# Software Configuration with Feature Logic

**Andreas Zeller**
Technische Universität Braunschweig
Abteilung Softwaretechnologie
D-38092 Braunschweig
zeller@acm.org

## Abstract

*Software configuration management (SCM) is the discipline for controlling the evolution of software systems. The central problems of SCM are closely related to central artificial intelligence (AI) topics, such as* knowledge representation *(how do we represent the features of versions and components, and how does this knowledge involve in time?),* configuration *(how do we compose a consistent configuration from components, and how do we express constraints?), and* planning *(how do we construct a software product from a source configuration, and what are the features of this product?).*

*Although the research communities of both SCM and AI work on configuration topics, the knowledge about the mutual problems and methods is still small. We show how* feature logic, *a description logic with boolean operations, can be used to represent both knowledge about versions and components, as well as to infer the consistency of possible configurations and thus solve configuration problems in SCM. This interplay of knowledge representation and configuration techniques shows immediate beneficial consequences in SCM, such as the integration and unification of SCM versioning concepts. Moreover, SCM may turn out as a playground for testing and validating new AI methods in practice.*

## 1 Introduction

Software Configuration Management (SCM) is the discipline for controlling the evolution of software systems. While early SCM tools were confined to basic tasks such as revision control (e.g. RCS, SCCS), variant control (e.g. CPP) or system construction (e.g. MAKE), today's SCM systems provide integrated support for tasks such as identification and retrieval of components and configurations, revision and variant control, or consistency checking.

One of the benefits of SCM is that it can be easily automated, since its items are already under computer control; all properties of configuration items can be immediately observed and deduced. It is thus surprising that the artificial intelligence (AI) configuration community has not yet discovered SCM as a fruitful application domain for configuration problems, just as it is surprising that the SCM literature is essentially devoid of formal approaches, let alone formal approaches to configuration problems. A possible reason for this gap is that SCM touches a wide range of divergent subjects, from component identification and configuration problems over software process modeling to general management issues, each with its distinct research community.

Our research group in Braunschweig aims to exploit recent fundamental achievements for the benefit of practice, notably the application of AI techniques in software engineering. We found the core techniques of SCM closely related to well-established AI research topics:

**Knowledge representation.** How do we express knowledge about a software component, such that we can identify and retrieve components and configurations? How does this knowledge evolve in time, and how does it propagate to configurations?

**Configuration.** How can we determine the consistency of a configuration of software components? How do we express configuration constraints, and how are these related to component identification?

**Planning.** How is software constructed and delivered? How do the properties of source components determine the properties of derived components?

Starting with these relationships, we decided to examine the current state of practical SCM, to identify SCM problems, and to devise possible solutions from AI research.

## 2 The Versioning Problem

We begin with a short introduction to SCM. In the SCM domain, we have the problem of maintaining components in several *versions*. Versions are created in several *versioning dimensions*, depending on the intentions of their creator. SCM research distinguishes three versioning dimensions.

**Historical versioning.** Versions that are created to *supersede* a specific version, e.g. for maintenance purposes, are called *revisions*. When a

new revision is created, evolution of the original version is phased out in favor of the new revision. In practice, a revision of a component is usually created by modifying a copy of the most recent revision. The old revisions are permanently stored for maintenance and documenting purposes.

**Logical versioning.** In contrast to revisions, a variant is created as an *alternative* to a specific version. *Permanent variants* are created when the product is adapted to different environments. Variance can again arise in several dimensions, including varying user requirements and varying system platforms, but also variants for testing and debugging.

**Cooperative versioning.** A *temporary variant* is a variant that will later be integrated (or merged) with another variant. Temporary variants are required, for example, to change an old revision while the new revision is already under development, or to realize cooperative work through parallel development threads.

Of these three versioning dimensions, only logical versioning is visible in the final product as different permanent variants. Since maintaining several product variants is more expensive than maintaining one single product, it is a general software engineering issue to keep the number of variants as small as possible. This is achieved through well-known software engineering principles like abstraction, parameterization, generalization, and localization.

One must be aware that only logical versioning can be planned in advance. The creation of revisions and temporary variants may be necessary at any time during the development process, resulting in a huge set of possible configurations, which must be identified and tested.

The problem becomes worse when individual *changes* are considered rather than versions, since each combination of changes results in a different configuration. While software engineering principles help to confine the impact of changes behind a certain abstraction, change combinations nonetheless must be identified and evaluated.

Furthermore, there is a transition from static configuration at compile-time to dynamic configuration at run-time, which results in new configuration problems in the final software product. Already in a small software system with but a few thousand components, SCM can rapidly turn into a nightmare unless intelligent tools help to manage this mess.

## 3 SCM Versioning Issues

The maintenance of several versions can be dramatically simplified by using an automated SCM system. Even the easiest SCM system provides some basic support for the following SCM tasks.

**Identification.** Each component in a software product must be uniquely identifiable and and accessible in some form. Identification schemes, as found in SCM systems, range from simple *revision numbering* (revision 1.0, 1.1, 1.2 ... ) up to *faceted classification* using attribute/value-pairs ($state = experimental \wedge version = 1.0$). The mechanisms to *select* component versions include creation dates, revision numbers, boolean attribute/value combinations, as well as high-level database query languages.

**Composition.** As specific versions of components are composed to configurations, these configurations must be identifiable and accessible as well. Simple SCM tools allow to tag individual component versions with a *label* identifying the configuration. Selecting a configuration is done through a label selecting the appropriate component versions. For instance, the label *REV_1_0* may denote a configuration including revision 1.4 of component $A$ and revision 1.7 of component $B$. More advanced SCM systems allow versioning of configurations just like versioning of components.

**Consistency.** Advanced SCM systems allow users to specify *configuration rules* constraining possible configurations. Such configuration rules may denote that certain changes imply or exclude each other, that changes may be applied to certain variants only, or that only specific variants and revisions result in a well-tested configuration.

Modern SCM systems provide adequate mechanisms to identify and compose software component versions in a software product. The most important SCM issue in this area is *generalizing*—that is, to find a common subset of versioning techniques to improve the interoperability of SCM systems.

Consistency control, however, is still a challenge—especially because most consistency problems arise through the integration of versioning dimensions, which is still at a very early stage.

As a simple example of the consistency problems as found in SCM, consider figure 1, illustrating the dependencies between changes and revisions. Each revision $R_i$ is the product of some changes $\delta_j$ applied to a baseline revision $R_0$. In the diagram, each set $\Delta_j$ contains the revisions the change $\delta_j$ has been applied to. Hence, revision $R_5$ is the product of the changes $\delta_1$ to $\delta_5$ applied to $R_0$, but the change $\delta_6$ has not been applied.

We see that the changes are not orthogonal to each other. For instance, the change $\delta_2$ implies that $\delta_1$ be applied first, since $\Delta_2$ is a subset of $\Delta_1$. Likewise, the changes $\delta_2$ and $\delta_6$ exclude each other, since the sets $\Delta_2$ and $\Delta_6$ are disjoint. The SCM system must ensure that these constraints are satisfied. The problem becomes worse if not only six, but several thou-

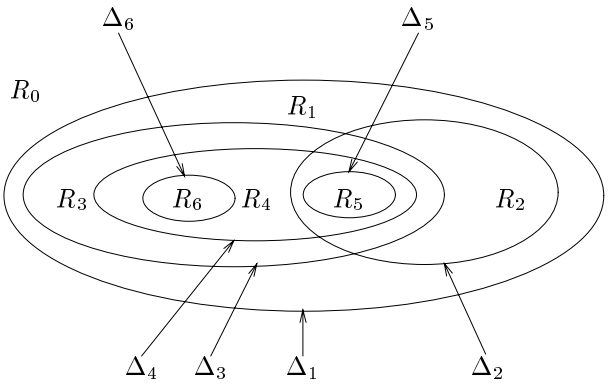


Figure 1: Changes and revisions

sand changes are involved, some of them restricted to a particular variant or configuration, which in turn may impose other constraints.

Finally, constraints are subject to changes as well. For example, a change may be initially visible in a user's temporary variant only. This means that there is a constraint associated with this change that implies a specific user. Later, the same change may be incorporated into the delivered product. This means that the original constraint must be replaced by a new constraint implying a specific configuration.

Just like constraints, the system hardware, process models, or even SCM policies may evolve in ways that cannot be foreseen, and this evolution may be subject to constraints again. This is the true challenge of SCM: Changes and constraints pervade every single item considered, from components to configurations to the processes themselves.

## 4 A SCM Foundation

In our quest for a SCM foundation, we searched for a formalism that allows us to capture and unify the core techniques of SCM, supporting evolution and versioning on every SCM level. As stated in the introduction, we have examined logical formalisms supporting knowledge representation as well as configuration and planning in the SCM domain. We identified two possible foundations:

**Description logics.** In the SCM area, it is common to identify component versions by faceted classification, using a set of *attribute*/*value* pairs. For us, description logics turned out to be a first-order choice for identifying components and expressing knowledge about their possible use in a configuration.

**First-order logic.** Another technique frequently found in SCM systems is to use first-order logic expressions to select and identify configurations, as well as to express consistency constraints; only configurations satisfying the constraints are consistent. Supporting first-order logic was essential for us in order to capture these selection schemes.

Fortunately, we found a formalism which captures both description and first-order logic: *Feature logic*, as defined by Smolka [1], is a well-founded description logic that, in contrast to most description logics, includes quantification, disjunction, and negation over attribution terms, forming a full boolean algebra. Feature logic gives us one single formalism for both knowledge representation and configuration problems.

## 5 Configuration with Feature Logic

Using feature logic, all components are identified by a *feature term*, describing the component through *features*—that is, a pair *feature-name*: *feature-value*. Typically, components and component versions are identified by a conjunction ("$\sqcap$" or "[...]") of their features. For instance, we can distinguish two versions of a *printer* component with respect to their respective printer language:

$$\begin{aligned} \mathit{printer}_1 &= [\mathit{print\text{-}language}\colon \mathit{postscript}] \\ \mathit{printer}_2 &= [\mathit{print\text{-}language}\colon \mathit{text}] \ . \end{aligned}$$

Feature logic now allows us to express feature negations ("$\sim$"), expressing undefined features, or disjunctions ("$\sqcup$" or "$\{\ldots\}$"), expressing alternatives. In figure 2, we have summarized the syntax of feature terms.

The ability to express alternatives is essential in SCM, since one frequently must handle all versions of a component as a single item or component family. Using alternatives, we can identify the component family *printer* containing both $\mathit{printer}_1$ and $\mathit{printer}_2$ as

$$\begin{aligned} \mathit{printer} &= \mathit{printer}_1 \sqcup \mathit{printer}_2 \\ &= [\mathit{print\text{-}language}\colon \{\mathit{postscript}, \mathit{text}\}] \end{aligned}$$

and thus immediately determine the features of the *printer* component family.

Negations can be used to identify component *revisions* by distinguishing whether a change has been applied ("$\delta_i\colon \top$"), or not ("$\sim\delta_i\colon \top$"). For instance, consider a *screen* component, where the $\delta_1$ change introduces a new revision which can handle display PostScript:

$$\begin{aligned} \mathit{screen}_1 &= [\sim\delta_1\colon \top, \mathit{screen\text{-}language}\colon \mathit{bitmap}] \\ \mathit{screen}_2 &= [\delta_1\colon \top, \mathit{screen\text{-}language}\colon \mathit{bitmap}] \\ \mathit{screen}_3 &= [\delta_1\colon \top, \mathit{screen\text{-}language}\colon \mathit{postscript}] \end{aligned}$$

This change determines the *screen* component family as

$$\begin{aligned} \mathit{screen} &= \mathit{screen}_1 \sqcup \mathit{screen}_2 \sqcup \mathit{screen}_3 \\ &= \big(\mathit{screen\text{-}language}\colon \{\mathit{postscript}, \mathit{bitmap}\}\big) \\ &\quad \sqcap \big(\mathit{screen\text{-}language}\colon \mathit{postscript}] \rightarrow [\delta_1\colon \top]\big) \end{aligned}$$

stating in an implication ("$\rightarrow$" with $A \rightarrow B \equiv \sim A \sqcup B$) that selecting the [*screen-language*: *postscript*] version implies that the $\delta_1$ change has been applied.

| Notation | Name | Interpretation |
|---|---|---|
| $a$ | Literal | |
| $V$ | Variable | |
| $\top$ (also $[]$) | Top | Ignorance |
| $\bot$ (also $\{\}$) | Bottom | Inconsistency |
| $f\colon S$ | Selection | The value of $f$ is $S$ |
| $f\colon \top$ | Existence | $f$ is defined |
| $f\!\uparrow$ | Divergence | $f$ is undefined |
| $f \downarrow g$ | Agreement | $f$ and $g$ have the same value |
| $f \uparrow g$ | Disagreement | $f$ and $g$ have different values |
| $S \sqcap T$ (also $[S,T]$) | Intersection | Both $S$ and $T$ hold |
| $S \sqcup T$ (also $\{S,T\}$) | Union | $S$ or $T$ holds |
| $\sim S$ | Complement | $S$ does not hold |
| $S \rightarrow T$ | Implication | If $S$ holds, then $T$ holds |
| $S \sqsupseteq T$ | Subsumption | $S$ subsumes $T$; $T$ implies $S$ |

Figure 2: The syntax of feature terms

When composing configurations, they inherit the features from their components; the feature values are unified. This allows us to use feature terms as configuration constraints. As an example, take a component *dumper* which copies data from the screen to a printer. Both formats must be identical, as expressed through the common variable $D$:

$$\mathit{dumper} = [\mathit{screen\text{-}language}\colon D, \mathit{print\text{-}language}\colon D]$$

Now consider the configuration *subsystem* composed from the three components *dumper*, *screen*, and *printer*. Its features are determined as:

$$\begin{aligned} \mathit{subsystem} &= \mathit{printer} \sqcap \mathit{screen} \sqcap \mathit{dumper} \\ &= [\delta_1\colon \top, \mathit{print\text{-}language}\colon \mathit{postscript}, \\ &\qquad \mathit{screen\text{-}language}\colon \mathit{postscript}] \ , \end{aligned}$$

where all other configurations have been excluded by the features of *dumper* and *screen*. We see how features can represent knowledge about the component as well as constraints about its usage in a specific configuration.

## 6 Constraints as Features

Our primary aim for using feature logic is to use one single identification formalism in all versioning dimensions. The resulting *version set model* uses feature terms to identify arbitrary versions: revisions are identified by changes applied or not applied ($\delta_{47}\colon \top$), temporary variants are identified by specific users ($\mathit{user}\colon \mathit{lisa}$) or teams ($\mathit{team}\colon \mathit{microserfs}$), and permanent variants are identified by general feature terms ($\mathit{os}\colon \{\mathit{windows95}, \mathit{windows\text{-}nt}\}$). Feature terms are used for identifying and selecting arbitrary subsets in all versioning dimensions. Many examples illustrating the usage of this model are given in [2, 3, 4], and especially in [5].

Besides this SCM-specific integration of versioning concepts, feature logic has another substantial advantage: the representation of configuration constraints as component features (or version features). Among the most important benefits are:

**Distributed constraints.** Rather than having one central instance defining the possible configurations, each component and each version defines the constraints related to its usage. If the component is excluded from a configuration, its usage constraints are excluded as well. Since the constraints are evaluated incrementally while the configuration is composed, developers can detect inconsistencies already at the subsystem or even at the component level, which avoids propagating inconsistencies across subsystem or workspace boundaries.

**Constraint evolution.** Since configuration constraints are associated with components, they are versioned like the components themselves. Upon creating a new component version, developers can choose whether to inherit the features and constraints of the base version, or to assign new features and constraints. Consequently, versions of the configuration space and component versions determine each other.

**One single representation.** Finally, through the exclusive use of feature logic, there is no hierarchy between objects to be configured and the configuration rules themselves. Configuration rules may imply other features, and vice versa—constraints may be subject to versioning just as specific versions may imply specific constraints. For instance, we may select configurations by stating their constraints ("Show all configurations where the $\delta_{43}$ change implies the UNIX operating system").

The drawback of this flexibility is *computation complexity*. Feature unification, the primary method to determine consistency of feature terms, is $\mathcal{NP}$-complete, which results in exponential worst-case

complexity for all SCM operations. However, "classical" SCM operations—that is, the ones that are used in today's SCM systems—impose no special complexity problems when modeled using feature logic. We found that the majority of SCM problems either imposes very *tight* or very *loose* configuration constraints. Tight constraints, as in change dependencies, are easy to handle since they reduce the configuration space. Loose constraints, as in orthogonal variance dimensions, are also easy to handle since their satisfaction is easily computed.

However, these simplifications apply to today's SCM systems only. Future SCM systems supporting arbitrary versioning dimensions and arbitrary configuration constraints will hurt this complexity barrier. This is a challenge for both SCM and AI researchers. In the SCM domain, we must find out how far new SCM tasks and procedures can go without being endangered by the underlying complexity. And in the AI domain, we must devise and exchange methods to handle huge sets of intertwined configuration constraints and alternatives.

## 7 Conclusion

Using feature logic for both knowledge representation and configuration constraints turned out to be a valuable contribution to the SCM area. Among the preliminary results are:

- The efficient integration of the four main SCM versioning models, resulting in a general SCM foundation [2];
- A unified versioning model for SCM, increasing flexibility in the software process [5];
- The development of FFS, a virtual file system which allows transparent access to arbitrary file and directory versions just by stating attribute constraints [3];
- The implementation of ICE, an inference-based SCM system supporting deductive software construction as well as interactive exploration of the configuration space on top of the FFS [4].

As a conclusion, the application of a theoretical AI foundation to a practical software engineering problem resulted in a success story. The interplay of knowledge representation and configuration techniques raised a number of potential complexity problems, but also showed immediate beneficial consequences in SCM. In general, the SCM domain may turn out as a valuable playground for testing and validating new AI methods in practice—hopefully somewhat closing the gap between configuration practice and configuration theory.

ICE is part of the inference-based software development environment NORA[1]. NORA aims at utilizing inference technology in software tools. ICE software for UNIX systems and related technical reports can be accessed through the ICE WWW page, http://www.cs.tu-bs.de/softech/ice/ as well as directly via anonymous FTP from ftp://ftp.ips.cs.tu-bs.de/pub/local/softech/ice/.

[1]NORA is a figure in Henrik Ibsen's play "A Dollhouse". Hence, NORA is NO Real Acronym.

**Acknowledgments.** This work was funded by the Deutsche Forschungsgemeinschaft, grants Sn11/1-2 and Sn11/2-2.

## References

[1] SMOLKA, G. Feature-constrained logics for unification grammars. *Journal of Logic Programming 12* (1992), 51–87.

[2] ZELLER, A. A unified version model for configuration management. In *Proc. 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering* (Washington, DC, Oct. 1995), G. Kaiser, Ed., vol. 20 (4) of *ACM Software Engineering Notes*, ACM Press, pp. 151–160.

[3] ZELLER, A. Smooth operations with square operators—The version set model in ICE. In *Proc. 6th International Workshop on Software Configuration Management* (Berlin, Germany, Mar. 1996), I. Sommerville, Ed., Lecture Notes in Computer Science, Springer-Verlag. To appear.

[4] ZELLER, A., AND SNELTING, G. Handling version sets through feature logic. In *Proc. 5th European Software Engineering Conference* (Sitges, Spain, Sept. 1995), W. Schäfer and P. Botella, Eds., vol. 989 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 191–204.

[5] ZELLER, A., AND SNELTING, G. Unified versioning through feature logic. Computer Science Report 96-01, Technical University of Braunschweig, Germany, Mar. 1996. Invited for submission to ACM Transactions on Software Engineering and Methodology.