Knowledge Controlled Version Selection in Software Configuration Management

Pavol Návrat, Mária Bieliková
Slovak Technical University, Dept. of Comp. Sci. and Engineering,
Ilkovičova 3, 812 19 Bratislava, Slovakia
E-mail: {navrat,bielikova}@elf.stuba.sk
Tel:(+ 42 7) 791 395
Fax:(+ 42 7) 720 415

Abstract

In the paper, the question of effective automated version selection is tackled. It is an important part of a method for building a software configuration. The paper concentrates on the method for version selection which allows specifying requirements which refer to properties of versions. It is controlled by knowledge in form of heuristic functions serving as necessary condition and suitability condition for selection. We present the method and describe its application in selecting software components.

Keywords: software configuration management, version control, selection controlled by heuristics

1 Introduction

The problem of version selection is a topic of much interest in the recent research within the area of software configuration management. Solving it is important for achieving quality of the configuration being built, but it also influences efficiency of the process of building a software system configuration.

Requirements for version selection can have various forms:

- *an empty description*, i.e. a version is selected solely on grounds of default requirements "hard-wired" in the system, e.g. *"select the most recently formed version"*, cf. [17],
- *an explicit identification*, i.e. a name of the component to be included is given, e.g. in the form of a table, cf. [9],
- *description by attributes*, i.e. a version is described implicitly by stating properties which it should fulfil, cf. [4, 8, 5, 19].

The empty description has an obvious disadvantage of being too inaccurate. The explicit description can lead to difficulties in practical situations when the number of components is high and to produce a list of all those which should be included is very error-prone. As the most suitable way appears to be the description by attributes. In fact, if we consider a name to be an attribute as well, we can treat the explicit identification to be just a special case of the description by attributes.

The component's attributes can be e.g. date of formation, author, status. Using such attributes, we can formulate the following requirement:

> *select a version which is being elaborated (status = check_out) by the author, who is the current user; if there is not such a one then select the most recent completed (status = check_in) version which has been already tested.*

The selection requirements formulate actually a query to a database of software components. To evaluate them, database query processing techniques can also be used, e.g. [1, 16].

Frequently used approach to version selection is to use conditions restricting properties of versions, e.g. [18, 4, 8, 5]. The conditions are often represented by a logic expression. For example, an expression

$(operating\_system = DOS \ \wedge \ communication\_language = Slovak)$

identifies all such versions which can run under DOS operating system and the communication with the user is in Slovak language. The ADELE system [5] is an example illustrating this view of version selection.

The language of logic expressions is sometimes enhanced by allowing defaults, conditional selections, and by introducing three-valued logic (i.e.*true, false, undefined*), cf. [13].

Zeller and Snelting in [19] propose for modelling version sets a unified approach based on feature logic. Version sets are identified by their features, that is, a boolean expression over $(name : value)$ attributes. This approach subsumes all the above mentioned approaches to identify versions of components.

In [1], version selection is based on logical conditions referring to values of attributes, too. Here, preferences can be specified as well. Preferences are in fact logical conditions which act as filters.

The system SHAPE, cf. [11] uses a similar way of selecting versions as in [1]. It is based on using a selection rule. The selection rule is a named sequence of alternatives. Predicates in alternatives allow to expres requirements with respect to the object attributes. If no selection rule is specified, the fixed default rule is active which is the same as in Make (*"select the busy object in the current directory"*).

System GYPSY [4] uses predicate to obtain a subset of the component versions. In case the resulting set has more than one element, the final selection is the result of applying a single default rule, i.e. *"select the most recent version"*. In [4], no precaution was made to handle the case when some or all of the filters failed to produce a nonempty set.

Another frequently used approach is to use rules. For example in the system DSEE [10] there is defined a set of rules which are interpreted sequentially until the sought component is selected. The language for writing rules allows defining default rules, dynamic rules (e.g. select the most recent version), and conditional rules (if-then). Despite the fact that such kinds of rules allow powerful means of selection, their power is limited in the DSEE system by the fact that the set of attributes is set in advance.

Our goal is to devise a method for version selection which offers a framework for specifying various heuristics describing which versions are to be preferred.

The paper is organised as follows. In the section 2, we give an outline of our approach. Then the new method for version selection is presented in a more formal way (section 3). An example demonstrating the application of our method is given in the section 4. We discuss results of evaluation of its performance in the section 5. Our concern for software version selection is part of a larger project aimed at developing method for building a software configuration. Version selection is in some sense the core of any such method. We give in the section 6 a very brief description of its context. The paper closes with our conclusions which also summarize the points in our approach that are new when compared to related works.

2 An outline of our approach

When either more than one version complies with the requirements, or none of the versions does difficulties in the process of configuration building can arise. When analyzing the problem of version selection, there are many similarities to be observed to problems approached by artificial intelligence techniques. Specifically, to evaluate the alternatives a heuristic information can be employed. The heuristic information can be expressed e.g. in form of a heuristic function which assigns to each alternative an evaluation from some well ordered set, most frequently an integer or real number. The evaluation expresses the suitability and perspective of selecting the particular alternative in the actual state. Recently, symbolic heuristics, often combined with various methods of numerical evaluation are used as well.

When devising the heuristic function, in most cases it suffices to evaluate the given alternative relatively to other admissible alternatives. In case of the problem of selecting a software component's version, an explicit formulation of a heuristic function that would define an ordering of versions according to their suitability is quite difficult. To accomplish selection, the heuristic function must reflect several aspects such as the nature of a software system, the given requirements, and version properties. The aspects should be assigned corresponding weights which would reflect their relative importance with respect to other aspects reflected in the heuristic function.

After having analyzed the problem of version selection, and having taken into account the known approaches to selecting alternatives of solution, it appears to be more advantageous not to strive for arranging explicitly the possible component versions according to their suitability ordering, but to delete "the least suitable" versions subsequently from the set of admissible versions instead, similarly to [6]. The requirements for version selection can be expressed as a sequence of heuristic functions which reduce the set of suitable versions. We can express the relative importance of a given evaluating criterion by modifying the order in which the heuristic functions are applied.

We have distributed the requirements for version selection into two parts:

- *the necessary selection condition*, which must be satisfied by every version selected as a potential candidate. The condition can be expressed by a heuristic function which maps a set of all versions into a set of admissible versions. The necessary condition can be either a condition that should be satisfied by all versions (e.g. that the operating system is DOS), or a condition that should be satisfied by no version (e.g. the state is inconsistent),
- *the suitability selection condition*, which is used in step by step reduction of the set of admissible versions aiming to select a single version. The condition is represented by heuristic functions $h_1, h_2, \ldots, h_n$.

One example of requirements for version selection may be as follows:

- *the necessary selection condition*
  h_0 : operating_system = DOS $\wedge$ communication_language = Slovak
- *the suitability selection condition*
  h_1 : problem_type = design $\vee$ algorithm = simple
  h_2 : communication_language $\neq$ undefined
  h_3 : prefer version with a greater number of defined attributes
  h_4 : operating_system $\neq$ undefined
  h_5 : programming_language = Prolog
  h_6 : prefer version with a smaller number of defined architectural relations with other components.

The heuristic functions represent knowledge about the degree of suitability of the respective versions. They refer to properties of versions as defined by their attributes. A heuristic function can often be evaluated separately for each version, thus it can have the form

$h : V \rightarrow \{ satisfies, does\_not\_satisfy\}$,

where V is a set of versions and h is a heuristic function. Requirements for version selection formulated in a similar fashion can be found in many existing systems, e.g. [18, 4, 8, 5].

Heuristic functions can also express knowledge captured during the software system development. Examples of such knowledge are

- prefer a version with the greatest number of defined attributes,
- prefer a version with the greatest number of defined attributes occuring in the necessary selection condition,
- prefer a version with the greatest number of defined attributes occuring in the suitability selection condition,
- prefer a version included in the greatest number of formed configurations,
- prefer a version which is involved in the least number of architectural relations with other components (in the context of the whole configuration),
- prefer a version which is involved in the least number of architectural relations with components which have not been included in the configuration being built,
- prefer more general versions

  Determining the generality/specificity relation, i.e. deciding whether a version V_1 is special case of a version V_2 is based on considering attributes with definite values (i.e. attributes with a values different from *undefined*) of those versions. A version V_1 is a special case of a version V_2 if

  - V_1 has equal or greater number of attributes with definite values as V_2,
  - there corresponds to each attribute with a definite value of V_1 an attribute of V_2 with definite value equal the above value,
  - sets of attributes (i.e., names along with values) of V_1 and V_2 are not equal.

  The above rule is based on the observation that if some attribute of some component is *undefined*, it satisfies any required value for that attribute. Therefore, a version with a greater number of *undefined* attributes is more general and has bigger chance to be accepted in the resulting configuration.

The heuristic for version selection described above, and other similar ones cannot be described by a function of the form $h : V \to \{satisfies, does\_not\_satisfy\}$ (the one that can be expressed by a logic expression, with atoms representing relations over attributes of components). The thing is these are the properties of versions which can only be investigated on sets of versions as a whole. We define therefore heuristic functions to be of the form $h : 2^V \to 2^V$, where V is a set of versions and h is a heuristic function.

Let us sketch an approach to version selection which makes use of such heuristic functions. First, the set of all versions will be reduced by applying the necessary selection condition into a set of admissible versions. Next, version selection continues as a successive application of the heuristic functions from the suitability selection condition, serving as filters until all the heuristic functions in the sequence are exhausted or until by applying one of them yields a one element set, with the element being the desired version.

If, after applying all the filters we get a set of more then one version, the final choice must be made in another way. As one option, a software engineer could intervene by entering into an interaction with the system. Important thing is that even in this case the set may indeed become reduced in that the number of elements is smaller than in the original set of all versions. Of course, it depends on the heuristic functions and actual properties of versions.

If, after applying some of the heuristic functions acting as filters we get an empty set, it is necessary to return to the previous set of admissible versions and to apply the next filter in the row, i.e. the filter which reduces to an empty set shall not be considered.

3 The method

The method relies on explicit formulation of the knowledge related to version selection. Specifically, the software engineer writes down the requirements which will in fact control the process of version selection.

Input to the method is:

- a set M of all available versions
- requirements on version selection
  - a necessary selection condition represented by a heuristic function h_0,
  - a suitability selection condition represented by a sequence of heuristic functions $[h_1, h_2, \ldots, h_n]$, where

  $h_j : 2^M \rightarrow 2^M, \ 0 \leq j \leq n$

Output from the method is "the most suitable" version v, $(v \in M)$, or failure.

The method can be described by the following steps:

1. Apply the necessary selection condition to reduce the set M of all available versions into a set of admissible versions:
   (a) $suit_0 = h_0(M)$
   (b) if $suit_0 = \emptyset$ then **halt**, the method has not been successful
   (c) if $suit_0 = \{v\}$, i.e. the set of admissible versions has exactly one element then **halt**, the method has been successful and the output is the version v
   (d) otherwise, continue
2. Apply the heuristic functions $[h_1, h_2, \ldots, h_n]$ in the order of their appearence to the actual set of admissible versions:
   (a) $j := 1$
   (b) apply the heuristic function (filter) h_j to the set $suit_{j-1}$ forming a set

$$suit_j = \begin{cases} h_j(suit_{j-1}), & \text{if } h_j(suit_{j-1}) \neq \emptyset \\ suit_{j-1}, & \text{otherwise} \end{cases}$$

   (c) if $suit_j = \{v\}$, i.e. the actual set of admissible versions has exactly one element then **halt**, the method has been successful and the output is the version v
   (d) if $j = n$ then **halt**, the method has been only partly successful so far. To determine its output, a version $v \in suit_n$ shall be found using some default way
   (e) $j := j + 1$ and continue with 2b.

From the above it is clear that the order of heuristic functions in the suitability selection condition is important and it reflects the relative importance of particular requirements represented by heuristic functions. The earlier a function is positioned, the more important it is. The order of heuristics is in fact a kind of control heuristic, i.e. a meta heuristic.

Sometimes, however, it is not necessary, or reasonable, or even possible to order a subset of heuristics, i.e. to decide about how important they are relatively to each other. We have therefore introduced two constructors for writing the admissibility condition. Besides a constructor for a sequence (meta heuristic corresponding to ordering according to importance), there is another constructor for a set (meta heuristic corresponding to a nondeterministic order of application). Syntax of the former is $[h_1, h_2, \ldots]$, syntax of the latter is $\{h_1, h_2, \ldots\}$. The constructors can be arbitrarily nested. In such a way, more complex meta heuristics can be specified. For example, suitability condition specified as $[h_1, \{h_2, h_3\}, h_4, \{h_5, h_6\}]$ prescribes that heuristic function h_1 will be applied first, then any one from h_2 and h_3, then the other one from these two, then h_4, then functions h_5 and h_6 in an arbitrary order.

There is an important difference between necessary selection condition and suitability selection condition. Whereas the necessary selection condition obviously must be satisfied, the suitability selection condition serves only to select "the most suitable" version from among those admissible ones. Therefore, if application of the former results in an empty set, there is no solution. On the other hand, if application of the latter (more precisely, of any heuristic function from the expression, i.e. of a filter) results in an empty set, the filter is simply ignored.

There occurs sometimes during the process of selecting "the best" version a situation when selecting a "less suitable" version would be desirable. For example, it could happen when decisions made during configuration building are to be revised later. This calls for complementing the above outlined way of filters' application with a way of selecting "less suitable" versions.

Let the necessary selection condition be represented by a heuristic function h_0 and the suitability selection condition be represented by an expression $[h_1, h_2, \ldots, h_n]$. Let the set of versions be denoted by M. By applying the heuristic functions h_j there are formed step by step sets $suit_j$, $unsuit_j$, $1 \leq j \leq n$, cf. Figure 1. The following holds:

$suit_{j-1} = suit_j \cup unsuit_j$ and

$M = suit_0 \cup unsuit_0$

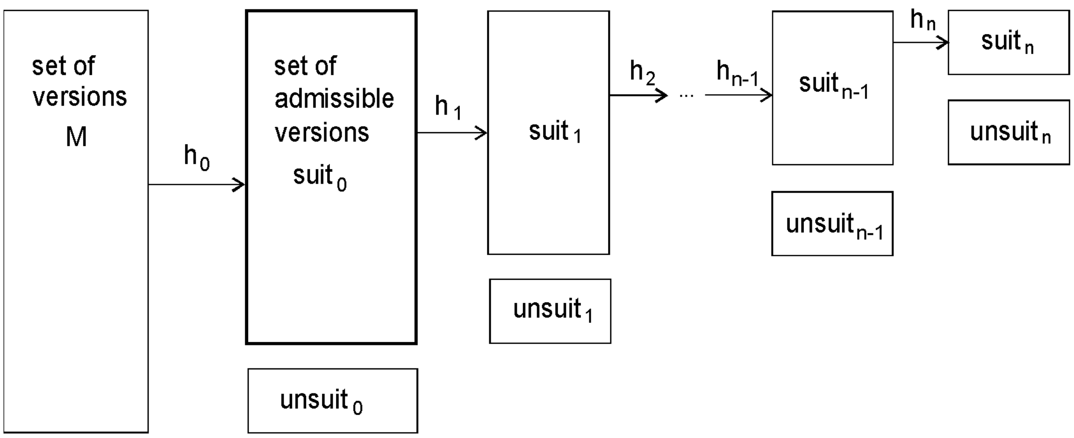

Figure 1: Step by step application of heuristic functions

Let us assume the selection process (involving application of heuristic functions as described above) results in a set $suit_i$, i.e. h_i was the last heuristic function applied. The set $suit_i$ contains "the best" versions. The set $unsuit_i$ contains those versions which were selected by all the heuristic functions h_j, $1 \leq j < i$ and were rejected only by the filter h_i. That is why the versions in $unsuit_i$ are those which are considered to be those "less suitable" than versions from $suit_i$, but "more suitable" than all the other ones. If we denote the ordering by the symbol $\gg$, we can write: $suit_i \gg unsuit_i$ (the expression $A \gg B$ for sets A and B states the fact that for any element $a \in A$ and any element $b \in B$, we have $a \gg b$).

Summing up, due to the fact that $suit_{i-1} = suit_i \cup unsuit_i$, it is reasonable to consider the set $unsuit_i$ which contains versions satisfying the greatest number of filters from among the remaining sets (i.e., besides those in $suit_i$).

The versions grouped in subsets of the original set of all versions can thus be ordered in the following way (from "the most suitable" to "the least suitable"):

$suit_i \gg unsuit_{i-1} \gg \ldots \gg unsuit_2 \gg unsuit_1$.

The set $unsuit_0$ contains versions which do not satisfy the necessary selection condition. Such versions are not acceptable as solutions, so they cannot be ordered among the more or less suitable versions.

Next, we need to arrange versions in order within the respective sets. Let us assume the selection process has terminated with the set $suit_i$ as the result of the last applied heuristic function h_i. There are three cases to be considered for the set $suit_i$:

1. $suit_i = \emptyset$. This can only happen when $i = 0$. No filter from the suitability selection condition has been applied. No version satisfies the necessary selection condition. There are no versions to be ordered.
2. $suit_i$ is a one element set. We have $i \leq n$ and because there is only one element in the set, it is also the most suitable element.
3. $suit_i$ contains more than one element. We have $i = n$, i.e. there have been applied all the filters. To arrange in order the versions in $suit_n$, some default procedure must be applied, e.g.
   - selection based on order of occurence,
   - random selection,
   - selection based on decision of the software engineer.

Now, let us turn our attention to the sets $unsuit_j, 1 \leq j \leq n$. To arrange versions in order within them, we can either apply one of the default ways, or we can make use of those filters which have not yet been applied to the respective sets. The cases to be considered are:

1. $unsuit_j$ is either empty or singleton set. Nothing to arrange.
2. $unsuit_j$ has more than one element, and $j = n$. The selection process terminated. There was applied the last heuristic function h_n from the sequence $[h_1, h_2, \ldots, h_n]$ to the set $suit_{n-1}$. Versions not satisfying the filter were left in $unsuit_n$. There are no more filters to apply. Ordering can be achieved only through some default way (e.g., see above).
3. $unsuit_j$ has more than one element, and $j < n$. There exist still filters which have not been applied to $unsuit_j$, i.e. heuristic functions $h_{j+1}, h_{j+2}, \ldots, h_n$. We apply suitability selection condition $[h_{j+1}, h_{j+2}, \ldots, h_n]$ and arrange elements in order in $unsuit_j$ in such a way. After exhausting all the filters without achieving a complete ordering some default way must be used.

Applying the above described procedure results in a complete ordering of the set of admissible versions defined by the given heuristic functions, with a possible amendement provided in some default way.

The assumption that suitability selection condition is formed solely by using the sequence constructor is not a real restriction. There can always be formed such an expression from any expression formed using both sequence and set constructors. In case of a set, there can always be devised a sequence of filters from that set such that each filter will occur in it once and only once.

4 Example

Let us present a relatively modest but sufficiently instructive example in order to demonstrate important features of our method. We assume the goal is to select the most suitable version from among a set of versions grouped in a family MANAGER. The family is a set of seven components:

$$M = \{ MANAGER.1, MANAGER.2, MANAGER.3, MANAGER.4, MANAGER.5, MANAGER.6, MANAGER.7\}$$

The components along with their properties are listed in Table 1. The requirements are expressed by the heuristic functions $h_0, \ldots, h_6$ given as in the example above (section 2). The suitability selection condition is represented by an expression $[h_1, h_2, h_3, h_4, h_5, h_6]$. For simplicity, we assume that all the versions have the same number of architectural relations to other components.

name	**operating system**	**problem type**	**communication language**	**algorithm**	**programming language**
$MANAGER.1$	DOS	$design$	$undefined$	$simple$	C
$MANAGER.2$	DOS	$design$	$undefined$	$undefined$	$Prolog$
$MANAGER.3$	DOS	$diagnosis$	$undefined$	$complex$	$Prolog$
$MANAGER.4$	DOS	$diagnosis$	$undefined$	$complex$	$Prolog$
$MANAGER.5$	$undefined$	$diagnosis$	$undefined$	$simple$	$Prolog$
$MANAGER.6$	$UNIX$	$design$	$undefined$	$undefined$	C
$MANAGER.7$	DOS	$design$	$undefined$	$complex$	$Prolog$

Table 1: Properties of the versions of the MANAGER family of software components

The selection proceeds as follows:

1. Set of admissible versions $suit_0$ is formed by an application of the necessary selection condition h_0: $operating\_system = DOS \wedge communication\_language = Slovak$

   $$suit_0 = h_0(M) = \{ MANAGER.1, MANAGER.2, MANAGER.3, MANAGER.4, MANAGER.5, MANAGER.7\}$$

   The version MANAGER.6 has been removed because its attribute *operating_system* has different value as the one required (DOS vs. $UNIX$).

2. Set of admissible versions is subsequently transformed by applying suitability selection conditions:

   - Application of h_1: $problem\_type = design \vee algorithm = simple$

     $$suit_1 = h_1(suit_0) = \{ MANAGER.1, MANAGER.2, MANAGER.5, MANAGER.7\}$$

     Versions MANAGER.3 and MANAGER.4 have both values of the attributes *design* and *problem_type* set to different values as required by the function h_1.

- Application of h_2: *communication_language* $\neq$ *undefined*

$$\begin{aligned} h_2(suit_1) &= \emptyset, \quad \text{therefore} \\ suit_2 = suit_1 &= \{MANAGER.1, MANAGER.2, \\ &\quad MANAGER.5, MANAGER.7\} \end{aligned}$$

- Application of h_3: *prefer version with a greater number of defined attributes*

$$suit_3 = \{MANAGER.1, MANAGER.7\}$$

- Application of h_4: *operating_system* $\neq$ *undefined*

$$suit_4 = \{MANAGER.1, MANAGER.7\}$$

- Application of h_5: *programming_language* $=$ *Prolog*

$$suit_5 = \{MANAGER.7\}$$

Application of the suitability selection condition terminated with forming a one element set after applying the filter h_5. As the most suitable version has been selected the version $MANAGER.7$.

5 Evaluation

We have analyzed the proposed method in order to estimate its efficiency. It seems reasonable to consider an application of a heuristic function to be the operation which contributes most significantly to the overall effort. Let V be a set all of versions, and h be a heuristic function $h : 2^V \rightarrow 2^V$. Let S be a set of versions, $S \subset V$. Application of the function h results in forming a set $h(S) \subset S$. Input to the method is a set of versions M and selection requirements represented by an expresssion $[h_0, h_1, \ldots, h_{n-1}]$, where h_0 is the necessary selection condition. Selectivity of the heuristic functions varies in general, i.e. the probability P_i that a particular version will be selected as a result of an application of the heuristic function h_i can vary across the different heuristic functions. The worst case is when there are to be applied all the heuristic functions from the selection requirements and no application of a heuristic function reduces the set of admissible versions. Let us denote the number of versions in set M by m. In the worst case, there must be evaluated (i.e., applied) $E_W = m * n$ heuristic functions.

In the average case, there are E_A applications of heuristic functions:

$$\begin{aligned} E_A(m, n, [P_0, P_1, \ldots, P_{n-1}]) &= (1 - P_0)^m + \\ \sum_{i=1}^{i=m} (m + E_A(i, n-1, [P_1, \ldots, P_{n-1}])) * C_i(m) * P_0^i * (1 - P_0)^{m-i}, & \\ & \quad \text{for } n > 1, m > 1 \\ &= 0, \text{ for } m = 1 \\ &= m, \text{ for } n = 1 \end{aligned}$$

where P_j is the probability that $v \in S \Rightarrow v \in h_j(S)$ for any version v, $0 \leq j \leq n-1$.
$C_i(m)$ denotes the number of ways a combination of m objects taken i at a time can be made.

The above formula takes into account the fact that the first heuristic function h_0 represents the necessary selection condition. Should it result in an empty set, there are no more filters applied.

We present in Figure 2 how the average number of applications of filters denoted by E_A depends on the number of heuristic functions n, assuming the number of versions is invariant. We assume further that the probabilities P_j, $0 \leq j \leq n-1$ were generated randomly. Dashed lines depict the functional dependencies in case the selection requirements are just a sequence of filters, with no necessary selection condition included. The graph shows that the value of E_A is not sensitive to a change of the number of filters included in the selection requirements for $m > 5$.

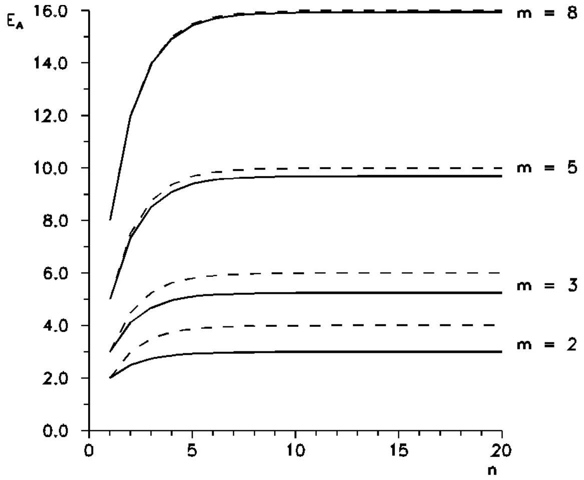

Figure 2: The average number of applications of heuristic functions E_A vs. the number of heuristic functions n

A complementary view is presented in Figure 3 which shows how the average number of applications of filters E_A depends on the number of versions m, assuming the number of filters is invariant. The case when the selection requirements are just a selection of filters is not depicted as the line is close to the depicted one up to the third decimal digit. The graph shows that the value of E_A linearly depends on the number of versions.

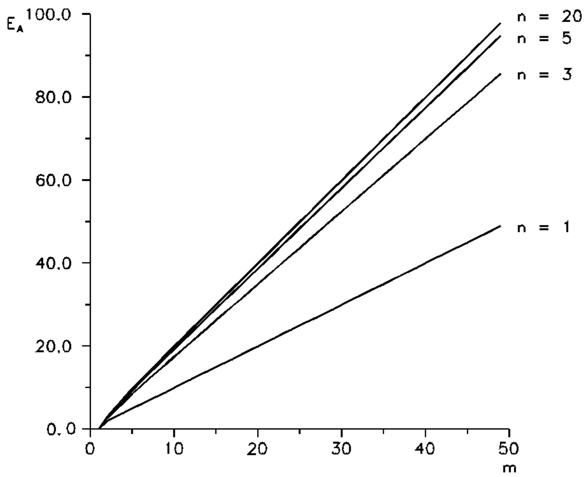

Figure 3: The average number of applications of heuristic functions E_A vs. the number of versions m

6 Building a configuration

We have stated right in the beginning of this paper that our motivation for devising a method for version selection is determined by the need to select versions, be it variants or revisions in the process of building configurations of software systems.

A software system configuration is a set of software components which satisfy the requirements and are consistent. We adopted the composition model of configuration management [3]. A configuration in this model consists of a system model and version selection requirements. A system model is used for listing all the components that make up a system. Version selection requirements indicate which version is to be chosen for each component to make up a configuration.

The form of the model is influenced by the intended application. In software configuration management, the software system is frequently modelled by a graph in various variations. Besides an acyclic graph and a tree, some authors also use an AND/OR graph [18, 5]. We find the AND/OR graphs suitable in supporting the process of software system configuration building.

In [2] we described the model of a software system as an AND/OR graph. In our model, we assume that there are two kinds of versions: variants and revisions. In the AND/OR graph, only variants, along with component families are represented. A family of software components comprises versions of a software component. Actually, we can always consider versions to be revisions. However, they are grouped into subsets called variants within a family. This is a conceptual novelty comparing to related works, e.g. [14], where revisions and variants are treated at the same level as components. Similar concepts were employed in [15], where there is proposed an orthogonal organization of variants and revisions.

A variety of architectural relations can be defined between variants and families. The fact that architectural relations can be defined between variants and families allows our model to be "more generic" as e.g. those of [1, 10, 11, 18] in the sense that several models are usually needed to describe the information contained in our model. In particular, they allow architectural relations to be defined only between component families.

Being essentially a graph search, it is inevitable to have a method for selecting a proper variant (i.e., a subset of software versions). It results in a generic configuration defined in terms of variants. Only after selecting from within every variant a particular component (i.e., a revision) we get a bound configuration. Therefore the role of a method for version selection is crucial for configuration building.

Instead of giving a detailed description of the model and our method for configuration building (the interested reader may refer to the original papers [2, 12]), we present the role of the method for version selection in its context. Actually, our method for configuration building makes use of the presented method for version selection twice. First, a variant is selected for each family of software components which is to be included in the configuration. Second, a revision is selected for each variant.

We find the separation important. It allows the user to formulate possibly different requirements for selection of variants and revisions. Moreover, the process of selection is simplified. For example, let us assume that there are v variants in a family of software components, and r revisions on average in each of them. Let the requirements for variant selection contain a elementary conditions, and let the requirements for revision selection contain b elementary conditions. In an unseparated selection process, there are $E_u = (a + b) * v * r$ tests to be performed in the worst case. In a separated selection process, there are $E_s = a * v + b * r$ tests to be performed. Obviously,

$$E_u = (a + b) * v * r > a * v + b * r = E_s, \text{for } a, b, v, r > 0$$

Important is the fact that our method for version selection can fail to select the most suitable version automatically. This means complication to automation of configuration building but reflects more adequately software develompent needs.

There is important requirement for configuration being built: versions of different component must be selected in a consistent manner. The system ADELE [5] takes this into account by propagating constraints resulting from former version selections. We adopted the similar approach. Consistency of the configuration is guaranteed by the method of configuration building and therefore all the components being included in the configuration must satisfy constraints specified in both configuration requirement and in the particular software components. While devising algorithm for implementing our method of configuration building, we were facing essentially the problem of searching AND/OR graph with constraints. This led us to techniques similar to those used in truth maintenance systems, and finally to devising a programming technique for implementing such algorithms, which uses markings to maintain consistency and identification the reason for a deadend [12].

We have developed an experimental implementation of the method for building a configuration which uses method for version selection in Prolog. A logic formalism, like any declarative formalism in general, is an excellent tool to support browsing and reasoning about versions of objects, relationships and dependencies [7]. In this experimental implementation, heuristic functions are represented by logic expressions and special functions described as Prolog predicates. For example, the heuristic *"prefer more general versions"* is represented by the predicate *prefer_general/1* which constrains the current set of admissible versions to those having the most general values of attributes.

7 Conclusions

The presented method for version selection allows selecting the most suitable version according to given requirements referencing properties of versions. Our approach offers to a software engineer a framework for specifying various heuristics describing which versions are to be prefered. Using heuristic functions not only makes the process potentially capable of building better configurations, but also documents preferences applied in selections.

In our approach, heuristic functions allow to investigate properties of versions on sets of versions as a whole. As far as we know, none of the mentioned approaches to version selection can describe a requirement on sets of versions as the above mentioned heuristic *"prefer more general versions"*. In the system SHAPE [11] one can use in selection rule very restricted form of selection on sets of versions by predefined predicate *attrmax* which requires object to have a maximal value in the specified attribute.

Our approach is similar to version selection in DSEE [10]. What is new is the distribution of requirements for version selection into the necessary selection condition and the suitability selection conditions along with their defining their interpretation. The distinction becomes important e.g., in selecting variants. Most of the requirement conditions are in fact just recommendations, so they can be formulated as suitability selection conditions. However, there are often certain requirements which must not be ignored, so they are formulated as the necessary selection condition.

In the version selection used by the ADELE system, if the set of versions becomes empty the user is notified that the given description is inconsistent or ambiguous. In our approach, if after applying some of the heuristic functions acting as filters we get an empty set, the filter which is responsible for it is not to be considered.

Another important issue is the possibility of explicit formulation of the relevant knowledge on version selection. The heuristic functions form at the same time a documentation and record the justifications why particular components were selected.

Our method can fail to select the most suitable version automatically. Therefore, it calls in such cases for an intervention from a software engineer. The requirements may be revised, or less suitable version could be sought.

One open problem is acquiring programming and problem knowledge on the suitability of component versions. In cases when attributes of components are not known no matter for what reason, methods of reverse engineeing could be attempted to supply them.

References

[1] Y. Bernard, M. Lacroix, P. Lavency, and M. Vanhoedenaghe. Configuration management in an open environment. In *Proc. 2nd European Software Engineering Conference*, pages 35–43. Springer-Verlag, 1987.

[2] M. Bieliková and P. Návrat. Modelling software systems in configuration management. *Applied Mathematics and Computer Science*, 5(4):751–764, 1995.

[3] A. Brown, S. Dart, P. Feiler, and K. Wallnau. The state of automated configuration management. Annual technical review, Software Engineering Institute, Carnegie Mellon University, Pennsylvania, 1991.

[4] E.S. Cohen, D.A. Soni, R. Gluecker, W.M. Hasling, R.W. Schwanke, and M.E. Wagner. Version management in Gypsy. In P. Hederson, editor, *Proc. ACM SIGSOFT'88*, pages 201–215, Boston, 1988. ACM Press.

[5] J. Estublier. The Adele configuration manager. Technical report, L.G.I., Grenoble, 1992.

[6] S.F. Fickas. Automating the transformational development of software. *IEEE Trans. on Software Engineering*, SE-11(11):1268–1277, 1985.

[7] A. Jazaa. Toward better software automation. *Software Engineering Notes*, 20(1):79 – 84, 1995.

[8] J. Kimball and A. Larson. Epochs, configuration schema and version cursors in the KBSA framework CCM model. In P.H. Feiler, editor, *Proc. of the 3rd Int. Workshop on Software Configuration Management*, pages 33–42. ACM SIGSOFT, 1991.

[9] G.D. Korn and E. Krell. A new dimension for the unix file system. *Software - Practice and Experience*, 20 (S1):S1/19 – S1/34, 1990.

[10] D.B. Leblang and R.P. Chase. Parallel software configuration management in a network environment. *IEEE Software*, 4(6):28–35, 1987.

[11] A. Mahler and A. Lampen. An integrated toolset for engineering software configurations. In P. Hederson, editor, *Proc. ACM SIGSOFT'88*, pages 191–200, Boston, 1988. ACM Press.

[12] P. Návrat and M. Bieliková. An approach to automated building of software system configuration. Technical report, Slovak Technical University, Bratislava, 1995.

[13] P.J. Nicklin. Managing multi-variant software configurations. In P.H. Feiler, editor, *Proc. of the 3rd Int. Workshop on Software Configuration Management*, pages 53–57. ACM SIGSOFT, 1991.

[14] J. Plaice and W.W. Wadge. A new approach to version control. *IEEE Transactions on Software Engineering*, 19(3):268–275, 1993.

[15] C. Reichenberger. Concepts and techniques for software version control. *Software - Concepts and Tools*, 15(3):97–104, 1994.

[16] C. Sheedy. Sorceress - a database approach to software configuration management. In P.H. Feiler, editor, *Proc. of the 3rd Int. Workshop on Software Configuration Management*, pages 121–126. ACM SIGSOFT, 1991.

[17] W.F. Tichy. RCS - a system for version control. *Software - Practice and Experience*, 15(7):637–654, 1985.

[18] W.F. Tichy. Tools for software configuration management. In *Proc. Int. Workshop on Software Version and Configuration Control*, pages 1–20, Stuttgart, 1988.

[19] A. Zeller and G. Snelting. Handling versions sets through feature logic. In W. Schäfer and P. Botella, editors, *Proc. 5th European Software Engineering Conference*, pages 191–204. Springer-Verlag, 1995.