A Knowledge Based Method for Building a Software System Configuration

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

A method for building a software configuration from its model and requirements describing its properties is proposed. Our model of software systems reflects architectural and development-induced relations among component families and variants. The method builds a generic configuration first and then proceeds to building a bound one. In building both of them, a method for version selection plays an important role. It is controlled by heuristics supplied by a software engineer. We present both methods. We also present results of experimental evaluation of the method for version control. The results support our hypothesis that more selective filters are to be preferred in order to increase efficiency.
Keywords: software configuration management, version control, heuristic control knowledge
Abbreviated title: Knowledge Based Configuration Building

Problem area and goal

Software systems consist of many components. During system development and maintenance, the components undergo changes. When attempting to build a system as a set of components, a decision must be made which components, and which versions of them should be included. The software engineer specifies either explicitly or implicitly which components to include. For software systems comprising several hundreds or thousands of components, the explicit specification becomes a burden. We attempted to tackle this problem by devising a method which would allow for a more convenient way of specification by means of stating the required properties of the system being built. Moreover, we attempted to propose a method which would offer means for incorporating heuristic knowledge to control the search process.

Our Approach

Versions

An important subclass of transformations causing the changes of components are those which preserve specification of a component and do not radically change the language. Such kind of transformations has been characterized[3] as meaning-preserving and lateral (producing a result at a similar level of abstraction).

Components resulting from such transformations are called versions. Similarly to other works in the area[6,9] we distinguish two kinds of versions: variants ('parallel' versions) and revisions ('serial' versions).

Notion of version is defined by the relation *is_version*, which holds between two components such that the second one was formed from the first one by applying the above described transformation. The relation *is_version* is reflexive, symmetric and transitive. It defines equivalence classes within the set of all components, each of which is described as a family of software components. Hence, a family is a set of all components which are versions of one another. Moreover, we can recognize within a family the binary relation *is_variant* determined by properties of software components (functional attributes, architectural relations, and constraints). The relation is an equivalence. Equivalence classes are called variants. We introduce a software component as a revision. We consider even the very first concretization of a variant a revision.

Software System Model

Relations between components can be either development-induced, such as e.g., *is_variant, has_revision*, or architectural, such as e.g., *depends_on, specifies, uses.* The architectural relations can be defined only between variants and families. As a consequence, any change of such relations during forming a new revision must result in a new variant. We assume that all revisions of a given variant have the same architectural relations.

The usual approach to building a configuration is to build a model of the software system first. Generally, various kinds of graphs are being used to model software systems. Tichy[10] and later Estublier[5] have presented a model based on A/O graphs. An important aspect stressed also by the latter work is reusability of the created configuration.

Our approach is to model software system by an A/O graph. For a family, a model represents links to all its variants. For a variant, the model should represent links to all such families that are referred to in the variant architectural relations. When referring to a software system model, we speak of a pair (N, E), denoting an A/O graph with set N of nodes and set E of edges. Each node is either an A-node or an O-node. An O-node represents a family. Its successors in graph are variants. Every O-node has at least one successor. An A-node represents a variant. Every A-node has exactly one predecessor. On every path, A-nodes and O-nodes alternate.

Building software system configurations

We understand, in accordance with most of the literature, the notion of a software system configuration to be a set of components which is complete, consistent and satisfying the required properties. In our terminology, this corresponds to the notion of a bound configuration.

Our approach to building a software system configuration makes use of some ideas that have been developed in the field.[8,2,5,1] When building a configuration, exactly one variant is to be included for each family found to be included in the configuration. Moreover, exactly all families are to be included for each variant found to be included in the configuration. Because the revisions have not been taken into consideration so far, the resulting configuration is still a generic one. Only after a revision has been selected for each variant included in the generic configuration, we have a bound configuration as a result.

There can be built several different configurations from a model of the software system, usually based on different required purposes of the desired configuration. There can be desired a configuration for an end user, a configuration for futher development, etc. Such configurations can be specified by different configuration requirements. In order to build a configuration, our method takes into account knowledge of the architectural relations between components, knowledge of selecting components (families) and also of selecting variants and revions for each family.

Selecting a version

During the process of configuration building according to our method, there are two occasions when selecting a version takes place. We have devised a method to handle selecting in general which is auxiliary to the method for configuration building. It uses heuristic knowledge.

The method for version selection is employed in solving two tasks. First, it is used to select a variant (i.e., a set of components equivalent according to their architectural properties and constraints). Second, the same method is used in selecting a revision (i.e., a component from among those included in one variant).

Our approach is based on observation that in the case of software components selection, it is difficult to express a heuristic function which would define an ordering of versions based on their suitability. We have found more advantageous not to attempt to order the alternatives (i.e. versions) according to their suitability, but rather to delete step by step those least suitable from the set of all admissible versions. Our strategy of version selection is based on a sequence of heuristic functions which reduce the set of suitable versions. By changing the order in which the heuristic functions are applied we can vary the importance of the evaluation criterion which the given function embodies.

Examples of such heuristics are *prefer version with smaller number of defined architectural relations with other components*, or $operating\_system = DOS \wedge communication\_language = Slovak$.

Method for version selection

Input to the method is:

- a set M of all available versions
- version selection requirement
  - 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 $h_j : 2^M \to 2^M,\ 0 \leq j \leq n$

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: $suit_0 = h_0(M)$.
   If $suit_0 = \emptyset$ then **halt**, the method has not been successful.
   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.
   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.

Method for building a configuration

Input to the method is:

- a software system model $M = (N, E)$ with roots $s_1, s_2, \ldots, s_m$,
- a generic configuration requirement $gcr_M = (Rel, VariantCond, ConfConstr)$, where Rel is a set of names of architectural relations, $VariantCond$ is an expression specifying condition for variant selection and $ConfConstr$ is an expression (built up from references to heuristic functions) specifying constraint for all components to be included in the configuration.
- a bound configuration requirement $bcr_M = (ExpCond, RevisionCond)$, where $ExpCond$ is an expression specifying a condition for selection of exported components (selected families) and $RevisionCond$ is an expression specifying the suitability selection condition for revisions.

Output from the method is:

- a generic configuration for the given model G_M, and
- a bound configuration B_{G_M} relative to the generic configuration, or
- failure.

In the sequel, we present the method by describing the inputs and outputs to the corresponding steps:

1. *Forming a generic configuration*

   (a) **Input:**

   - a software system model $M = (N, E)$, and
   - a generic configuration requirement gcr_M

   Output: a graph $F = (FN, FE)$, where $FN \subseteq N, FE \subseteq E$.
   There is formed a subgraph F of the A/O graph M modelling the system by considering the relations specified in $gcr_M.Rel$. In F, all the A-nodes represent only relations given in $gcr_M.Rel$.

(b) **Input:**

- a graph $F = (FN, FE)$ formed in step 1a,
- generic configuration requirement gcr_M

Output: generic configuration $G_M = (U, H)$, where $U \subseteq FN, H \subseteq FE$.
There is formed a generic configuration by selecting exactly one successor to each O-node (according to variant selection requirement $gcr_M.VariantCond$), and by selecting all the successors to each A-node included in the graph. All the nodes being included in the graph G_M must also satisfy the constraints specified in both the generic configuration requirement $gcr_M.ConfConstr$ and in the particular nodes. To select a successor to an O-node n, our method for version selection is applied with the following inputs:

- a set of versions $MU = \{x | x \in FN \wedge (n, x) \in FE\}$, i.e. all successors of node n
- $gcr_M.VariantCond$ serving as the version selection requirement.

Output is the selected version, i.e. one successor of the node n.

2. *Forming a bound configuration*

(a) **Input:**

- a generic configuration $G_M = (U, H)$ formed in step 1b,
- a bound configuration requirement bcr_M

Output: a set of exported variants VE.
There is formed a set of exported variants by selecting all variants from the generic configuration G_M formed in the step 1b such that they satisfy the condition $bcr_M.ExpCond$ for exported components.

(b) **Input:**

- a set of exported variants VE formed in step 2a,
- a bound configuration requirement bcr_M

Output: a bound configuration, i.e. a set of software components V.
There is formed a bound configuration by selecting a revision for each variant from the set VE formed in the step 2a such that it satisfies the revision selection condition specified in $bcr_M.RevisionCond$. To select a revision for a variant v, the method of version selection is applied with the following inputs:

- a set of versions in consideration $MU = \{k | k \in v\}$,
- $bcr_M.RevisionCond$ taken as the version selection requirement.

Output is the selected version, i.e. a software component (revision) included in the variant v.

Experiments

We have analyzed the method for version selection independently in order to evaluate 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.

Input to the method is a set of versions M and a selection requirement 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 requirement 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.

We report elsewhere[7] on our results of an analysis how on average number of applications of heuristic functions (denoted as E_A) depends on the number of heuristic functions (filters) denoted as n, and on the number of versions on the input of the method, denoted as m. The results show that the value of E_A is not sensitive to a change of the number of filters included in the selection requirement for $n > 5$ and that E_A depends approximately linearly on the number of versions.

Efficiency of the proposed method depends not only on the number of versions and the number of heuristic functions, but also on the heuristic functions themselves and on properties of versions to which heuristic functions are applied. We present in Figure 1 how the average number of applications of filters (E_A) changes with the change of the probability P_i that a particular version will be selected as a result of an application of the heuristic function h_i. We assume that all the heuristic functions included in the selection requirement have an equal probability. Dashed lines depict functional dependencies in the case the selection requirement is just a sequence of filters, with no necessary selection condition included. The graphs (the left one parametrized by number of versions m, the right one parametrized by number of filters n) show that the method for version selection is more effective for heuristic functions with a smaller probability that a particular version will be selected by an application of the heuristic function.

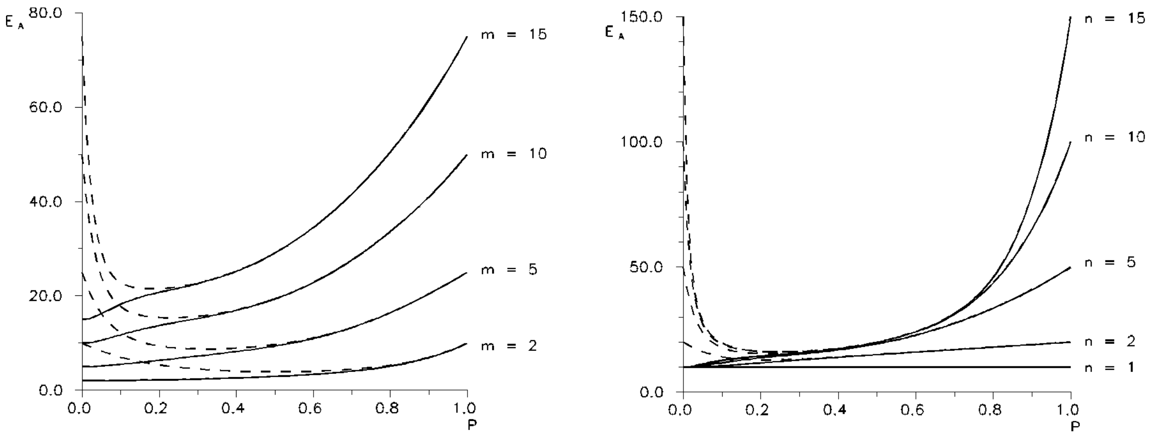

Figure 1: The average number of applications of heuristic functions E_A vs. the probability P.

Let us note that heuristic functions are important not only from the efficiency point of view, but from the point of a quality of the configuration being built as well. There is a trade-off between these two criteria, with the quality being more important in our view.

Conclusions

Our approach to divide the building process into building a generic configuration first, and building a bound configuration later supports reusability. Frequently, the same generic configuration can be used to generate different configurations.

In version selection, using heuristic functions not only makes the process potentially more efficient, but also documents the preferences applied in selections. We have performed numerous experiments aiming to investigate various aspects of selection method efficiency. The results show that selection with heuristics requires on average, in fact almost always (except of such extreme cases as one component system) less effort than without them.

The implementation of our approach has been an exciting research endeavour in itself. We have implemented it in Prolog. However, we found suitable to devise programming techniques based on such known concepts as meta-programming, reflection, or non-chronological search to facilitate the implementation work.[4]

References

[1]N. Belkhatir and M. Ahmed-Nacer. Major issues on PSEE: Process software engineering environments. *Computers and Artificial Intelligence*, 12(3):279–298, 1993.

[2]K. Berrada, F. Lopez, and R. Minot. VMCM, a PCTE based version and configuration management system. In J. Feiler, editor, *Proc. of the 3rd Int. Workshop on Software Configuration Management*, pages 43–52, 1991.

[3]V. Berzins, Luqi, and A. Yehudai. Using transformations in specification-based prototyping. *IEEE Transactions on Software Engineering*, 19(5):436 – 452, 1993.

[4]M. Bieliková and P. Návrat. A programming technique of implementing multi-level logic programs. Technical report, Slovak Technical University, Bratislava, 1995.

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

[6]Luqi. A graph model for software evolution. *Trans. on Software Engineering*, 16(8):917–927, 1990.

[7]P. Návrat and M. Bieliková. Knowledge controlled version selection in software configuration management. Technical report, Slovak Technical University, Bratislava, 1995.

[8]F. Oquendo, K. Berrada, F. Gallo, R. Minot, and I. Thomas. Version management in the PACT integrated software engineering environment. In *Proc. European Software Engineering Conference ESEC'89*, pages 222–242. Springer-Verlag, 1989. LNCS 387.

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

[10]W.F. Tichy. A data model for programming support environments and its application. In B. Langefors, A.A. Verrijn-Stuart, and G. Bracchi, editors, *Trends in Information Systems*, pages 219–236. North Holland, 1986.