all original fragments is `true`. Our ambition is b_1, which is also our choice. Inserted fragments obtain visibility b_1, deleted fragments $\neg b_1$, the original copies of the updated fragments $\neg b_1$, and the changed copies b_1.

4.3 Testing the first bug fix

The modules affected by the changes need to be tested, to see if the changes corrected the error and that no new bugs have been introduced. As the fix passes the different tests, we update the corresponding validities, i.e. $v\_* = v\_* \vee b_1$.

4.4 The next bug

Yet another bug b_2 is found, and we introduce a new option b_2 to fix it. We could have fixed the bug under option b_1, but that would invalidate all the validities that we have updated.

The ambition for b_2 is set to $b_1 \wedge b_2$, i.e. we do not intend to select b_2 without also selecting b_1. Bug fixes are much like revisions; they are usually introduced in a sequence, one based on all previous ones. Parallel bug fixes are treated later. The choice must also be $b_1 \wedge b_2$, and we carry out the same procedure as for b_1. The tested validities will be updated to $v\_* = v\_* \vee (b_1 \wedge b_2)$ when the new code passes the corresponding tests.

To indicate that b_2 depends on b_1, we introduce a new constraint $depends = b_2 \Rightarrow b_1$.

We can now introduce our first aggregate, which we call *bug_fixes*, and it is going to provide all the bug fixes, i.e.

$$bug\_fixes = b_1 \wedge b_2$$

4.5 Four enhancements

We decide to enhance the product with another parallel version, e.g. another operating system. First, we introduce a new option e_1 to describe the necessary changes.

The ambition and choice for the changes needed to implement e_1 are not very exciting, as they will include all options set to `true`, i.e. $bug\_fixes \wedge e_1$. We perform the implementation as for a bug fix.

We want to enhance the product with a new parallel enhancement e_2, which is orthogonal to e_1. The ambition of the changes to implement e_2 must leave e_1 unspecified, i.e. we want them to be visible regardless of whether we choose e_1, thus the ambition must be $bug\_fixes \wedge e_2 \wedge ?e_1$. For

the choice, which determines what version we see of the product, we have to set the only free option e_1. We could also leave it unspecified if we have some means of editing a multi-version product. Let us suppose we use e_1; we make the changes and test the result. The tested validities are set to $v\_* = v\_* \vee (b_1 \wedge b_2 \wedge e_1 \wedge e_2)$. We can start a new change job with the ambition $bug\_fixes \wedge \neg e_1 \wedge e_2$ to make the additional changes needed to also implement e_2 for the choice $\neg e_1$. Note here that we need two separate change jobs to implement the functional change e_2.

A third enhancement e_3, which currently builds on e_2, has ambition $bug\_fixes \wedge e_3$, and we introduce a new constraint $temp\_depends = (e_3 \Rightarrow e_2)$. Thus, we have used constraints to record both temporary and more permanent properties. If we later decide to also implement e_3 for $\neg e_2$, we see all the changes made during the first implementation, and we can delete *temp_depends*. If we want to integrate b_2 without b_1, we have to start from the beginning because of the ambition used when implementing b_2.

The ambitions and choices used must be specified by the implementer. The ambition allows the implementer a tight control over the propagation of the changes. In particular, it must be considered if it is worth propagating changes, and integrate them in a second change job, or not propagating and start from the beginning in the second change job.

We can now introduce an aggregate to include all dependencies, $all\_dep = depends \wedge temp\_depends$.

A forth enhancement e_4 is orthogonal to e_1 and e_2, but a delivered product version can have at most one of e_1 or e_4 set to `true`. e_4 might be another operating system. To model this, we introduce a new constraint $one\_os = e_1 \otimes e_4$. Note that this constraint is only intended to affect future choices, not ambitions. It is still meaningful to fix a bug b_3 with ambition $bug\_fixes \wedge b_3$. These changes will be visible regardless of the choice for e_i. Any new operating system enhancement e_i should be included in *one_os*, i.e. $one\_os = one\_os \otimes e_i$.

As we test different combinations of e_1 to e_4, we update the corresponding v_*.

When we deliver a new version of the product to a customer, we must include the delivered choice in the *stable* validity. Now *stable* is just the initial version, i.e. $\neg b_1 \wedge \neg b_2 \wedge \neg e_1 \wedge \neg e_2 \wedge \neg e_3 \wedge \neg e_4$

4.6 Several parallel enhancements and bug fixes

We can envisage that several changes are carried out in parallel on the product. These changes are working with different ambitions, introducing new options, updating the validities and constraints as they proceed. It can be difficult for a new job to choose the right ambition and choice for the changes. Some examples of how the preceding constructs could be used to make this selection easier might be

- $e_1 \wedge e_5 \wedge ?e_2 \wedge ?e_3 \wedge \neg stable$
  We want to integrate e_1 and e_5, and we want to leave e_2 and e_3 unset. Furthermore, we want the result to not interfere with what is stable. This is a plausible specification of an ambition; and we call the resulting option bindings *amb*, i.e. $amb = eval(e_1 \wedge e_5 \wedge ?e_2 \wedge ?e_3 \wedge \neg stable)$

- $amb \wedge 0.5[bug\_fixes] \wedge one\_os \wedge [amb]v\_m$
  This specifies that we must have *amb* and that we prefer to have as many bug fixes as possible; we only want one operating system version; and the choice, when we ignore the options in the ambition, should have passed module testing. This is a plausible choice specification to find a version to do the changes on. If the ambition is a single option b_3, $[b_3]v\_m$ gives us a version to start with that was tested for $\neg b_3$, which is what we want.

- $e_1 \wedge e_3 \wedge all\_deps \wedge one\_os \wedge v\_p$
  Here we specify that we want enhancement e_1 and e_3, all dependencies, one operating system (e_1), and the result should be product tested. This could be an example of a new release, i.e. to be included in the *stable* validity and delivered.

The most beneficial uses of these concepts are to complete the ambition to give a desired choice with which to start implementing the ambition, and to choose a product with certain properties. The ambition is usually stated manually, but it is convenient to include $\neg stable$ by default in ambitions, to guarantee that we do not interfere with something that is delivered.

Working on two bug fixes in parallel is similar to implementing two orthogonal enhancements, which must later be merged. When a new option is introduced while other work is in progress,

the persons involved are notified and the option is negated into their choice, but the ambition still leaves the option unspecified. When we later want to integrate the two bug fixes, we set both to `true`.

This example just shows some uses of our version description possibilities.

5 Propagating changes

Uptil now we have only used options to describe functional changes, i.e. bugfixes and enhancements. We can also use options to control the propagation of changes between parallel change jobs, where ones ($u1$) ambition overlaps another's ($u2$) choice. In general, $u1$'s changes will be immediately propagated to $u2$. To avoid this, we can extend each change job with a new option. By using this option, other parallel change jobs can decide if they want the changes immediately propagated or not. These options can be included in an aggregate set to `true`, when there are no more parallel changes. Note that such non-functional options are also useful for canceling unintended change jobs.

6 The evaluation process

As mentioned earlier, evaluating a version description is a search in the choice space. The basic evaluation strategy is the same for both ambitions and choices. Our algorithm produces a *possible*, but not necessarily an optimal, solution, i.e. the least number of bound options for an ambition or maximum number of positive options for a choice. In each step, it tries to introduce as few new bindings for options as possible. Mandatory features must be fulfilled; otherwise, the version description is rejected. Preferred features guide the heuristics, i.e. order the search for a possible solution.

The following strategy is used: first, defaults are inserted. Aggregate definitions are inserted recursively (macro-like expansion) and preferences multiplied in. The result after ordering is a list of mandatory atoms and constraints, preferenced atoms, positive and negative validities, and preferenced validities. The rest of the algorithm contains one step for each mandatory validity. First positive validities are considered; then negative. Each step computes the minimal set of additional option bindings that ensures that the current validity expression is `true` or `false`, respectively. For each new binding, the constraints are checked. Preferences on atoms are used in the heuristics when selecting new bindings. If subsequent steps fail, the algorithm backtracks and an alternative set of bindings is computed.

6.1 Positive validities

A validity is a disjunction ($\vee$) of terms, where each term is a conjunction ($\wedge$) of factors. Each factor is either an option or a negated option. In order for the disjunction to be `true`, at least *one* of the terms must be `true`.

To find a minimal set of bindings that ensures that the validity expression is `true`, the mandatory bindings are first inserted. If the validity is not already `true`, the algorithm uses the preferences to picks one term to force to `true`. Corresponding new bindings are introduced. A term containing an option with a mandatory binding to `unset` cannot be selected. The evaluation fails (backtracks) if is not possible to fulfill the validity.

6.2 Negative validities

If multiplying in the negation, a negative validity is a conjunction ($\wedge$) of terms, where each term is a disjunction ($\vee$) of factors. To obtain `true` for the evaluation of the total conjunction, *all* terms must be `true`.

For each possible remaining option binding, the number of eliminated terms (matching occurrence) and the number of terms with fewer components (opposite occurrence) are easily computed. According to these numbers and to the given preferences, a weight is computed for each possible new binding. The bindings are sorted by these weights. To fulfill the validity, bindings are selected from this sorted list and possible terms eliminated. Options with mandatory bindings to `unset` may not be selected. This process is repeated until all terms are `true`.

An important special case is when the length of a term is one. In this case, a binding for this option *must* be introduced to fulfill the validity.

7 Comparison

7.1 SCCS + conditional compilation

The traditional way of handling a versioned system is to use conditional compilation to handle

variants (e.g. different operating systems, different devices, optional features) and SCCS to handle revision chains (e.g. bug-fixes, mandatory enhancements). This works well in a well planned, sequential development environment. If parallel work has to be done, the usual procedure is to create several branches in SCCS. If these branches are later to be merged, a difference tool (e.g. diff3) is run and the merged revision (possibly manually changed) is included in the SCCS graph. This has to be repeated in *some* sequence if there are several branches to be merged. However, no prehistory is kept in the new merged version.

The main disadvantage of this approach comes from the separation of variants and revisions, i.e. everything which does not fit nicely into these two categories creates problems. Two examples are

- an optional feature that depends on another optional feature (may be expressed if the conditional compilation condition can be a conjunction).
- a bug that only appears in one variant, without being local to the different code (we must include conditional compilation manually around all the changes, as we must do with the ordinary variants).

SCCS also lacks the concept of configuration descriptions.

7.2 Related Work

In an early work, Belady and Merlin [5] attempt to devise a formal model for evolving software systems, and they discuss how some actual systems used within IBM can be viewed as conforming to this model. An individual selectable unit (similar to a COV option) may implement some functionality or repair other units, and may span several modules. To meet specific operational requirements, a combination of units (a configuration) must be determined.

Their paper discusses the problem of expressing and efficiently representing permitted configurations. In the second part of the paper, a module is introduced as the fragment size for versioning. Since no visibility concept associated with each fragment is used, a complex global function expressing both fragment selection and composition must be represented. Different forms of this function, and the formidable problems of redefining it when new units are added, are discussed at some length. The paper does not address the problem of deriving the relations between the units and between the units and the module versions. The model presented is mainly a model of software as seen by the user or installation manager, not the developers. In this paper, we show how our approach also supports the development phase, including the creation of relations to describe useful configurations. Not only functional requirements, but also other properties (denoted by validities) may be used to describe the desired configuration.

In the PIE system [6], a layered network is used to allow alternative software designs to co-exist. Layers identify functional changes and may span several modules. A layer is either an alternative of another layers (placed in the same context) or independently selectable. However since the versioning granularity is Smalltalk methods, some of the advantages of combining individual functional changes are lost.

The work reported in References [7] and [8] is primarily concerned with editing and maintaining individual multi-version text files. Concerning versioning models, the papers contains notions very similar to those in COV. Specifically, a version is defined as a collection of fragments, where each fragment is associated with a conditional expression (*visibility*, such as (SYSTEM = UNIX) & (TIME > 1986). SYSTEM and TIME are multi-valued variables, called *dimensions*, somewhat similar to options in COV. There is also a notion comparable to ambition, which is called an *edit set*, but no validity concept. In their approach dimensions seem to be used to classify a set of existing versions, as opposed to our options which describe a space of **potential** versions.

Adele [9] is a CM system with a classic VOV model. A configuration is specified using selection predicates and constraints over attributes, which are attached to versions of components. The configuration selection is accomplished by satisfying selection predicates and constraints in a intermixed version/product selection traverse through the component structure.

8 Conclusion and further work

The mechanisms described in this paper are implemented in Prolog, and we have made some initial experiments with version descriptions. We are now working on a larger example, modeling the development history of a 500.000 line prod-

uct developed and maintained by 20 persons. We are also working on a partly automatic tool for transferring systems from traditional SCCS and conditional compilation-based development to be put under change-oriented versioning. A first version of a fully COV software engineering database (EPOSDB) has also been developed [4, 10].

9 Acknowledgments

The authors would like to thank the other members of the EPOS group, especially Reidar Conradi, for discussions and critical reading of drafts of this article. We would also thank the referees for their constructive comments and suggestions.

References

[1] Rochkind, M.J.: 'The Source Code Control System', *IEEE Trans. on Software Engineering*, December 1975, SE-1(4), pp. 364–370.

[2] Holager, P.: 'Elements of the Design of a Change Oriented Configuration Management Tool'. Technical Report STF44-A88023, ELAB, SINTEF, Trondheim, Norway, February 1988.

[3] Lie, A., Conradi, R., Didriksen, T.M., Karlsson, E.A., Hallsteinsen, S.O., and Holager, P.: 'Change Oriented Versioning in a Software Engineering Database'. *in* Tichy, W.F. (Ed.): Proc. 2nd Int. Workshop on Software Configuration Management, Princeton, (see also ACM SIGSOFT Software Engineering Notes, 17(7), November 1989, pp. 56–65).

[4] Lie, A.: 'Versioning in Software Engineering Databases'. PhD Thesis, Technical Report 1/90, DCST, NTH, Trondheim, Norway, January 1990.

[5] Belady, L.A., and Merlin, P.M.: 'Evolving Parts and Relations — A Model of System Families'. Technical Report RC-6677, IBM T.J.Watson Research Center, Yorktown Heights, NY, 1977, (see also Lehman, M.M., and Belady, L.A. (Eds.): 'Program Evolution — Process of Software Change', Academic Press, 1985, pp. 221–236).

[6] Goldstein, I.P., and Bobrow, D.G.: 'A Layered Approach to Software Design'. Report CSL-80-5, Xerox Palo Alto Research Center, California, (see also Barstow, D.R., Shrobe, S.E., and Sandewall, E. (Eds.): 'Interactive Programming Environments', McGraw-Hill, New York, 1984, pp. 387–413).

[7] Kruskal, V.: 'Managing Multi-Version Programs with an Editor'. IBM Journal of Research and Development, January 1984, 28(1), pp. 74–81.

[8] Sarnak, N., Bernstein, N., and Kruskal, V.: 'Creation and maintenance of multiple versions'. *in* Winkler, J.F.H. (Ed.): 'Proc. ACM Workshop on Software Version and Configuration Control', Grassau, Germany, January 1988, B.G.Teubner Verlag, Stuttgart, pp. 1–20.

[9] Belkhatir, N., and Estublier, J.: 'Experience with a data base of programs'. *in* Henderson, P.B. (Ed.): 'Proc. 2nd ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments', Palo Alto, (see also ACM SIGPLAN Notices, January 1987, 22(1), pp. 84–91).

[10] Odberg, E., Munch, B., Gulla, B., and Bratsberg, S.E.: 'Preliminary design of EPOSDB II'. DCST, NTH, Trondheim, Norway, September 1990.

A Syntax

version_descr	→	comp { '∧' comp }
comp	→	mand_comp
	|	weight '[' mand_comp ']'
weight	→	[**real**]
mand_comp	→	atom
	|	validity_usage
	|	**constr_name**
	|	**aggr_name**
atom	→	**opt_name**
	|	'¬' **opt_name**
	|	'?' **opt_name**
validity_usage	→	['¬'] [ignore_list] **val_name**
ignore_list	→	'[' term ']'
option_def	→	**opt_name**

aggregate_def → **aggr_name** '='
version_descr
| **aggr_name** '=' **eval**
'(' version_descr ')'

validity_def → **val_name** '=' expr
expr → term { '∨' term }
term → option_binding
{ '∧' option_binding }
option_binding → **opt_name**
| '¬' **opt_name**

constraint_def → **constr_name** '='
constraint
constraint → **opt_name**
{ '⊗' **opt_name** }
| **opt_name** '⇒' term
| **not** '(' term ')'

default_def → version_descr