Incremental Configuration Management Based on Feature Unification

Andreas Zeller and Gregor Snelting

Informatik-Bericht No. 94-04
18. April 1994

Incremental Configuration Management Based on Feature Unification

Andreas Zeller and Gregor Snelting
Institut für Programmiersprachen und Informationssysteme
Abteilung Softwaretechnologie
Technische Universität Braunschweig
Gaußstraße 17
D-38092 Braunschweig/Germany
{zeller,snelting}@ips.cs.tu-bs.de

Abstract

We apply *feature logic* to the problem of incremental configuration management. Feature logic has originally been developed in computer linguistics as a knowledge representation and inference mechanism. It offers a uniform formalism for the description of variants and revisions, where *sets of versions* rather than single versions are the basic units of reasoning. Feature logic thus opens a whole algebra of version sets, which includes specific configurations as special cases. Our approach allows for interactive configuration management, where a configuration thread is constructed by adding or modifying configuration constraints until either a complete configuration or an inconsistency can be deduced. A set of versions of a software component can be represented and processed as a single source file enriched with preprocessor statements. Thus, our tool can be used as an intelligent front end to more traditional techniques.

Key words: software configuration management, version control, deduction and theorem proving, knowledge representation formalisms and methods.

1 Introduction

Software Configuration Management (CM) is a discipline for controlling the evolution of software systems. CM provides support for problems such as *identification of components and systems* [LM88, LCD$^+$89, Nar89, Nic91], *revision and variant control* [Tic85, Rei89], or *consistency checking* [Est88, PF89]. Available tools are however not completely satisfying, since often they are neither interactive nor incremental or force the programmer to explicitly specify redundant details, until a configuration thread can be determined or inconsistencies can be detected.

In order to overcome these shortcomings, we present a unified approach to configuration management based on *feature logic* [Smo92]. Feature terms are boolean expressions over (*name*: *value*)-attributions, called features, where values may be atomic constants, variables or nested feature terms. We use feature logic to identify components by their features. A feature term represents an (infinite) set of variable-free ground terms. Therefore, feature logic allows us to operate on arbitrary *version sets*, specified by the common features of the versions and providing intersection, union and complement operations. *Feature unification* is used to implement intersection of infinite version sets, allowing for deduction of configuration threads from incomplete specifications and early detection of inconsistencies. Feature terms can be transformed into graphical panels, allowing for interactive configuration management.

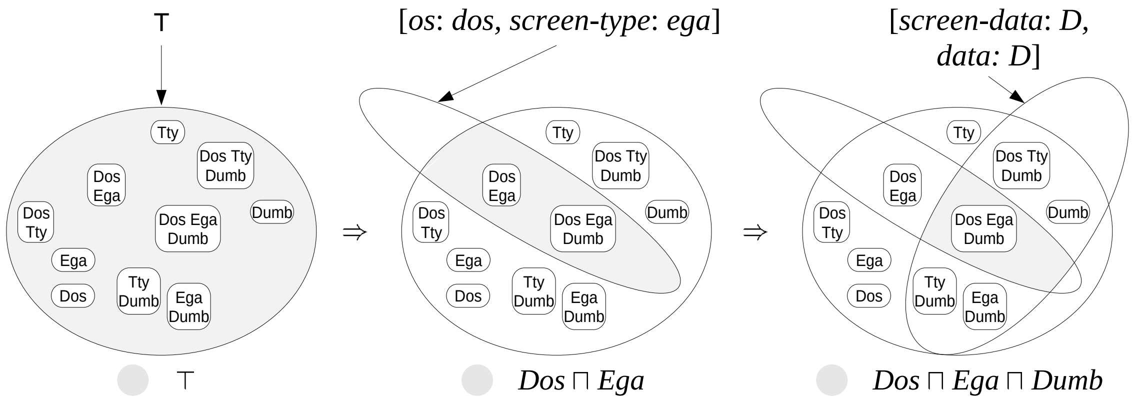


Figure 1: Incremental selection of a configuration

As a simple example, consider figure 1, where the big bubbles denote software components and the boxes inside denote variants. Starting with an unspecified configuration, we can specify more and more features of the configuration we want. For instance, we can select all versions with the *dos* operating system and *ega* screen; this is done by clicking buttons on a feature panel. Feature logic is used to deduce which versions may still be included in the configuration. Once a specific version has been deduced for all components in question, the configuration thread is complete. For example, if we add the constraint that screen type and device driver type must be equal (this is expressed by use of the variable D), the set of possible configurations collapses to a specific configuration, namely the "dos-ega-dumb" configuration.

Feature logic guarantees consistency of configurations with respect to their features. If the user selects contradictory features, this will be detected as soon as a consistent configuration is no longer possible (that is, the set of possible versions has become empty). Finally, feature logic allows to model changes in time as well, resulting in a unified scheme to incrementally select and configure arbitrary consistent sets of revisions and variants.

We have implemented a tool, called *Incremental Configuration Engine* (ICE), which realizes the above approach. In order to "plug in" ICE into the standard UNIX environment, version sets as selected by the user or deduced by ICE can be transformed to more traditional representations. A widely used representation of a set of versions of a software component is a single source file enriched with preprocessor statements, where configuration-dependent code pieces are enclosed in **#if** … **#endif** brackets. Our tool can produce and process such representations; **#if**s will only be used for ambiguities which could not be resolved by ICE.

ICE is part of the inference-based software development environment NORA[1]. NORA aims at utilizing unification theory and inference technology in software tools; concepts and preliminary results can be found in [SGS91, GS93, SFG$^+$94, KS94].

[1] NORA is a play by the Norwegian writer Henrik Ibsen. Hence, NORA is NO Real Acronym.

2 Basic Notions of Feature Logic

We begin with a short overview of feature logic. Feature terms and feature unification have originally been developed for semantic analysis of natural language [SUP$^+$83, Kay84]. Later, they were used as a general mechanism for knowledge representation [BL84, NS90], and as a basis for logic programming [KR86, SAK90]. In our presentation, we will concentrate on an intuitive understanding, more formal descriptions including semantic aspects can be found elsewhere [Smo92]. Note that we use full feature logic including — besides the "basic" intersection — unions as well as complements. By use of Skolem functions, full feature logic is equivalent to first order predicate logic, but for many purposes (including configuration management), the *feature*: *value* notation is much more natural and appropriate.

Feature terms are record-like structures, which are used to collect descriptions or attributes of certain objects. In their simplest form they consist of a list of named features, where each feature represents an attribute of an object. Slots may have values; values may be (atomic or integer) constants, variables (which can be used to state that certain yet unknown values must be equal) and (nested) features. As an example, consider the following feature term V, which expresses linguistic properties of a piece of natural language:

$$V = \begin{bmatrix} \textit{tense}\colon \textit{present}, \\ \textit{pred}\colon [\textit{verb}\colon \textit{sing}, \textit{agent}\colon X, \textit{what}\colon Y]\,, \\ \textit{subj}\colon [X, \textit{num}\colon \textit{sg}, \textit{person}\colon \textit{3rd}]\,, \\ \textit{obj}\colon Y \end{bmatrix}$$

This term says that the language fragment is in present tense, third person singular, that the agent of the predicate is equal to the subject etc.: V is a representation of the sentence template "X sings Y".

The syntax of feature terms is given in table 1. In this table, a denotes an atomic value or an integer constant (first-order terms as in Prolog may also be used), X, Y, Z denote variables, f denotes a slot name, S and T denote feature terms, and A denotes a sort name. The basic elements of feature terms are name-value pairs, where $f\colon X$ is read as "feature f has value X".

Feature descriptions can be combined using intersection, union, and complement operations. If $S = f\colon X$, $T = g\colon Y$, then $S \sqcap T = [f\colon X, g\colon Y]$, which is read as "$f$ has value X and g has value Y". Similarly, $S \sqcup T = \{f\colon X, g\colon Y\}$, which is read as "$f$ has value X or g has value Y". Note that $f\colon \{X,\ Y\}$ is equivalent to $\{f\colon X, f\colon Y\}$ and is read "f has value X or Y". Sometimes it is necessary to specify that a feature exists (i.e. is defined, but without giving any value), or that a feature does not exist in a feature term. This is written $f\colon \top$ resp. $\sim\! f\colon \top$ (abbreviated as $f\!\uparrow$).

The possibility to specify complements greatly increases the expressive power of the logic. For example, the term $\sim$[*compiler*: *gcc*] denotes all objects whose feature *compiler* is either undefined or has another value than *gcc*. The term [*compiler*: $\sim$*gcc*] however denotes all objects whose feature *compiler* is defined, but with a value different from *gcc*. All laws for Boolean algebras (e.g. de Morgan's law) hold for feature terms as well.

Feature terms denote infinite sets of variable-free feature terms, namely those terms which can be obtained by substituting variables or adding more features. A feature term can thus be interpreted as a representation of the infinite set of all ground terms which are *subsumed* by the original term (for an exact definition of subsumption, see below).

Notation	Name	Interpretation
a	literal	
X	variable	
$\top$	universe	Ignorance
$\bot$	empty set	Inconsistence
$f\colon S$	selection	The value of f is S
$f\colon \top$	existence	f is defined
$f\uparrow$	divergence	f is undefined
$S \sqcap T$ or $[S, T]$	intersection	Both S and T hold
$S \sqcup T$ or $\{S, T\}$	union	S or T holds
$\sim S$	complement	S does not hold
$A\ S$	sort	A is the sort of S
$A \equiv S$	definition	Sort A is defined as S

Table 1: The operations of feature logic

Feature terms can be grouped into *sorts*. A sort comprises feature terms with a similar structure and hence is itself given in form of a feature term. Sorts impose constraints on feature terms by requiring that certain slots must be present and must have specific values. For us, sorts serve as templates for certain feature terms.

The characteristic property of feature terms is *feature consistency*: the value of any feature must be unique. Hence the term [*os*: *dos*, *os*: *unix*] is inconsistent and equivalent to $\bot$. When computing an intersection of feature terms, feature unification is used to ensure consistency. For terms without unions and complements, feature unification works similar to classical unification of first-order terms; the only difference is that subterms are not identified by position (as in Prolog), but by feature name. Adding unions forces unification to compute a (finite) union of unifiers as well, whereas complements are usually handled by constraint solving (similar to negation as failure).

As an example of feature unification, consider the terms

$$S = [f\colon t(X),\ \{g\colon X,\ h\colon b\}] \qquad T = [\sim h\colon Y,\ f\colon t(a),\ i\colon c]$$

The unification problem $S \sqcap T$ follows the general pattern $(A \sqcup B) \sqcap \sim A = B$. Furthermore, $X = a$ must hold, hence $S \sqcap T = [f\colon t(a),\ g\colon a,\ i\colon c]$.

If $S \sqcap T = T$, we say S *subsumes* T, written $T \sqsubseteq S$. The set of all feature terms is partially ordered by $\sqsubseteq$; moreover, it forms a complete lattice, the so-called *subsumption lattice*. In the subsumption lattice, the infimum is computed by unification, and the supremum is given by the "inverse" operation, namely the most specific generalization of two terms.

3 Features of Versions, Components, and Systems

3.1 Features of Versions

Let us now return to configuration management. Here, we have a universe of *components*, where each component is given in one or more *versions*. According with Winkler [Win87], our version concept encompasses both *revisions* and *variants*; revisions supersede an existing version, while variants do not. For purposes of configuration management, every version is assigned a feature term describing its features and uniquely identifying both version and component. Note that we are not restricted to a pure enumeration of features — we may use *complements* such as $\sim$[*operating-system*: *dos*] to express that a version must not be used under the DOS operating system, *variables* such as [*host-arch*: X, *target-arch*: $\sim X$] to describe a cross-compiler (host and target architecture must not be equal), or *unions* like [*state*: $\{$*experimental*, *revised*$\}$] to specify alternatives.

3.2 Features of Components

We assume that at most one version of a component can be used in a system. Thus, we model the features of a component as *alternatives* or as *union* of the version features. For instance, if we have a component *printer* in two versions

$$\begin{array}{lcll} \textit{printer}_1 & = & [\textit{print-language}: \textit{postscript}, & \textit{print-bitmaps}: \textit{true}] \\ \textit{printer}_2 & = & [\textit{print-language}: \textit{ascii}, & \textit{print-bitmaps}: \textit{false}] \end{array}$$

the feature term describing the component is

$$\textit{printer} = \textit{printer}_1 \sqcup \textit{printer}_2 = \left\{ \begin{array}{ll} [\textit{print-language}: \textit{postscript}, & \textit{print-bitmaps}: \textit{true}], \\ [\textit{print-language}: \textit{ascii}, & \textit{print-bitmaps}: \textit{false}] \end{array} \right\}$$

Formally, if we have a component T in n versions $T_1, T_2, \ldots, T_n$, the features of T are given by

$$T = T_1 \sqcup T_2 \sqcup \cdots \sqcup T_n = \bigsqcup_{i=1}^{n} T_i$$

Note that features of the component itself (for instance, the component name) are the same across all versions, and hence can be factored out through $(A \sqcap B) \sqcup (A \sqcap C) = A \sqcap (B \sqcup C)$.

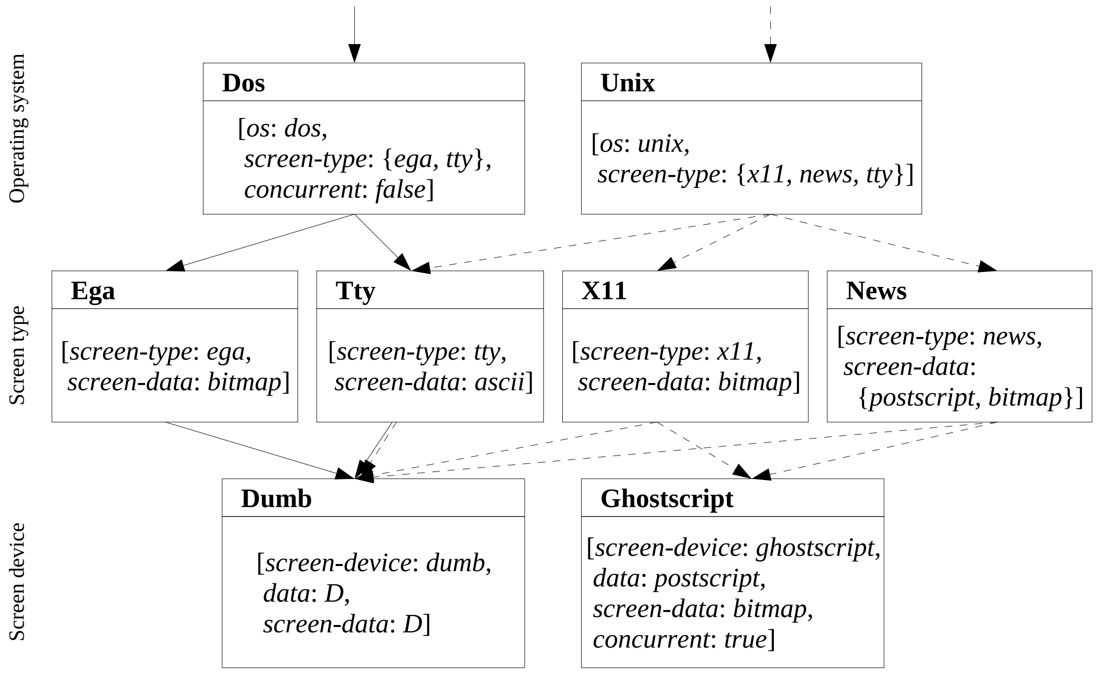
To retrieve a specific version, we specify a *selection key* S giving the features of the desired version. For any selection key S and a set of versions T, we can identify the versions satisfying S by calculating $T' = T \sqcap S$ — that is, the set of versions that are in S as well as in T. If $T' = \perp$, or if T' does not denote any existing version, selection fails.

In our example, selecting $S =$ [*print-language*: *postscript*] from *printer* returns *printer*$_1$, since *printer*$\sqcap S = ($*printer*$_1 \sqcup$*printer*$_2) \sqcap S = ($*printer*$_1 \sqcap S) \sqcup ($*printer*$_2 \sqcap S) =$ *printer*$_1 \sqcup \perp =$ *printer*$_1$. Here, *printer*$_2 \sqcap S = \perp$ holds since the *print-language* feature may have only one value.

Note that T' is just another set of versions, which need not be singleton. We may now give a second selection key S' and select $T'' = T' \sqcap S'$, give a third selection key S'', and so on, narrowing the choice set incrementally until a singleton set is selected.

3.3 Features of Systems

A *system*, in our setting, is a set of component versions. The crucial point when composing systems from versions is to ensure that all versions fit together. For instance, we cannot incorporate two versions with the features [*operating-system*: *windows-nt*] and [*operating-system*: *unix*] in a system, since a system is usually built for only one operating system. Thus, we model the features of a system



Operating system
Dos
[os: dos, screen-type: {ega, tty}, concurrent: false]
Unix
[os: unix, screen-type: {x11, news, tty}]
Screen type
Ega
[screen-type: ega, screen-data: bitmap]
Tty
[screen-type: tty, screen-data: ascii]
X11
[screen-type: x11, screen-data: bitmap]
News
[screen-type: news, screen-data: {postscript, bitmap}]
Screen device
Dumb
[screen-device: dumb, data: D, screen-data: D]
Ghostscript
[screen-device: ghostscript, data: postscript, screen-data: bitmap, concurrent: true]

Figure 2: Consistent configurations in a text/graphic editor

as *intersection* of the version features.[2] Formally, if we have a system of n versions $T_1, T_2, \ldots, T_n$, the features of the system S are given by

$$S = T_1 \sqcap T_2 \sqcap \cdots \sqcap T_n = \bigsqcap_{i=1}^{n} T_i$$

A system is called *consistent* if $S = \bigsqcap_{i=1}^{n} T_i \neq \bot$.

Let us illustrate consistency by an example. In figure 2, we see three components of a text editor, where each component comes in several versions. We can choose between two operating systems (*Dos* and *Unix*), four screen types (*Ega*, *Tty*, *X11* and *News*) and two screen device drivers (*Dumb* and *Ghostscript*). The *Dumb* driver assumes that the screen type can handle the data directly (expressed through variable D); the *Ghostscript* driver is a separate process that can convert postscript data into a bitmap. Note that the features of the versions imply that at most one version of each component be chosen.

We shall now compose a consistent system, beginning by selecting the operating system, and choose the *Dos* version. This implies that we cannot choose the *X11* or *News* screen types, since (in our example), *Dos* does not support them. (Formally, we cannot use *X11* or *News* screen types, since *X11* $\sqcap$ *News* $= [\textit{os}: \textit{dos}, \textit{screen-type}: \{\textit{ega}, \textit{tty}\}] \sqcap [\textit{screen-type}: \{\textit{news}, \textit{x11}\}] = \bot$). We can, however, choose *Ega* or *Tty*, as indicated by plain lines. Now, we must choose the screen device driver. *Ghostscript* cannot be chosen, since it would imply that *concurrent* be true, which is not the case under *Dos*. The *Dumb* driver remains; D is instantiated to *bitmap* or *ascii*, depending on

[2] Features irrelevant for consistency, such as *author* or *last-change-date*, are ignored.

Feature term	CPP expression
$[\mathit{gcc}\colon \mathit{2}]$	$\mathit{gcc} \equiv 2$
$[\mathit{optimize}\colon \mathit{false}]$	$\neg \mathit{optimize}$
$[\mathit{unix}\colon \top]$	$\mathbf{defined}(\mathit{unix})$
$[\mathit{bugs}\uparrow]$	$\neg\mathbf{defined}(\mathit{bugs})$
$[\mathit{lang}\colon \{c, \mathit{pascal}\}]$	$\mathit{lang} \equiv c \vee \mathit{lang} \equiv \mathit{pascal}$
$[\mathit{sin}\colon \sim\!\mathit{proc}]$	$\mathit{sin} \not\equiv \mathit{proc}$
$\sim\![\mathit{sin}\colon \mathit{proc}]$	$\neg(\mathit{sin} \equiv \mathit{proc})$
$[\mathit{host}\colon X, \mathit{target}\colon X]$	$\mathit{host} \equiv X \wedge \mathit{target} \equiv X$

Table 2: Translating feature terms into CPP constraints

the screen type, making our choice complete. As an alternative, consider the choice $[\mathit{os}\colon \mathit{unix}]$, as indicated by dashed lines. Again, each path stands for a consistent configuration.

The composition process may also be interpreted as a selection in the set of possible systems, as shown in figure 1 (omitting the UNIX versions for clarity). By giving more and more features, the choice set is subsequently narrowed until we get the system we want. This results in a configuration scheme where whole systems are determined uniquely by their features.

4 Representing Version Sets

Version sets need not be an abstract concept. In fact, many programmers are already handling version sets, although they would not call it so. *Conditional compilation*, using the C preprocessor (CPP), for instance, represents all versions of a component in a single source file. Code pieces relevant for certain versions only are enclosed in **#if** C … **#endif** pairs, where C expresses the condition under which the code piece is to be included. Upon compilation, CPP selects a single version out of this set, feeding it to the compiler front end.

The obvious advantage of conditional compilation is that the programmer may perform changes simultaneously on a whole set of versions. Unfortunately, code containing **#if**-directives becomes quite unreadable beyond a certain number of versions. This is the drawback of the CPP "one-from-all" approach — we can either handle all versions at once (the source file) or one version (the CPP output). In our approach, this is different. We can represent arbitrary version sets as CPP files, making the choice between "one version" and "all versions" but special cases of a wide spectrum of possible selections, and giving the user a familiar, well-understood representation.

In our CPP representation, the feature terms governing code pieces are mapped to boolean CPP expressions, where feature names are expressed as CPP symbols. In table 2, we give some examples catching the transformation spirit[3].

All version set operators can be applied to CPP files. Union is implemented by computing the *textual difference* for arbitrary disjunct version sets. Selecting a subset S (that is, intersection) is done by unifying the CPP expressions T with S; if $T \sqcap S = \bot$, the appropriate code piece is excluded; if $T \sqcap S = S$, the **#if** directive is removed.

[3] For better readability, the C tokens `==`, `!=`, `&&`, `||`, and `!` are represented as $\equiv$, $\not\equiv$, $\wedge$, $\vee$, and $\neg$, respectively.

```
get_load.c [os: unix, hcx↑]
void InitLoadPoint()
{
  extern void nlist();
#if defined(AIXV3)
  knlist(namelist, 1, ...)
#else
  nlist(KERNEL_FILE, namelist);
#endif
  if (namelist[...].n_type ≡ 0 ∨
      namelist[...].n_value ≡ 0) {
     xload_error(...);
     exit(−1);
  }
```

⊔

```
get_load.c [os: unix, hcx: ⊤]
void InitLoadPoint()
{
  extern void nlist();
  nlist(KERNEL_FILE, namelist);
  if (namelist[...].n_type ≡ 0 ∧
      namelist[...].n_value ≡ 0) {
     xload_error(...);
     exit(−1);
  }
```

=

```
get_load.c [os: unix]
void InitLoadPoint()
{
  extern void nlist();
#if defined(AIXV3) ∧ ¬defined(hcx)
  knlist(namelist, 1, ...)
#else
  nlist(KERNEL_FILE, namelist);
#endif
#if defined(hcx)
  if (namelist[...].n_type ≡ 0 ∧
#else
  if (namelist[...].n_type ≡ 0 ∨
#endif
       namelist[...].n_value ≡ 0) {
     xload_error(...);
     exit(−1);
  }
```

Figure 3: Uniting version sets represented as CPP files

An example is shown in figure 3.[4] Here, two version sets $[os\colon unix, hcx\uparrow]$ and $[os\colon unix, hcx\colon \top]$ are merged to form a new version set $[os\colon unix]$. The two original sets can be selected from $[os\colon unix]$ by giving selection keys $S = [hcx\uparrow]$ or $S = [hcx\colon \top]$, respectively — or, in CPP representation, $S = \neg\mathbf{defined}(hcx)$ and $S = \mathbf{defined}(hcx)$. Again, refinement is possible until we have a singleton set — that is, a version without **#if**-directives.

Since conversion works both ways, extending the original CPP semantics, our system can directly handle existing CPP files, providing simple re-use of existing variant repositories and allowing smooth embedding into common industry standards.[5]

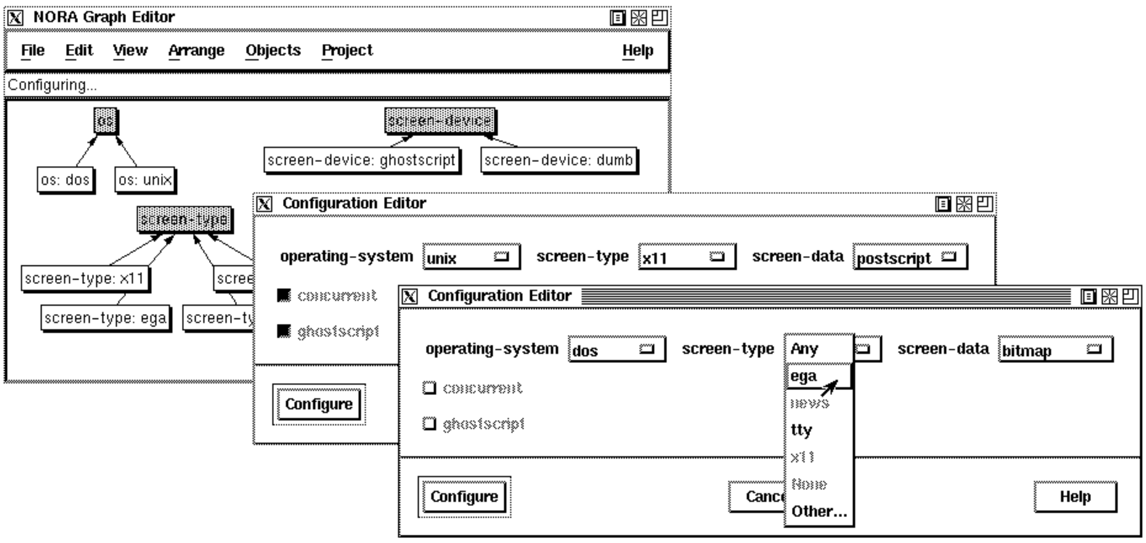
5 Interactive Configuration Selection

With CPP, we have deliberately chosen a representation for hard-core programmers, in order to introduce theoretical foundations into solving of practical problems. In this section, we shall address the promoters of *interactive environments*. It turns out that every feature term has a canonical graphical representation in form of a panel. Therefore we generate *interactive panels* for selecting and modifying configurations.

For each feature occurring in the term describing a component, we generate a menu or a button for selecting the possible values. Additionally, a menu for the feature f contains two entries **Any** and **None**, standing for $[f\colon \top]$ and $[f\uparrow]$, and an **Other** entry for entering arbitrary values.

[4] The code shown is taken from the *xload* program, a tool to display the system load for several architectures.

[5] Even better than re-use of existing CPP files is *restructuring* using the consistency notion of feature logic. Consider the CPP attributes *dos* and *unix*, for example. Obviously, these attributes are meant to be mutually exclusive. But this is better stated explicitly using the feature terms $[os\colon dos]$ and $[os\colon unix]$. Where such knowledge about the semantics of CPP attributes is missing — for instance, are *hcx* and *AIXV3* really mutually exclusive? — *concept analysis*, as shown by Krone and Snelting [KS94], may help to infer the attribute relations.



operating-system unix screen-type x11 screen-data postscript

operating-system dos screen-type Any screen-data bitmap

Figure 4: Configuration selection using panels

In figure 4, some panels for the example from figure 2 are shown. The upper window shows the NORA graph editor, displaying the components and their variants. All three components are selected for configuration. The middle and lower windows are configuration panels in different states.

To ensure consistency, menu items are made *insensitive* (that is, they cannot be selected), if choosing them would lead to an inconsistency with features already chosen. Checking sensitivity is straight-forward: if the system features are T, and T' has already been selected, each item $[f\colon x]$ is sensitive iff $T \sqcap T' \sqcap [f\colon x] \neq \bot$. In our example, the items **news**, **x11**, and **None** are insensitive (shown as "greyed out"), because we have already selected the *dos* operating system. To choose one of these screen types, we first must revoke our choice for the operating system (by resetting the menu to **Any**). Then, we can select an arbitrary screen type — which, in turn, may determine the operating system. All this leads to an *interactive exploration scheme* of the configuration universe, where the global effects of choice refining and revoking are immediately visualized.

In practice, it often suffices to give but a few features of the desired system to see all other features be deducable. In such a case, the user need do no more than acknowledge the remaining features — all other possibilities having been excluded by ICE.

6 A Unified Approach to Variants and Revisions

In this last section, we shall show how changes in time can be handled using feature logic. Traditionally, the concepts of *variants* and *revisions* have been strictly separated, often resulting in two-layered systems — for instance, a combination of RCS and CPP, where first the revision is chosen, then the specific variant; or multi-workspace development, where each variant has its own revision history.

Using feature logic, we propose a *unified* approach allowing arbitrary selection of variants and revisions. Assuming that revisions are created by applying a *change* (or *delta*) to an already existing revision, we may distinguish the old and the new revision by checking whether the change has been

$$\begin{tikzcd}
\Delta_0 \arrow[r, "\delta_1"] & \Delta_1 \arrow[r, "\delta_2"] \arrow[dr, "\delta_3"'] & \Delta_2 \arrow[rr, "\delta_5"] & & \Delta_5 \\
 & & \Delta_3 \arrow[r, "\delta_4"] & \Delta_4 \arrow[ur] &
\end{tikzcd}$$

Figure 5: An evolution diagram

$$\begin{tikzcd}
\top \arrow[r, "\sqsupseteq"] & {[\delta_1 \colon \top]} \arrow[r, "\sqsupseteq"] \arrow[dr, "\sqsupseteq"'] & {[\delta_2 \colon \top]} \arrow[rr, "\sqsupseteq"] & & {[\delta_5 \colon \top]} \\
 & & {[\delta_3 \colon \top]} \arrow[r, "\sqsupseteq"] & {[\delta_4 \colon \top]} \arrow[ur] &
\end{tikzcd}$$

Figure 6: Evolution as subsumption lattice

applied or not — which we represent by absence or presence of a *delta feature* standing for the change application.

In the following, we shall denote the individual revisions of a component by $\Delta_0, \Delta_1, \Delta_2, \ldots$[6] The change leading up to a revision Δ_i is denoted by δ_i. A linear evolution of revisions may thus be depicted as an *evolution diagram*:

$$\Delta_0 \xrightarrow{\delta_1} \Delta_1 \xrightarrow{\delta_2} \cdots \xrightarrow{\delta_i} \Delta_i \xrightarrow{\delta_{i+1}} \Delta_{i+1} \cdots$$

To every Δ_i revision except Δ_0, we assign a delta feature $[\delta_i \colon \top]$. Each Δ_i revision also inherits the delta features of all revisions it is based upon, such that a Δ_2 revision based on the Δ_1 revision has the features $[\delta_1 \colon \top, \delta_2 \colon \top]$ — that is, Δ_2 is the result of the δ_1 and δ_2 changes applied to Δ_0.

As a more complex example, consider figure 5. Here, we have a *split* evolution path: besides Δ_2, the Δ_3 revision is also based on Δ_1. Later, the two evolution paths are *joined* again, resulting in a new revision Δ_5. The Δ_5 revision has the delta features of all its ancestors, namely $[\delta_1 \colon \top, \delta_2 \colon \top, \ldots, \delta_5 \colon \top]$, while the Δ_4 revision does not include the $[\delta_2 \colon \top]$ and $[\delta_5 \colon \top]$ features.[7]

Unfortunately, in order to retrieve an individual revision, one has to exclude later changes explicitly. For example, to retrieve the Δ_4 revision, we must specify $S = [\delta_4 \colon \top, \delta_5 \uparrow]$ so that our set does not include the Δ_5 revision. What is missing here, is a term saying "I want *exactly* these features, and no others." Therefore, ICE generates *sort definitions* for the common task of retrieving previous revisions. In our example, sorts like $\Delta_0 \equiv [\delta_1 \uparrow]$ and $\Delta_1 \equiv [\delta_1 \colon \top, \delta_2 \uparrow, \delta_3 \uparrow]$ may be used for directly retrieving the Δ_0 revision or the Δ_1 revision; the intuitive $\Delta_0 \sqcup \Delta_1$ correctly returns

[6] Since the indices represent points in time, any ordered identification is possible — for instance the classical *major-version.minor-version* scheme as used in RCS. In practice, however, revisions are named by assigning appropriate features.

[7] This assignment of delta features makes evolution diagrams isomorphic to the *subsumption lattice* given by the delta features. For example, in figure 6, an arrow $S \longrightarrow T$ implies that $S \sqsupseteq T$. The set $\top$ includes all revisions; its subset $[\delta_1 \colon \top]$ includes all revisions where the $[\delta_1 \colon \top]$ change has been applied — that is, all revisions except Δ_0. Both $[\delta_2 \colon \top]$ and $[\delta_3 \colon \top]$ again are subsets of $[\delta_1 \colon \top]$ — that is, $[\delta_1 \colon \top] \sqsupseteq [\delta_2 \colon \top] \sqcup [\delta_3 \colon \top]$. Finally, the $[\delta_5 \colon \top]$ revision set containing Δ_5 is a subset of both $[\delta_2 \colon \top]$ and $[\delta_4 \colon \top]$ — that is, $[\delta_2 \colon \top] \sqcap [\delta_4 \colon \top] \sqsupseteq [\delta_5 \colon \top]$. In practice, the evolution diagram can always be reconstructed by drawing the subsumption lattice of the delta features.

$[\delta_2\uparrow, \delta_3\uparrow]$ — the set of both revisions Δ_0 and Δ_1. These sort definitions are kept up to date after each change to the revision set, so that the user need not bother about excluding later changes.

Using delta features, we may now select versions

- according to their respective features (e.g. $[f\colon \mathit{function}]$ — that is, a version with f being a function), which leaves us the choice of the specific revision; or
- according to the changes applied (e.g. $[\delta_1\colon \top, \delta_5\uparrow]$ — that is, Δ_1 or a descendant thereof, but not Δ_5 or one of its descendants), which leaves us the choice of the specific variants; or
- according to both (e.g. $[\Delta_4, f\colon \mathit{procedure}]$ — that is, the variant of the Δ_4 revision where f is a procedure).

The mechanisms introduced in previous sections apply to revisions as well: we may view multiple revisions at a time, using the CPP representation, and incrementally select revisions and variants using interactive panels. Anytime, we are free to choose and refine arbitrary version sets as they evolve in the software process.

7 Related Work

Feature-like *attribution schemes* using name/value-pairs as attributes are widespread across CM systems. In the *attributed file system* (ATFS) of Lampen and Mahler [LM88], each component can have an arbitrary conjunction of user-defined attributes; components are selected by a *pattern* containing the desired values. The *option space* of Lie *et al.* [LCD$^+$89], allows for storing and retrieving objects giving a set of *options*, where each option can be either *true* or *false*. None of these approaches is incremental by nature; ambiguities are not allowed. In contrast, Nicklin's *context model* [Nic91] allows *undefined* options and arbitrary boolean operations and can thus handle ambiguous and incomplete specifications. However, incremental issues and deductive abilities are not mentioned.

For an *attribution methodology*, see the successful *faceted classification scheme* founded by Prieto-Diaz in [PD87].

The ADELE system by Estublier [Est85, Est88] performs *consistency checking* for variants by evaluating boolean constraints containing attribute equations. In contrast to our approach, ADELE follows only one alternative by choosing the "best satisfying" variant; system consistency constraints (called *generic consistency rules*) are given explicitly on a global level. The CMA system by Ploedereder and Fergany [PF89] allows incremental consistency checking; the consistency notion must be coded by the user.

Speaking of *version sets*, the *variant-specific editor* by Narayanaswamy [Nar89] gives individual views on either individual versions or the whole set; versions are identified using CPP-like **@if**-directives. There is no support for arbitrary version sets. In [SB93], Singleton and Brereton mention CPP constraints as a potentially useful application of partial evaluation, but without going into details of how this could be done.

The concepts of *revisions and variants* are usually strictly separated [Tic85, Rei89]. However, Rich and Solomon, who use feature logic for system modelling [RS91], anticipated "version *(i.e. revision)* selection as a further instantiation of object terms" — which is presented in this paper.

8 Conclusion

We have seen that use of feature logic and feature unification allows for incremental and interactive configuration management; often, complete configuration threads can be deduced from incomplete specifications, and inconsistencies are detected earlier than in conventional tools. Version sets can be represented as CPP files; adaption to other configuration management formalisms (e.g. RCS [Tic85] or *shape* [LM88]) is straight-forward and under way.

The Incremental Configuration Engine ICE is part of the inference-based software development environment NORA, which aims at utilizing unification theory and inference technology in software tools. In particular, NORA offers a tool for inferring configuration structures from source code, which also detects suspicious interferences between configuration threads [KS94]; this tool, which is called NORA/RECS, acts complementary to NORA/ICE.

A UNIX stand-alone versions of ICE is available via anonymous FTP from **ftp.ips.cs.tu-bs.de: /pub/local/softech/ice**. RECS and other parts of NORA are found in the **recs** and **nora** directories.

Acknowledgements

Lars Düning designed textual merging of version sets and implemented the CPP-style storage system. Ragnar Stahl built the gemeric feature panel editor shown here, based on an initial program by Thorsten Sommer.

Bernd Fischer, Petra Funk, Franz-Josef Grosch, Matthias Kievernagel, and Christian Lindig provided valuable comments on early revisions of this paper.

NORA is funded by the Deutsche Forschungsgemeinschaft, grants Sn11/1-2 and Sn11/2-1.

References

[BL84] R. J. Brachman and H. J. Levesque. The tractability of subsumption in frame-based description languages. In *Proc. of the 4th National Conference of the American Association for Artificial Intelligence*, pages 34–37, Austin, Texas, August 1984.

[Est85] J. Estublier. A configuration manager: The ADELE data base of programs. In *Proc. of the Workshop on Software Engineering Environments for Programming-in-the-Large*, pages 140–147, June 1985.

[Est88] J. Estublier. Configuration management: The notion and the tools. In *Proc. of the International Workshop on Software Version and Configuration*, January 1988.

[GS93] Franz-Josef Grosch and Gregor Snelting. Polymorphic components for monomorphic languages. In Rubén Prieto-Diaz and William B. Frakes, editors, *Proc. of the 2nd International Workshop on Software Reusability*, pages 47–55, Lucca, Italy, March 1993. IEEE Computer Society Press.

[Kay84] M. Kay. Functional unification grammar: A formalism for machine translation. In *Proc. 10th International Joint Conference on Artificial Intelligence*, pages 75–78, 1984.

[KR86] R. T. Kasper and W. C. Rounds. A logical semantics for feature structures. In *Proc. of the 24th Annual Meeting of the ACL*, pages 257–265, Columbia University, New York, 1986.

[KS94] Maren Krone and Gregor Snelting. On the inference of configuration structures from source code. In *Proc. 16th International Conference on Software Engineering*. IEEE Computer Society Press, 1994. To appear.

[LCD$^+$89] Anund Lie, Reidar Conradi, Tor M. Didriksen, Even-André Karlsson, Svein O. Hallsteinsen, and Per Holager. Change oriented versioning in a software engineering database. In *Proc. 2nd International Workshop on Software Configuration Management*, pages 56–65. ACM Press, October 1989.

[LM88] Andreas Lampen and Axel Mahler. An object base for attributed software objects. In *Proc. of the Fall '88 EUUG Conference*, pages 95–105, October 1988.

[Nar89] K. Narayanaswamy. A text-based representation for program variants. In *Proc. 2nd International Workshop on Software Configuration Management*, pages 30–33. ACM Press, October 1989.

[Nic91] Peter Nicklin. Managing multi-variant software configurations. In Peter H. Feiler, editor, *Proc. 3rd International Workshop on Software Configuration Management*, pages 53–57. ACM Press, June 1991.

[NS90] B. Nebel and G. Smolka. Representation and reasoning with attributive descriptions. In K. Bläsius, U. Hedstück, and C.-R. Rollinger, editors, *Sorts and Types in Artificial Intelligence*, volume 418 of *Lecture Notes in Computer Science*, pages 112–139. Springer-Verlag, 1990.

[PD87] Rubén Prieto-Diaz. Classifying software for reusability. *IEEE Software*, 4(1), January 1987.

[PF89] Erhard Ploedereder and Adel Fergany. The data model of the configuration management assistant. In *Proc. 2nd International Workshop on Software Configuration Management*, pages 5–13. ACM Press, October 1989.

[Rei89] Christoph Reichenberger. Orthogonal version management. In *Proc. 2nd International Workshop on Software Configuration Management*, pages 137–140. ACM Press, October 1989.

[RS91] Anthony Rich and Marvin Solomon. A logic-based approach to system modelling. In Peter H. Feiler, editor, *Proc. 3rd International Workshop on Software Configuration Management*, pages 84–93. ACM Press, June 1991.

[SAK90] Gerd Smolka and Hassan Aït-Kaci. Inheritance hierarchies: Semantics and unification. In Claude Kirchner, editor, *Unification*, pages 489–516. Academic Press, London, 1990.

[SB93] Paul Singleton and Pearl Brereton. A case for declarative programming-in-the-large. Technical Report tr93-14, Keele University, GB, June 1993.

[SFG$^+$94] Gregor Snelting, Bernd Fischer, Franz-Josef Grosch, Matthias Kievernagel, and Andreas Zeller. Die inferenzbasierte Softwareentwicklungsumgebung NORA. *Informatik – Forschung und Entwicklung*, 1994. To appear.

[SGS91] Gregor Snelting, Franz-Josef Grosch, and Ulrik Schroeder. Inference-based support for programming in the large. In A. van Lamsweerde and A. Fugetta, editors, *Proc. 3rd European Software Engineering Conference*, volume 550 of *Lecture Notes in Computer Science*, pages 396–408. Springer-Verlag, October 1991.

[Smo92] Gert Smolka. Feature-constrained logics for unification grammars. *Journal of Logic Programming*, 12:51–87, 1992.

[SUP$^+$83] S. Shieber, H. Uszkorzeit, F. Pereira, J. Robinson, and M. Tyson. The formalism and implementation of PATR-II. In J. Bresnan, editor, *Research on Interactive Acquisition and Use of Knowledge*. SRI International, 1983.

[Tic85] Walter F. Tichy. RCS—A system for version control. *Software—Practice and Experience*, 15(7):637–654, July 1985.

[Win87] Jürgen F.H. Winkler. Version control in families of large programs. In *Proc. 9th International Conference on Software Engineering*, pages 91–105. IEEE Computer Society Press, March 1987.