Configuring Versioned Software Products

Reidar Conradi[1] and Bernhard Westfechtel[2]*

[1] Department of Computer Systems and Telematics
Norwegian Institute of Technology (NTH), N-7034 Trondheim
conradi@idt.unit.no
[2] Lehrstuhl für Informatik III, RWTH Aachen
Ahornstr. 55, D-52074 Aachen
bernhard@i3.informatik.rwth-aachen.de

Abstract. Despite recent advances in software configuration management (SCM), constructing consistent configurations of large and complex versioned software products still remains a challenge. We provide an overview of existing approaches which address this problem. These approaches are compared by means of a taxonomy which is based on an analogy to deductive databases: construction of a configuration corresponds to evaluation of a query against a versioned database with stored version selection rules.

1 Introduction

Consistently configuring a large product existing in many versions is a difficult task. Many *constraints* on combining revisions and variants must be taken into account. Frequently, these constraints are neither documented properly nor specified formally. Then, it is up to the user of a SCM system to select consistent combinations of versions. Furthermore, dependencies between changes must also be taken into account, e.g. bug fixes may require other bug fixes to be included. If the user commits an error, this may turn out only after testing the selected source configuration, or even by failures at the customer site.

This paper provides an overview of existing approaches to configuring versioned products. From the perspective of *deductive databases* [20], configuring a versioned product corresponds to evaluating a query against a versioned database with stored version selection rules. In section 2, a taxonomy is developed which is based on this analogy.

SCM systems are based on a wide spectrum of different *version models*. A version model defines the objects to be versioned, version identification and organization, as well as operations for retrieving existing versions and constructing new versions. This paper focuses on support for configuring versioned products from an intensional description. In our survey, we distinguish between two classes of version models which are treated in sections 3 and 4, respectively:

* This work was carried out during a research stay at NTH. Support from NTH is gratefully acknowledged.

- *Version-oriented models* describe configurations (i.e. product versions) in terms of explicit versions of components. Component versions are arranged in version graphs describing their evolution histories. A configuration is described by version rules which select appropriate versions with the help of attributes and history information. Constraints refer to consistent combinations of versions of different components (in general, only a small fraction of all potential combinations are actually compatible). Differences between versions are described by deltas which, however, are not referenced in the version rules.
- *Change-oriented models* describe configurations in terms of changes relative to some base configuration. There are no version graphs, just version rules describing potential combinations of changes. Changes differ from deltas in several ways: changes are named, they comprise logically related modifications to multiple components, and they can be applied in a flexible way (while a delta is tied to a specific pair of versions). In contrast to version-oriented models, change-oriented models do not maintain (and are not restricted to use) explicit component versions. Rather, new component versions are implicitly constructed by merging changes, resulting in higher flexibility than in version-oriented models. However, this combinability also has a drawback: inconsistent configurations may be produced easily. Therefore, constraints are required to express conditions on consistent change combinations.

At the ends of sections 3 and 4, a table is presented which summarizes the main points of the comparison. A final conclusion is given in section 5.

2 Conceptual Framework

This section defines the terminology used throughout the rest of this paper and lays the foundations for comparing specific approaches in the next sections. Let us start with some general definitions (along the lines of [27] and [2]):

- A *(software) object* (item) is any kind of identifiable entity put under SCM control. An object may be either *atomic*, i.e. it is not decomposed any further (internals are irrelevant to SCM), or *composite*. A composite object is related to its components by *composition relationships*. Furthermore, there may be *dependency relationships* between dependent and master objects.
- A *version* is an implicit or explicit instance of a versioned object which groups "similar" objects sharing some properties. We distinguish between *revisions* (historical versioning) and *variants* (parallel versions).
- A *configuration* is a consistent and complete version of a composite object, i.e. a set of component versions and their relationships. A configuration is called *bound* if it exclusively contains versions as its components. Conversely, an *unbound* configuration exclusively consists of versioned objects. A *partly bound* configuration lies in between.

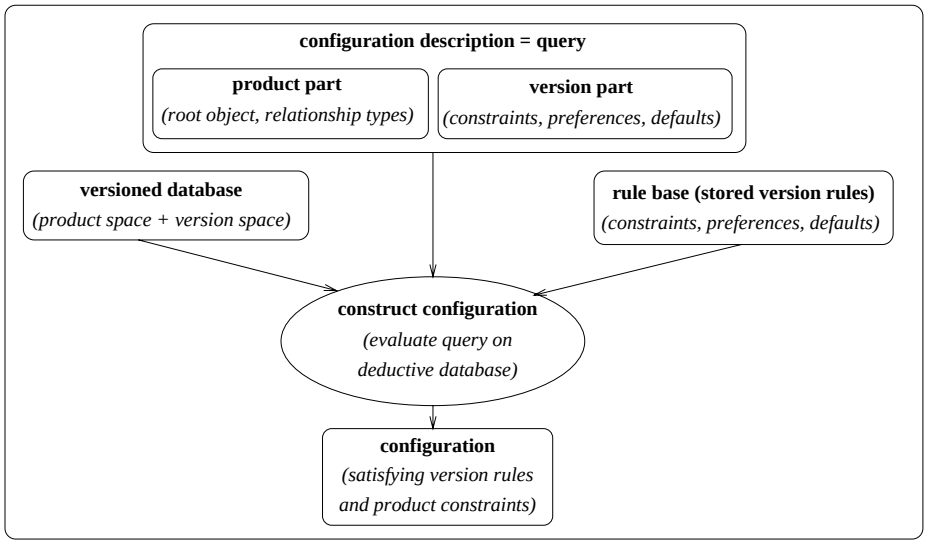

Fig. 1. Construction of a configuration from a deductive database perspective

Construction of a configuration may be regarded as a search problem on a *deductive database* (Fig. 1): A configuration description is a query which is evaluated against a versioned database augmented with a rule base for version selection. The configuration to be constructed is the result of evaluating the query.

The *versioned database* contains the versioned products to be configured. It consists of two parts:

- The *product space* is composed of the stored objects (items) and their relationships. Typical software objects (often documents) are requirements specifications, source files, relocatable files, user documentation, test data, review forms, measurement data, project plans, etc. The product space is organized by relationships between objects, e.g. composition relationships (`part-of`, e.g. a file is part of a directory) and dependency relationships (e.g. `include` dependencies in C programs or traceability dependencies between lifecycle documents).
- The *version space* is composed of version structures, e.g. version rules and/or histories that are characterized by version attributes with specified domains. Such attributes may be globally or locally defined.
  In version-oriented models, *version graphs* are used to describe the revisions (historical versioning) and variants (parallel versions) of software objects, annotated by given attribute values. The version space is an N-dimensional space, where one dimension corresponds to time and there is one dimension for each variant attribute.
  In change-oriented models, versions are described by *changes* which can be

dynamically combined ("merged") to produce specific versions. Each change, described by an attribute value, defines a dimension of an N-dimensional space characterizing all potential versions.

Before discussing the relations between product and version space more thoroughly later, let us pull forward an important point: In general, the product structure may depend on the versions selected; different versions may have different outgoing relationships (*versioned relationships*). Conversely, the version space may depend on the product space, i.e. the version attributes may depend on the part of the product to be configured. For example, there may be an attribute characterizing the style of the user interface, e.g. `Motif` or `OpenWindows`. This attribute does not apply to the database part of the product, which may e.g. be `Oracle` or `Ingres`.

A *configuration description* specifies the configuration to be constructed and consists of two parts:

- The *product part* describes the product as a composite object. This can be done by specifying one *root object* (or a set of such) and a set of *relationship types* (composition or dependency relationships). A product is constructed by a transitive closure from the root object following the specified relationship types.
- The *version part* contains the version selection rules, which constrain and guide the search (see below).

Further selection rules are contained in a *rule base* of stored rules. The rule base exists independently from the query. We may distinguish between the following kinds of *version rules* (this classification applies both to the rule base and the configuration description):

- A *constraint* is a mandatory rule which must be satisfied. Any violation of a constraint indicates an inconsistency, e.g. the `SUN` and the `VAX` variant must not be selected simultaneously.
- A *preference* is an optional rule which is only applied when it may be satisfied. It corresponds to a property which is preferred but not enforced, e.g. `released` module versions are preferred.
- A *default* is also an optional rule, but it is weaker than a preference. A default rule is only applied when otherwise no unique selection could be performed, e.g. the latest version of the main branch in a version graph may be selected by default.

A tool which constructs a configuration by evaluating a configuration description with respect to a versioned database and a rule base is called a *configurator*. Note that we intentionally avoid the notion of builder here, since we are only concerned with source configurations (no classical builds using tool-based rederivations are considered).

We use *AND/OR graphs* [25] both for describing the structure of a versioned product and for explaining how version selections are performed during the configuration process. An AND node represents a product part. When an AND node

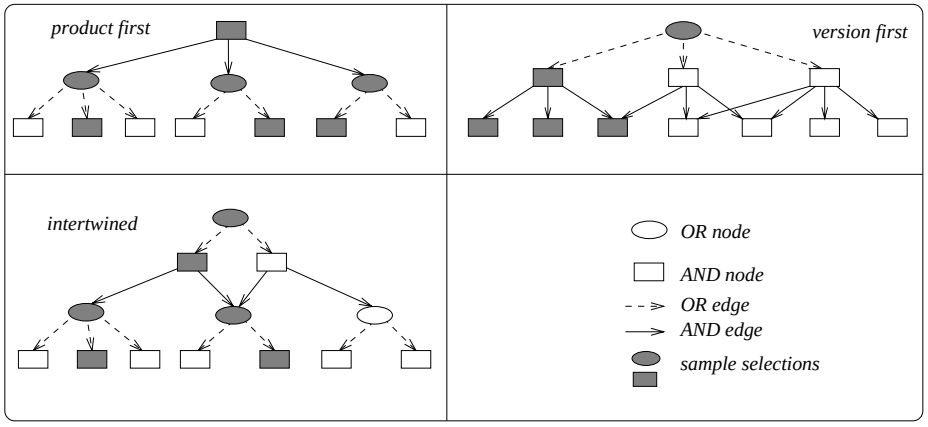

Fig. 2. Different kinds of AND/OR graphs and selection orders

is selected, all outgoing edges (relationships) need to be followed for constructing a configuration. An OR node represents a version group from which one member must be selected.

The following kinds of *selection orders* may be distinguished (Fig. 2):

Product first. First, the entire product structure is selected (AND selection). In this case, we have a composite object consisting of a known set of components. Each component may expand into multiple versions (OR selection).

Version first. This is the inverse approach: The product version is selected first (OR selection). Subsequently, an AND selection is performed to obtain all components belonging to the selected version. In general, the set of components depends on the version selected.

Intertwined. Product selections are intermixed with version selections (multi-level selections). Often, the AND/OR graphs are structured such that AND and OR selections alternate (bipartite AND/OR graph).

During the configuration process, non-deterministic selections (e.g. based on defaults) may need to be performed. These selections can be performed either automatically (*automatic configurator*) or with user assistance (*interactive configurator*). In either case, a non-deterministic choice may be wrong, and the configurator must therefore be able to *backtrack* from wrong selections.

Furthermore, configurators may be classified according to when references to versioned objects are bound to specific versions. *Static binding* means that a mapping is constructed beforehand. In case of *dynamic binding*, the configuration description is evaluated only when an object is actually accessed.

We still have to define *consistency*: A configuration is consistent when it satisfies the constraints (mandatory conditions) both in the configuration description and the rule base. In this paper, we concentrate on *version-level consistency*, i.e.

versioned database	
selection order	product first, version first, intertwined
product space	files, modular programs, ...
version space	revisions, variants, changes, ...
configuration description	
formalism	SQL-like queries, boolean expressions, ...
rule classes	constraints, preferences, defaults
rule base	
formalism	(see above)
rule classes	(see above)
configurator	
binding modes	static, dyamic
degree of automation	automatic, interactive
backtracking	yes, no

Table 1. Taxonomy for comparing configurators

on constraints describing legal version combinations. Typical examples are "select the same operating system variant throughout the whole configuration" or "include change `c` together with all changes on which `c` depends". We do not address *product-related consistency*, which must also be satisfied by a consistent configuration. In particular, the configurators described in this paper do not guarantee syntactic or semantic consistency of the result of a configuration process. Product-related consistency has been treated in module interconnection languages and systems which are based on such languages, e.g. SVCE [10], NuMil [18], and Inscape [19].

The framework developed above applies to both version-oriented and change-oriented models. However, there are some specific points which are discussed below:

- In version-oriented models, version rules refer to versioned components. In change-oriented models, (versions of) components are not mentioned in the version rules.
- In change-oriented models, the *version* (of the whole product) is always selected *first*. In version-oriented models, any selection order can be applied.
- In version-oriented models, *explicit versions* of components are used to construct configurations. For a product consisting of m modules existing in v versions, there exist v^m potential configurations, i.e. the number of potential configurations grows polynomially in v. The actual number may be very large, e.g. 5^{1000} for 1000 modules each of which exists in 5 versions.
- In change-oriented models, construction of a configuration does not rely on explicit component versions. Rather, *versions* of components are constructed *implicitly* by merging changes at a fine-grained level, e.g. text lines rather than files. In case of unconstrained combination of changes (each change

Product `foo` supports different operating systems (e.g. `DOS`, `Unix`, `VMS`), window systems (e.g. `X11`, `SunView`, `Windows`), and database systems (e.g. `Oracle`, `Informix`, `dbase`).
There are some straightforward constraints among these variants, e. g. `dbase` is not available under `VMS`, or `X11` does not run under `DOS`.
Furthermore, various changes are performed during maintenance of `foo`, such as e.g. bug fixes which are denoted by `Fix1`, `Fix2`, etc. Changes may be mutually exclusive; they may also depend on each other.
Finally, versions of (components of) `foo` have states indicating their degree of consistency (e.g. `coded`, `tested`, `released`).

Fig. 3. Running example

may either be applied or skipped), there are 2^v potential configurations for v changes, i.e. the number of potential configurations even grows exponentially in v. For example, for 1000 changes there are 2^{1000} potential combinations.

Table 1 shows a taxonomy which we will use for the comparisons in sections 3 and 4. The taxonomy is structured according to Fig. 1. Fig. 3 shows a small running example which will be used in sections 3 and 4, as well.

3 Version-Oriented Models

In the following, we survey version-oriented configurators. Each of them is based on some sort of *AND/OR graph* which is used to represent the versioned product to be configured. The AND/OR graph is traversed starting from some root node, with version selections performed at OR nodes.

3.1 RCS

RCS [26] is a successor of SCCS [21] and manages versions of text files. RCS selects the *product first* and thus cannot express versioned relationships. A version is a member of a *version group* which is contained in some directory. The members of a version group are arranged in a *version graph*, which consists of several branches (variants) each of which is composed of a sequence of revisions. An example is given in Fig. 4, where our sample product `foo` is represented by a directory containing version groups for the source files and the make file[3].

Configuration descriptions take the simple form of options supplied to a `checkout` command. Options describe *preferences* in terms of values of built-in version attributes. For example, the following command retrieves the latest versions with state `Released` which were created before the specified date:

[3] For the sake of convenience, a version group is represented by an oval surrounding its members.

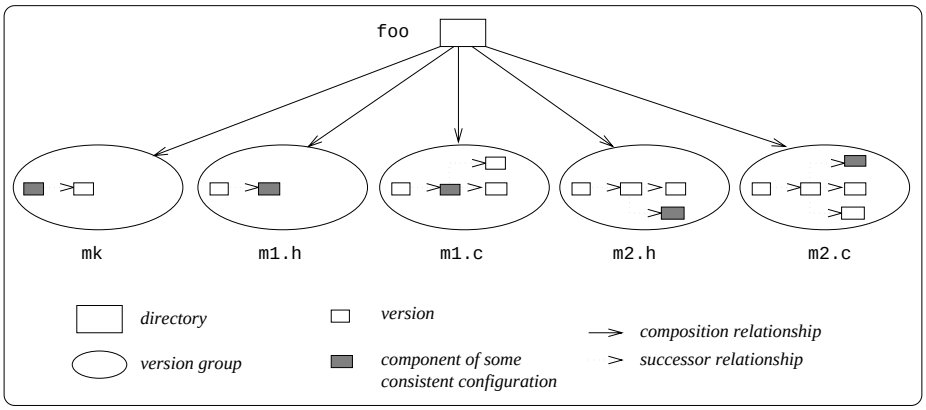

Fig. 4. RCS

```
co -d''1995/05/05'' -sReleased *,v
```

RCS does not support a rule base for version selection. Consistent version selections are difficult to perform. A consistent configuration may consist of versions which are located at different places of the respective version graphs (see the grey boxes in Fig. 4). A configuration may be regarded as a *thread* through the version graphs. Such a thread may be represented by tagging all versions with the same symbolic name.

For each version group, the `checkout` command selects the latest version satisfying all options. If no matching version exists or no options are specified, the latest version on the main branch is selected by default.

3.2 ClearCase

ClearCase [12], a successor to DSEE [13], adopts like many other SCM systems the SCCS/RCS approach to versioning of individual files. In addition, directories may be versioned, e.g. directory `foo` in Fig. 5). All kinds of versioned objects (files/directories) are uniformly denoted as *elements*. In contrast to RCS, ClearCase selects the *version first*. A single-version *view* is established on the versioned file system by a configuration description. The view is a filter which dynamically binds generic references to specific versions. The filter is used for both read and write accesses. It excludes components which are not used in the specified configuration (`m3.h` in Fig. 5).

A configuration description consists of a *sequence* of *rules*. There is no distinction between constraints, preferences, and defaults. The product part of a rule describes its scope, which may be a specific element, all elements in a certain subtree, or simply all visible elements. The version part is a boolean expression which refers to version numbers, branches, or values of version attributes. For example, in the configuration description

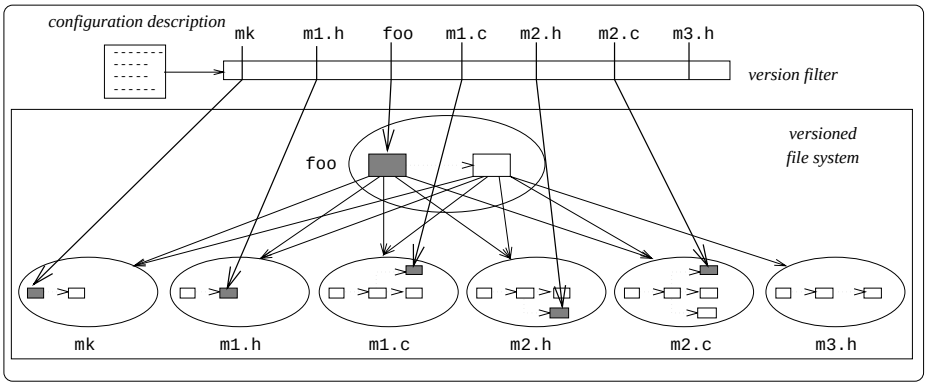

Fig. 5. ClearCase

```
element /foo/... /X11
element /foo/... /Oracle -time 10-Oct
element /foo/... /Unix   -time 10-Oct
element /foo/... /main   -time 10-Oct
```

`/foo/...` denotes the product part (subtree), `X11`, `Oracle`, ... specify branches, and a cut-off date is given by the `time` option[4].

ClearCase does not support stored version rules in addition to the configuration description. In particular, there is no way to state constraints, e.g. "`X11` and `DOS` are not compatible".

An operating system process, e.g. for compiling and linking a program, is executed under some configuration description which is supplied at process start. The configuration description is evaluated only when an element is accessed. Rules are evaluated sequentially, the ordering of rules indicates their priority. Evaluation stops when a unique match is found or all versions returned by a query reside on the same branch. In this case, the latest version on that branch is selected. An error is reported when no matching rule has been found.

3.3 CONFIG

CONFIG is a language-independent approach for configuring *modular programs* [28]. Versions of modular programs are represented by a bipartite AND/OR graph as given in Fig. 2. Accordingly, CONFIG performs *intertwined selections*.

CONFIG distinguishes between *revisions* and *variants*. A modular program evolves into a set of revisions which are denoted by natural numbers below. The structure of the evolution history is irrelevant to CONFIG. Each program revision may exist in multiple variants. Variants are characterized by attributes.

[4] This description can be used to perform changes specific to `X11` in a stable work context.

A *configuration description* is a pair *(re,va)*, where *re* denotes a revision and *va* a tuple of attribute values. For all attributes characterizing a program variant, a value must be specified; incomplete specifications are not allowed. For example,

```
(1, (os = DOS, ws = Windows, db = dbase))
```

selects a PC variant of the first revision of our sample program `foo`.

The rule base consists of two parts. First, for each version of a module a *version description set*, denoted as *VD*, is maintained which consists of pairs such as described above. *VD* indicates to which configurations the module version belongs. The version description sets of different versions of the same module must be mutually disjoint. For example, a version of the `main` module may have the following version description set:

```
{(1, (os = DOS, ws = Windows, db = dbase)),
 (1, (os = Unix, ws = X11, db = Oracle)),
 (2, (os = DOS, ws = Windows, db = dbase)),
 (2, (os = Unix, ws = X11, db = Oracle))}
```

According to lines 1 and 2 (3 and 4), this version is used in both the `DOS` and the `Unix` variant of program revision `1` (`2`).

Second, each version puts constraints on the versions of modules on which it depends. These constraints are attached to dependencies. A constraint is defined by a *mapping* from the version description *vd* of the current module to a version description *vd'* of a used module. In the simplest case, this mapping is the identity, e.g.

```
USE m VERS = (SAME, SAME)
```

propagates the current version description to the used module `m`.

The configuration process starts with a complete configuration description *vd* (=*(re,va)*, see above) at the root module. *vd* uniquely selects a version, and new version descriptions are computed for selecting versions of used modules. Since the version description sets of different module versions are mutually disjoint, the configurator operates in a deterministic way (no non-deterministic choices and hence no backtracking). If a module is used by multiple modules, the first selection wins, and inconsistencies between different selections are not detected.

3.4 Adele

Like CONFIG, the Adele configuration manager [4, 5, 6] performs sophisticated, *intertwined* product and version *selections* when configuring a modular program. In Adele, modular programs are organized into *families*. The structure of a family and relationships between families are depicted in Fig. 6[5]. Each family may

[5] For the sake of convenience, we make use of hybrid AND/OR nodes to represent interfaces and variants.

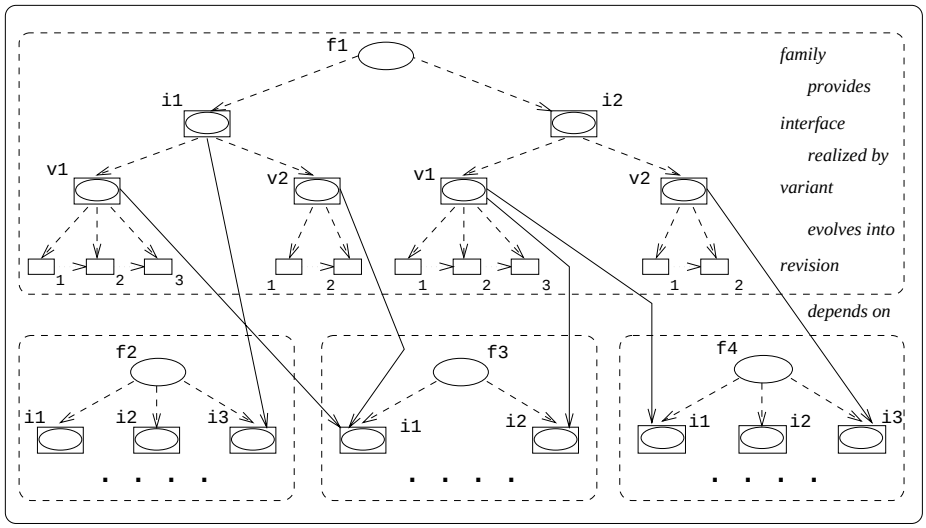

Fig. 6. Adele

have multiple interfaces, realized by alternative variants evolving into sequences of revisions. Note that realization variants may be both module bodies and previously constructed subconfigurations. Dependencies emanate from interfaces and variants, and they end at interfaces.

Adele supports both configuration descriptions and stored version rules, the latter of which are attached to families. Configuration descriptions and stored version rules are specified using the same language. *Version rules* are similar to those used in ClearCase. The product part of a rule defines its scope, which may be a specific family or a set of families below some root family. The version part consists of a boolean expression over version attributes, e.g.

```
([os = Unix] and [ws = SunViews]
 and [db = Informix] and [state = tested])
```

Unlike ClearCase, Adele explicitly distinguishes three different classes of rules, namely *constraints*, *preferences*, and *defaults*. Using these rules, short intensional descriptions can be given for large and complex configurations.

Given a configuration description for an interface, the *configurator* traverses the AND/OR graph of families, selecting one revision for each family. The configurator maintains a set of version rules V initialized with the configuration description. V is extended gradually by adding the stored rules attached to selected versions. To avoid wrong choices, version selection for some family `f` is performed only after all constraints are known. This is the case when versions of all families above `f` have been selected. However, once a selection has been made (possibly with the help of preferences and defaults), it cannot be retracted (no backtracking).

3.5 SIO

SIO [1, 11] is a SCM system which extends *relational database technology*. SIO selects the *product first*. A product consists of a fixed set of versioned modules. Each module is represented by a RDBMS-like relation, where each tuple corresponds to a single version. A version is characterized by attributes (fields of the tuple). The set of fields may vary from one module to another.

Configurations are described in an SQL-like manner. A query selects one version from each module. SIO distinguishes between *constraints* and *preferences*. Preferences act as filters on query results and are applied only when the outcome of filter application is not empty. Preferences may be ordered sequentially, resulting in sequential filtering according to user-defined priorities. For example, the following query specifies a configuration for `Unix`, `X11`, and `Oracle` (constraints), with versions in state `released` preferred over those in state `tested` (preferences):

```
select the instances of FOO having
   the versions of all the modules having
      os = Unix and ws = X11 and db = Oracle
from which prefer those having
   the versions of all the modules having state = released
from which prefer those having
   the versions of all the modules having state = tested
```

The rule base contains constraints which are specified by *compatibility rules*. A compatibility rule is an assertion in a restricted first order predicate calculus. The conditions under which two versions from different modules are compatible are stated in terms of version attributes. Constraints which quantify over (versions of) all modules are not supported. Due to the restricted form of constraints, SIO can efficiently check for contradictions between them.

A configuration description is evaluated against a database of module relations, taking compatibility rules into account. The *query evaluator* is a deductive component which goes beyond conventional database technology. In addition to checking the compatibility rules when constructing a new configuration, SIO analyses whether existing configurations would satisfy a new compatibility rule. Modification of the rule base is disallowed if it makes any existing configuration inconsistent.

3.6 NORA

ICE [31, 30], the SCM system of the software development environment NORA, is based on *feature logic*. A feature denotes a property of some object, e.g. the feature `os` denotes the underlying operating system. In its simplest form, a *feature term* `q` consists of a list of pairs of features and values, e.g.

```
q = [os : Unix, ws : X11, db : Oracle]
```

Feature terms are used both for configuration descriptions and stored rules. `q` serves as an example of a configuration description (query). All feature terms describe *constraints*; preferences and defaults are not supported. Versions of some object are represented by feature terms containing a special `object` feature whose value is an object identifier. For example, the following terms denote two versions of some user interface module and database module, respectively[6]:

```
ui1 = [object : UI, ws : X11, os : Unix]
ui2 = [object : UI, ws : Windows, os : DOS]
db1 = [object : DB, db : dbase, os : DOS]
db2 = [object : DB, db : Oracle, os : {VMS, Unix}]
```

The feature term of a configuration is constructed by an intersection operator (*unification*) which handles the `object` feature in a special way (union of the values instead of intersection). An intersection fails if corresponding features cannot be unified (mutually inconsistent constraints). For example, `ui1` may be combined with `db2`, but not with `db1`:

```
c1 = ui1 ⊓ db2 =
   [object : {UI, DB}, ws : X11, os : Unix, db : Oracle]
c2 = ui1 ⊓ db1 = ⊥
```

NORA supports *incremental construction* of a configuration with constraint-based guidance. Features can be specified step by step by selection from menus (rather than simultaneously as in the query `q` above); inconsistent choices are disabled by NORA. In our example, selecting `Unix` as operating system uniquely identifies the configuration `c1`, eliminating the need for further choices.

As used above, NORA may be classified as a version-oriented approach. However, it is interesting to note that feature logic can be applied to SCM in a more general way. In [30], Zeller proposes a unified version model for SCM which covers both version-oriented and change-oriented versioning. In particular, changes are modeled as features which can be either included or omitted (see also COV in the next section).

3.7 Summary

Table 2 summarizes the approaches described in this section. Different selection orders are used (version first, product first, intertwined). The product space is composed of files, components (with no assumption about object contents), or modular programs, either flat or hierarchical. The formalisms for describing version rules differ widely, ranging from command options to fully developed query languages. Only Adele and NORA represent configuration descriptions and stored rules in a uniform way. Except ClearCase, all approaches perform static binding. Only NORA and Adele support interactive construction of a configuration. Finally, not all approaches supporting constraints can backtrack from wrong choices (CONFIG even excludes non-determinism at all).

[6] In `db2`, the feature `os` is set-valued, which is indicated by braces.

	RCS	ClearCase	CONFIG	Adele	SIO	NORA
vers. db.						
selection order	product first	version first	intertwined	intertwined	product first	product first
product space	flat files	file hierarchy	flat modular programs	hierarchical modular programs	component hierarchy	nested components
version space	version graphs	version graphs	revisions variants	variants revisions	version sets	version sets
conf. descr.						
formalism	checkout options	first-order expressions	attribute tuple	first-order expressions	extended SQL	feature terms
rule classes	preferences	priority-ordered rules	constraints	constraints preferences defaults	constraints preferences	constraints
rule base						
formalism	—	—	attribute functions	first-order expressions	first-order expressions	feature terms
rule classes	—	—	constraints	constraints preferences defaults	constraints	constraints
configurator						
binding modes	static	dynamic	static	static	static	static
degree of automation	automatic	automatic	automatic	automatic interactive	automatic	interactive
backtracking	—	—	no	no	yes	yes

Table 2. Comparison of version-oriented approaches

4 Change-Oriented Models

In the following, we survey change-oriented configurators. While version-oriented configurators are based on AND/OR graphs, change-oriented configurators rather origin from *conditional compilation*. Conditional compilation addresses the multiple maintenance problem by storing multiple versions in a single file and using preprocessor statements to control the visibilities of fragments (sequences of text lines). Editing source files with many embedded preprocessor statements may become very confusing. Therefore, all change-oriented approaches described below automate management of visibilities and hide the corresponding control expressions from the users who are offered a single-version view on the versioned database.

4.1 PIE

An early approach to change-oriented versioning has been developed at XEROX PARC [7]. PIE manages configurations of Smalltalk programs which are internally represented by graph-like data structures. Each change is placed in a *layer*. Layers are aggregated into *contexts* which act as search paths through the layers[7].

When constructing a context, there are two degrees of freedom: First, each layer may either be included or omitted; second, the layers included can be arranged in any sequential order. This combinatorial complexity can be overcome by defining contexts in terms of other contexts (*aggregates*) which contain consistent and reusable combinations of layers. Furthermore, PIE provides *relationships* to document constraints for the combination of layers. For example, `A depends on B` implies that each context containing `A` should include `B`, as well. Conversely, `B repairs A` indicates that a bug in layer `A` has been fixed in layer `B`. Therefore, `B` should be included whenever `A` is selected. However, PIE does not enforce any constraints. Rather, the documented relationships are merely used to warn the user of possible inconsistencies.

4.2 Aide-de-Camp

Aide-de-Camp [3, 23] describes versions of products in terms of *change sets* relative to a base version. A change set describes a physical change which may affect multiple files. The finest grain of change is a text line. In contrast to layers, change sets are totally ordered according to their creation times. If change set `c1` was created before `c2`, `c1` will be applied before `c2` when both change sets are included in some product version. Each change set may be viewed as a switch which can either be turned on or off.

Aide-de-Camp detects *physical conflicts* when a change set is applied to some product version. A conflict occurs when a modification included in the change set refers to text lines which are not part of that product version. Furthermore, Aide-de-Camp provides a 3-way *merge tool* which combines alternative versions with respect to a common base and detects contradictory modifications to the same text lines. Unlike PIE, Aide-de-Camp does not support relationships which can be used to detect inconsistent combinations of change sets.

4.3 MVPE

MVPE [22] is a text editor which supports simultaneous editing of multiple versions of a text file. A text file consists of a collection of fragments (sequences of words). To each fragment, a *visibility* is attached which determines the versions to which the fragment belongs. A versioned file may vary along multiple dimensions. The version space is modeled as a table whose columns correspond to these dimensions and whose rows represent specific versions (Fig.7).

[7] DaSC [15] is based on a similar approach.

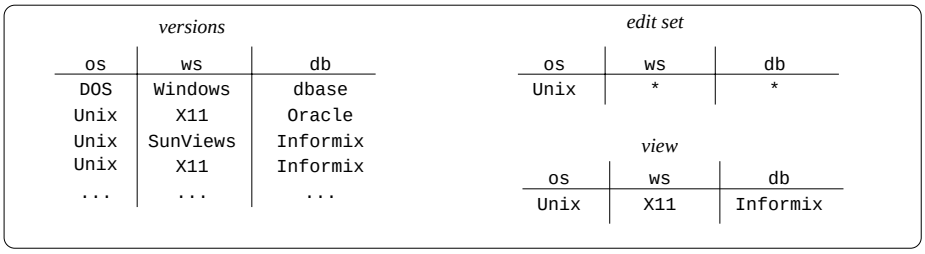

Fig. 7. MVPE

MVPE distinguishes between an edit set and a view. An *edit set* is a write filter controlling which versions will be affected by a change. The edit set is specified in a query-by-example style with the help of regular expressions (e.g. the edit set in Fig. 7 denotes all `Unix` versions). A *view* is a read filter and selects from the edit set a single version which is displayed to the user[8].

MVPE does not support stored version rules. In particular, there is no way to state constraints on combinations of different dimensions.

4.4 COV

Change-oriented versioning [14, 17] emphasizes management of *logical changes*. The version space is structured by global *options*. Each option defines a dimension of the version space. Options may denote variants, e.g. `Unix`, `DOS`, and `VMS` define different variants of the underlying operating system. Options may also be used to represent changes, e.g. `Fix1` and `Fix2` represent certain bug fixes. An option may be bound to either `true` or `false`, or it may be left unbound. A set of option bindings corresponds to a region of an N-dimensional version space.

A versioned database consists of a collection of *versioned fragments*. An example is given in Fig. 8 (see the box "database before update" on the left-hand side). The sample database consists of three fragments, denoted by `f1`, `f2`, and `f3`. While there is only a single version of `f3`, `f1` and `f2` both exist in two versions, denoted by `f11`, `f12`, and `f21`, `f22`, respectively. To each fragment version, a *visibility* is attached. The visibility of a fragment version is a boolean expression describing the product versions to which the fragment version belongs. For example, the visibility of `f11` is `Unix`. This means that `f11` is contained in all `Unix` versions, regardless of the window system selected.

Like MVPE, COV has been designed for multi-version editing and distinguishes between a read filter, called *choice*, and a write filter, named *ambition*. Changes are performed in transactions, which have both an ambition and a choice. Both ambition and choice are sets of option bindings. The choice extends the ambition, i.e. all option bindings of the ambition are included in the

[8] Note that ClearCase, which also sets up a view on a versioned database, does not distinguish between read and write filter.

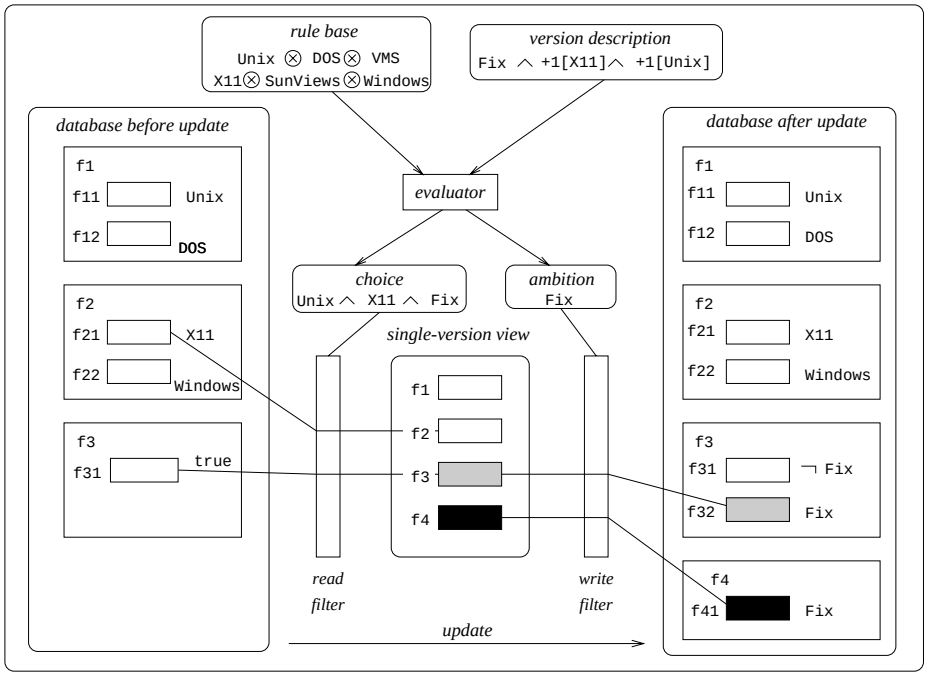

Fig. 8. Change-oriented Versioning

choice. The choice corresponds to a single point in the version space. This point is contained in the region corresponding to the ambition.

For example, in Fig. 8 the choice is

$$\texttt{Unix} \wedge \texttt{X11} \wedge \texttt{Fix}$$

In particular, this means that the change is performed in a version which runs under `Unix` and makes use of the `X11` window system. The ambition is set to `Fix`. This indicates that a bug fix is applied, being valid regardless of the operating system and the window system. The scope of changes is reflected in the visibilities which are assigned to updated or inserted fragments. In Fig. 8, `f3` is updated and `f4` is inserted. Fragment versions `f32` and `f41` both have the visibility `Fix`.

On top of these base mechanisms, more high-level concepts are provided [9]. *Validities* are used to express states of versions, e.g. `tested`, `released`, etc. Validities are global, they refer to the whole product rather than to individual components. A validity is defined by a boolean expression. It may be referenced in a version description (see below) to ensure that the selected product version has a certain state, e.g. we may want to work on a `tested` version.

A *version description* consists of *constraints* and *preferences*. A constraint is a mandatory condition on option bindings. A preference consists of option

bindings which are not enforced. Preferences are weighted by rational numbers. A positive number means that binding of an option to a certain value is preferred; analogously, a negative number indicates that the option binding should be avoided. In Fig. 8, the version description requires inclusion of `Fix` (constraint) and prefers inclusion of `X11` and `Unix`:

$$\texttt{Fix} \wedge \texttt{+1[X11]} \wedge \texttt{+1[Unix]}$$

To keep version descriptions short, *aggregates* are introduced. An aggregate is a named version rule which refers to a set of mandatory or preferred option bindings. For example, the following aggregate denotes a sequence of fixes to be applied together:

$$\texttt{Fixes} = \texttt{Fix1} \wedge \texttt{Fix2} \wedge \texttt{Fix3}$$

The *rule base* consists of version rules of the same form as used in version descriptions. Aggregates stored in the rule base may be referenced in version descriptions. Note that aggregates can be used to modularize the rule base; modules can be activated as required by mentioning their names in version descriptions. Furthermore, *defaults* are added implicitly when a version description is evaluated. In Fig. 8, all version rules are defaults which express *mutual exclusion* of options. These constraints are defined by means of the operator $\otimes$:

$$\texttt{Unix} \otimes \texttt{DOS} \otimes \texttt{VMS}$$

$$\texttt{X11} \otimes \texttt{SunViews} \otimes \texttt{Windows}$$

The *evaluator* takes the version description and the rule base and calculates a choice (or an ambition). In general, there is no unique "best" solution satisfying a version description. A heuristic algorithm searches for a solution, guided by preferences. In case of an ambition, the algorithm tries to minimize the number of option bindings; in case of a choice, it tries to maximize the number of options set to true. In our example, the ambition evaluates to `Fix`, i.e. preferences do not narrow the ambition. However, they do affect the choice (options `X11` and `Unix`).

Recent work on high-level extensions of change-oriented versioning is described in a companion paper [16]. That paper introduces additional types of constraints not covered above, e.g. option dependencies, and it also describes tools for managing the option space and supporting consistent version selection.

4.5 Summary

Table 3 summarizes the change-oriented models which we have presented in this section. The many blank entries clearly indicate that more research in version rules is required. Only COV addresses this issue to some extent.

	PIE	Aide-de-Camp	MVPE	COV
vers. db.				
selection order	version first	version first	version first	version first
product space	Smalltalk programs	ER database (file entities)	text files	EER database (file entities)
version space	layers	change sets	dimensions	options
conf. descr.				
formalism	(extensional only)	(extensional only)	query by example	boolean expressions
rule classes	—	—	no distinction	constraints preferences
rule base				
formalism	relationships between layers	—	—	boolean expressions
rule classes	constraints	—	—	constraints preferences defaults validities

Table 3. Comparison of change-oriented approaches

5 Conclusion

Table 4 contrasts the main features of version-oriented and change-oriented approaches to configuring versioned products. Let us summarize their *strengths* and *weaknesses*:

- Change-oriented models have a nice link to change requests and long transactions. The user directly refers to a change spanning multiple components and is not bothered with the tedious task of bookkeeping which component versions make up a logical change. The flexibility is extremely high, since new component versions can be constructed as required by merging changes. However, constraints on change combinations have to be managed carefully; raw merging often yields an inconsistent result.
- In version-oriented models, the product structure is referenced in the version rules (white box approach). Therefore, version-oriented models may express product-related version concepts — e.g. alternative realization variants of an interface, see Adele — which go beyond the black box approach of change-oriented models. On the other hand, version-oriented models lack grouping of logical changes affecting multiple components. Furthermore, flexibility is limited since only already existing component versions can be used for the construction of a configuration (no implicit merges).

We suggest the following topics to be addressed by future work:

	version-oriented models	**change-oriented models**
version space	version graphs (revisions and variants) version attributes	product-level changes attributes controlling change application
configuration	$\sum$ component versions	base version + $\sum$ changes
product structure	white box approach (query references the structure)	black box approach (structure transparent to the query)
version rules	expressions over version attributes	expressions over change attributes
constraints	conditions on version attributes (e.g. consistent variant selection)	conditions on change combinations (e.g. c_1 implies c_2)
versioning	explicit (members of the version graph)	implicit (any change combination)
combinability	v^m (m modules in v versions)	2^v (v changes)

Table 4. Comparison of version-oriented and change-oriented models

- Change-oriented and version-oriented models have complementary strengths. Initial attempts to combine these approaches have been undertaken, but more work has to be done to come up with a *unified model*.
- In this paper, construction of a configuration is viewed as evaluation of a query against a *deductive database*. However, deductive databases are not yet used widely, neither in general nor in SCM. The potentials of applying deductive databases need to be investigated further.
- We have focused primarily on technical issues, in particular concerning the formalisms used for writing version rules. *Experiences* gained from actual use of configurators have to be discussed, as well.
- Many configurators operate on a low *semantic level*, e.g. raw text-oriented merges in change-oriented approaches or composition of versioned files in version-oriented approaches. Raising this semantic level may improve detection of inconsistencies and conflicts.
- More support is required for *managing* the complexity of the *version space*. In both version- and change-oriented models, the version space may become untractable when a large software product evolves over a long period. Constraints excluding inconsistent combinations of versions or changes are essential for managing complexity. Furthermore, appropriate *visualization* techniques may prove very helpful, see e.g. [8].
- Finally, we have tacitly assumed that the rule base is located on top of the versioned database, without being versioned itself. However, the rule base is

subject to change as the underlying product evolves. *Versioning* of the *rule base* raises some difficult modeling issues, concerning e.g. relations between evolution of the rule base and the product, or meta rules for configuring the rule base.

Acknowledgements

We would like to thank the anonymous reviewers as well as Bjørn Gulla and Bjørn Munch for helpful and constructive comments on this paper.

References

1. Y. Bernard, M. Lacroix, P. Lavency, and M. Vanhoedenaghe. Configuration management in an open environment. In G. Goos and J. Hartmanis, editors, *Proceedings of the 1st European Software Engineering Conference*, LNCS 289, pages 35–43. Springer-Verlag, Sept. 1987.
2. R. Conradi. Configuration management. Course material, NTH, June 1995.
3. R. D. Cronk. Tributaries and deltas. *BYTE*, pages 177–186, January 1992.
4. J. Estublier. A configuration manager: The Adele data base of programs. In *Proceedings of the Workshop on Software Engineering Environments for Programming-in-the-Large*, pages 140–147, Harwichport, Massachusetts, June 1985.
5. J. Estublier. Configuration management: The notion and the tools. In Winkler [29], pages 38–61.
6. J. Estublier and R. Casallas. The Adele configuration manager. In Tichy [24], pages 99–134.
7. I. P. Goldstein and D. G. Bobrow. A layered approach to software design. Technical Report CSL-80-5, XEROX PARC, 1980.
8. B. Gulla. *User Support Facilities for Software Configuration Management*. PhD thesis, NTH Trondheim, 1996.
9. B. Gulla, E.-A. Karlsson, and D. Yeh. Change-oriented version descriptions in EPOS. *Software Engineering Journal*, 6(6):378–386, Nov. 1991.
10. G. E. Kaiser and A. N. Habermann. An environment for system version control. In *Digest of Papers of Spring CompCon '83*, pages 415–420. IEEE Computer Society Press, Feb. 1983.
11. P. Lavency and M. Vanhoedenaghe. Knowledge based configuration management. In B. Shriver, editor, *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, pages 83–92, 1988.
12. D. Leblang. The CM challenge: Configuration management that works. In Tichy [24], pages 1–38.
13. D. B. Leblang and G. D. McLean, Jr. Configuration management for large-scale software development efforts. In *Proceedings of the Workshop on Software Engineering Environments for Programming-in-the-Large*, pages 122–127, Harwichport, Massachusetts, June 1985.
14. A. Lie, R. Conradi, T. Didriksen, E. Karlsson, S. O. Hallsteinsen, and P. Holager. Change oriented versioning. In C. Ghezzi and J. A. McDermid, editors, *Proceedings of the 2nd European Software Engineering Conference*, LNCS 387, pages 191–202. Springer-Verlag, Sept. 1989.

15. S. A. MacKay. The state-of-the-art in concurrent, distributed configuration management. In J. Estublier, editor, *Proceedings of the 5th International Workshop on Software Configuration Management*, LNCS 1005, pages 180–194. Springer Verlag, 1995.
16. B. Munch. HiCOV: Managing the version space. In *Proceedings of the 6th International Workshop on Software Configuration Management*, 1996.
17. B. P. Munch, J.-O. Larsen, B. Gulla, R. Conradi, and E.-A. Karlsson. Uniform versioning: The change-oriented model. In *Proceedings of the 4th International Workshop on Software Configuration Management (Preprint)*, pages 188–196, Baltimore, MD, May 1993.
18. K. Narayanaswamy and W. Scacchi. Maintaining configurations of evolving software systems. *IEEE Transactions on Software Engineering*, SE-13(3):324–334, Mar. 1987.
19. D. E. Perry. Version control in the Inscape environment. In *Proceedings of the 9th International Conference on Software Engineering*, pages 142–149, Monterey, CA, Mar. 1987.
20. K. Ramamohanarao and J. Harland. An introduction to deductive database languages and systems. *The VLDB Journal*, 3(2):107–122, April 1994.
21. M. J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, SE-1(4):364–370, Dec. 1975.
22. N. Sarnak, R. Bernstein, and V. Kruskal. Creation and maintenance of multiple versions. In Winkler [29], pages 264–275.
23. Software Maintenance and Development Systems. *Aide-de-Camp Product Overview*, 1990.
24. W. Tichy, editor. *Configuration Management*. John Wiley and Sons, New York, 1994.
25. W. F. Tichy. A data model for programming support environments. In *Proceedings of the IFIP WG 8.1 Working Conference on Automated Tools for Information System Design and Development*, pages 31–48, Jan. 1982.
26. W. F. Tichy. RCS – A system for version control. *Software–Practice and Experience*, 15(7):637–654, July 1985.
27. W. F. Tichy. Tools for software configuration management. In Winkler [29], pages 1–20.
28. J. F. H. Winkler. Version control in families of large programs. In *Proceedings of the 9th International Conference on Software Engineering*, pages 150–161, Monterey, CA, Mar. 1987.
29. J. F. H. Winkler, editor. *Proceedings of the International Workshop on Software Version and Configuration Control*, Stuttgart, Germany, 1988. German Chapter of the ACM, B.G. Teubner.
30. A. Zeller. A unified version model for configuration management. In *Proceedings of the ACM SIGSOFT '95 Symposium on the Foundations of Software Engineering*, pages 151–160, 1995.
31. A. Zeller and G. Snelting. Handling version sets through feature logic. In *Proceedings 5th European Software Engineering Conference*, LNCS 989, pages 191–204. Springer Verlag, 1995.

This article was processed using the LaTeX macro package with LLNCS style