APPROACH TO IMPROVING SOFTWARE CONFIGURATION MANAGEMENT

Summary:
Software configuration management (SCM) is one of the areas closely related to achieving and maintaining quality of software in a broader sense. Our approach to the problem of building a software system configuration is based on an assumption that explicit repesentation of important properties of software components can be utilized for both improving the process of building the configuration, and also in a certain sense, for improving the configuration (the product) itself, i.e. its quality. Our goal is to devise a method for building software system configuration from the system's model and requirements. We describe the proposed way of modelling software systems, its components and configuration requirement. We present a new method for building configurations and report on experiments and their evaluation.

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

1. Problem area and goal

Software quality improvement is now an objective of most organizations. There is a growing need to develop and adapt approaches to quality management, to define appropriate procedures and standards and to check that these are followed by all engineers. Our contribution to improving software quality is based on a known assumption that quality of the software development process affects the quality of delivered software product. The relationship between software development process and software product is rather complex. Changing the process does not necessarily lead to improved software quality. Nevertheless, quality of software development process is considered important (Rombach, Verlage, 1995).

Software configuration management (SCM) is one of the areas closely related to achieving and maintaining quality of software by improving some aspects of software process (Frühauf, 1990). I. Sommerville in (Sommerville, 1996) states: *Configuration management may be a part of a more general software quality management process. In some organizations, the same manager may share quality management and configuration management responsibilities.*

One of the important problems in SCM is how to build configurations of software systems. The problem is a consequence of the fact that software is complex both as a product and as a process of its development. Software systems consist of many components which are naturally simpler than the whole, reducing the complexity in that way. In the course of system development and maintenance, the components undergo changes. When attempting to build the system as a set of components, a decision must be made which components, and which versions of them should be included. However, the space of configurations is extremely large even for modest systems. For example, assuming a rather small system with 100 components and with 2 versions for each of them, we have 2^{100} different versions of the software system (i.e., system configurations). The thing is that practically only very few of them are useful, either as results of a purposeful maintenance or as subsets of futher development. To solve the problem how to find a desired configuration of the software system without having to search the space of all possible versions is therefore a very practical question. Important issue is the quality of the configuration to be built. Our understanding of the notion of the configuration quality reflects also the extent to which a configuration meets requirements stated by a software engineer and as a consequence it is usable for the desired purpose.

Our research is concerned with the problem of building a software system configuration which satisfies requirements imposed upon it by the software engineer. Our approach to the above problem is based on an assumption (i.e., an initial hypothesis) that explicit representation of important properties of software components can be advantageous for both improving the process of building the configuration, and also in a certain sense, for improving the configuration (the product) itself. The software engineer describes the desired configuration by a set of requirements (i.e. implicitly) rather than by having the burden of listing (explicitly) all of them.

Our goal is to devise a method for building software system configuration from the system's model and requirements. We describe the proposed way of modelling software systems, its components and configuration requirement. We present a new method for building configurations and report on experiments and their evaluation.

2. Structure of a software component

Software components are the basic elements of a software system. When attempting to describe them, it is instructive to bear in mind that a software system is being created in a development process which can be viewed as a sequence of transformations. An important subclass of transformations causing the changes of components are those which transform one component just to one and do not radically change the language. Components resulting from such transformations are called versions. In accordance with relevant literature (Luqi, 1990; Plaice, Wadge, 1993; Reichenberger, 1994) we distinguish two kinds of versions: variants ('parallel' versions) and revisions ('serial' versions). Variants of software components are alternative implementations of a particular concept. Revisions, on the other hand, are modifications intended to replace the previous version. A family of software components comprises all components which are versions of one another.
We can identify different relations between software components within a software system. Relations can be of two kinds: (1) relations expressing the system's architecture, concerned especially with a funcionality of the components and a structure of the system, such as *depends_on*, *specifies*, *uses*, (2) relations expressing certain aspects of the system's development process, with important consequences especially for the version management, such as *is_variant*, *has_revision*.

Let us start now to formalise the concepts outlined above.
For a software system S we have a set $COMPONENT_S$ of components which are from S. Next, a binary relation $is\_versions_S \subseteq COMPONENT_S \times COMPONENT_S$

is given as the reflexive and transitive closure of another binary relation which is defined by elementary transformations describing such modifications of software components that they can still be considered to be expressing essentially the same concept.

The set of all equivalence classes induced by the relation $is\_version_S$ is denoted $FAMILY_S$ and called a set of families of software components of the software system S. An element of $FAMILY_S$ is called a *family* of software components.

Next, we focus our attention to the structure of a software component itself. Based on that, we can define variants as sets of those components which share certain attributes.

We call *a software component* a quintuple c_S:

$c_S = \{ArchRel, FunAttr, CompAttr, Constr, Realis\}$,

where $ArchRel$ is a set of pairs consisting of a name $RelationId$ of an architectural relation and a name $FamilyId$ of a family, $FunAttr$ is a set of functional attributes, $CompAttr$ is a set of other attributes of that component, $Constr$ is an expression for the constraint, and $Realis$ is the actual text of the component.

From the point of view of the software configuration management is the realisation part of only secondary importance. More important are the relations between components, component's properties and the constraint.

For example, let us consider a software component $c1$, for which there exists an architectural relation $uses$ with a family of software components PRINT, and a relation $has\_document$ with a family DOCUM:

$$
\begin{array}{ll}
c1 = (\{(uses, PRINT), (has\_document, DOCUM)\}, & \%\text{architectural relations} \\
\quad \{(phase, implementation), (operating\_sys, DOS), & \%\text{functional attributes} \\
\quad (prog\_language, C), (algorithm, simple), & \\
\quad (communication\_language, slovak)\}, & \\
\quad \{(author, kate), (date, 95\_02\_07), (status, tested)\}, & \%\text{other attributes} \\
\quad (parameters = ordered) \Rightarrow (sys\_ver = DOS\_6\_2), & \%\text{constraint} \\
\quad \#define... \quad) & \%\text{program in C language}
\end{array}
$$

In order to describe variants, we define a binary relation $is\_variant$ which determines a set of software components with the same architectural relations, functional attributes and constraints within a given family.

We define a binary relation $is\_variant_S \subseteq COMPONENT_S \times COMPONENT_S$:

$$
\begin{aligned}
x \ is\_variant_S \ y \Leftrightarrow x \ is\_version_S \ y \ \wedge x.ArchRel = y.ArchRel \ \wedge \\
x.FunAttr = y.FunAttr \wedge x.Constr = y.Constr
\end{aligned}
$$

It can be easily seen that the relation $is\_variant$ is an equivalence. A set of all equivalence classes in the relation $is\_variant_S$ will be denoted by $VARIANT_S$ and called a set of variants of a software system S. An element of the set $VARIANT_S$ is called a *variant*.

Variants are important to simplify management of software component versions in selecting a revision of some component, or in building a configuration. We can treat a whole group of components in a uniform way due to the fact that all of them have the relevant properties defined as equal.

We introduce a software component as a revision. According to our concept, even the very first concretization of a variant is called a revision. Based on that, elements of a software system are organised in a hierarchy. Families of software components comprise variants and variants comprise revisions. Architectural relations are defined at the level of variants (they are the same for all revisions within a variant) between variant and a family of software components.

Let us stress once more that we consider families to be equivalence classes. Variants are sets of components, i.e. revisions. This is a conceptual novelty comparing to related works, e.g. (Plaice, Wadge, 1993; Reichenberger, 1994) where revisions and variants are treated at the same level as components.

3. Model of a software system

The concepts introduced above allow us to formulate a model of a software system. When proposing it, we attempted to find a model which would support the process of configuration building. That is why in our model, those parts and relations of a software system are represented explicitly which are crucial in the process of configuration building.

It can be seen from the above that revisions do not have the 'sovereignty' to maintain own architectural relations. All their relations are completely determined by the variant they belong to. This observation is very important because it allows us to simplify the situation and to include into the software system model only two kinds of elements: families and variants.

From the point of view of a family, the model should represent families and variants included in them. From the point of view of a variant, the model should represent architectural relations which are defined for each variant. When building a configuration, for each family already included in a configuration there must be selected precisely one variant. For each variant already included in a configuration, there must be included all the families related by architectural relations to that variant.

Our method of modelling a software system S is to describe it by an oriented graph $M_S = (N, E)$, with nodes representing families (called O-nodes) and variants (called A-nodes) in such a way that these two kinds of nodes alternate on every path.

Edges originating in a variant (A-node) represent architectural relations. Edges from family to variant mirror $has\_variant$ relation. Such graphs are denoted as A/O graphs.

As an example, let us include an A/O graph, see Figure 1 depicting a software system with families A, B, C, D, E. Each family includes several software components which form variants, e.g. family A includes three sets of components (variants).

Software systems are frequently modelled by graphs, cf. (Tichy, 1988; Estublier, 1992). However, none of the related works defines architectural relationships between variants and families. A more radically different approach to modelling software systems was taken by Holt (Holt, Mancoridis, 1994) who uses so called tube graphs. A more detailed description of our approach to modelling software systems is presented in (Bieliková, Návrat, 1995a).

A model of a software system describes all the possible configurations. The model of software system which we presented above simplifies greatly the problem of building a configuration. The problem can be formulated in terms of graph searching. The usual interpretation is that the A-nodes are origins of edges leading to nodes, all of which must be considered provided the A-node is under consideration (logical AND).

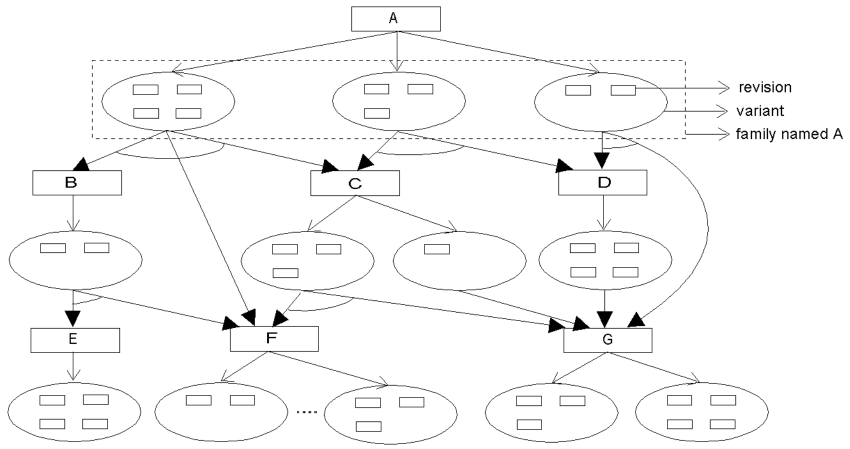

Figure 1: Example of a software system model.

Similarly, the O-nodes are origins of edges leading to nodes, from among which exactly one must be considered provided the O-node is under consideration (logical OR).

Any configuration we build by searching the model can only be a generic one because it can identify several configurations of the software system. A configuration of a software system built solely from software components, i.e. revisions, is called a bound one. This organization provides for a high degree of reuse. When building a new configuration, we can reuse the current generic configuration as long as all the changes have been revisions within the desired generic configuration. Only when the change resulted in modifying the set of variants, also a new generic configuration must be built.

Let $M = (N, E)$ be a model of a software system. Then *a generic configuration* for a given model is an oriented graph $G_M = (U, H)$, where $U \subseteq N, H \subseteq E$, such that it includes exactly one successor of each O-node (i.e. family) included in it. Moreover, it includes at least one O-node (i.e. family) and at least one A-node (i.e. variant).

A bound configuration for a given generic configuration $G_M = (U, H)$ is a set of software components B_{G_M} such that at most one revision for each variant in the generic configuration is selected.

4. Process of configuration building

We understand, in accordance with most of the literature, the notion of 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 bound configuration.

Our method of building a software system configuration makes use of some ideas from previous works of others (Tichy, 1988; Oquendo, Berrada, et al., 1989; Berrada, Lopez, Minot, 1991; Estublier, 1992; Belkhatir, Ahmed-Nacer, 1993). The method describes a procedure how to find the set of components included in the bound configuration. During the course of procedure application, a generic configuration is to be formed. This product can be used when a change of the software system is attempted. Reusing it frees us from the necessity of creating the configuration from the scratch next time.

Being essentially a graph search, it is inevitable to have a method for selecting a proper variant or revision. Having taken into account the known approaches to selecting alternatives of solution (Leblang, Chase, 1987; Tichy, 1988; Cohen, Soni, et al, 1988; Kimball, Larson, 1991; Estublier, 1992) we have devised method for version selection which relies on explicit formulation of the knowledge related to version selection (Návrat, Bieliková, 1995). We understand the strategy of version selection to be based on a sequence of heuristic functions which reduce the set of suitable versions as identified by the software component family.

Heuristic functions are evaluated in two steps: (1) applying the necessary selection condition and (2) applying the suitability selection condition. The necessary selection condition must be satisfied by every version selected as a potential candidate. The suitability selection condition is used in step by step reduction of the set of admissible versions aiming to select a single version.

Input to the method for building a configuration is:

- a software system model $M = (N, E)$ with roots $s_1, s_2, \dots, s_m$,
- 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 (necessary selection condition and suitability selection condition), 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, and $RevisionCond$ is an expression specifying the suitability selection condition for revisions.

Output from the method for building a configuration 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:** software system model $M = (N, E)$, and
   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. For selecting successor to an O-node, our method for version selection is used.

The method of version selection for O-node n is applied with the following inputs: (i) a set $MU = \{x | x \in FN \wedge (n, x) \in FE\}$, (ii) $gcr_M.VariantCond$ serving as the selection requirement. Output is the selected element from the set MU, i.e. a successor of the node n.

2. *Forming a bound configuration*

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

Output: 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 exported components condition $bcr_M.ExpCond$.

(b) **Input:** set of exported variants VE formed in step 2a,
bound configuration requirement bcr_M

Output: 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 $bcr_M.RevisionCond$. The method for version selection is applied.
In this case, the method of version selection for the variant v is applied with the following inputs: (i) a set $MU = \{k | k \in v\}$, (ii) $bcr_M.RevisionCond$ taken as the selection requirement. Output is the selected element from the set MU, i.e. the software component (revision) included in the variant v.

Main strengths of our approach to configuration building are (1) consideration of the conceptual distinction between variants and revisions, (2) consideration of architectural relations at the variant level and (3) distribution of requirements to several parts (a condition for selection of exported components and a set of names of architectural relations). To the best of our knowledge only system ADELE (Estublier, 1992) allows consideration of the subset of defined architectural relations in a system model. Other approaches simple consider all parts defined in a system model.

In version selection, 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.

5. Experiments

The proposed methods have been implemented in order to perform experiments aiming to analyze their properties. 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 (Jazaa, 1995).

The implementation endeavour became an interesting research theme by itself. While devising algorithm implementing our method, we were facing essentially the problem of searching A/O 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. It attempts to find a place in the graph where the search for an alternative solution should be resumed.

As an independent measure to test efficiency we have chosen to count the number of consistency checks performed during searching a particular graph. An alternative measure can be the number of deadends occuring during searching together with the number of visited nodes. In fact, we have experimented with all of these measures. Similar measure was used also by (Freuder, Wallace, 1992; Dechter, Meiri, 1994). Due to space limitations, we report only on some results using the former measure in this paper. A more detailed description is reported in (Bieliková, Návrat, 1995b).

Let us present one of our experimental results in Figure 2. The aim is to keep the number of performed consistency checks as small as possible. On horizontal axis, there is number of consistency checks (TESTC) as performed by the standard searching algorithm with chronological backtracking (denoted as CHB). On vertical axis, there is an average number of consistency checks performed by the new proposed algorithms applied to those problems for which number of consistency checks by CHB is given by horizontal axis. The first new algorithm is a special version of a dependency-directed backtracking to resolve the deadend situation by analyzing the reasons of inconsistency (denoted as RB). The second new algorithm is an enhanced RB algorithm with marking mechanism (denoted as RB-M) which allows recording and propagating results of an analysis of the current deadend node.

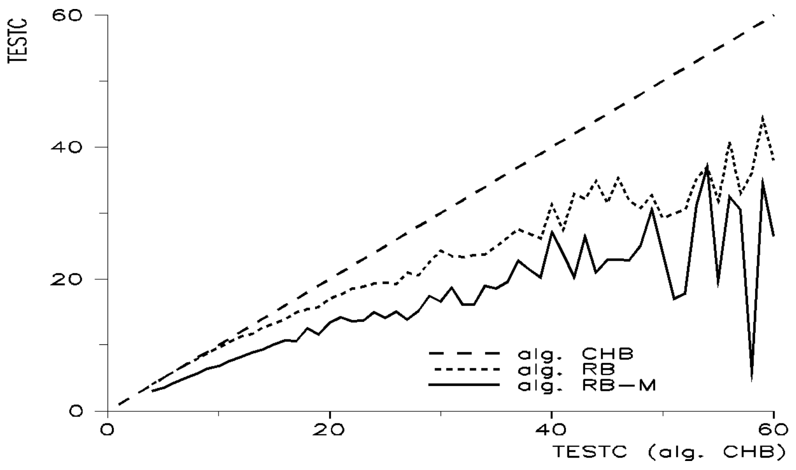

Figure 2: Number of consistency checks TESTC.

The presented graph is based on results of searching about 9000 generated graphs. For all experiments, the values representing our new algorithms RB and RB-M are always lower than values representing the standard algorithm. Moreover, the enhanced new algorithm RB-M performs slightly better than RB.

When reading the graph, we recognize that it is significantly smoother at lower values of TESTC. The reason for it is that generating randomly problems for which TESTC is higher was much less frequent. As a consequence, such points are results of fewer values taken into an average, sometimes even representing just a single problem.

6. Conclusions

We have presented a new method for building software system configurations. We have devised a way of modelling software systems which is based on notions of software components, variants and revisions, and generic and bound configurations. In some aspects, our approach comes close to the system reported by (Estublier, 1992). The configuration can be specified implicitly, i.e. the requirements constrain the space of admissible configurations. In this aspect, the method involves a special algorithm to search an A/O graph. Our algorithm makes use of heuristic knowledge and according to the experimental evaluation performs better then the traditional backtrack searching most of the time. For the particular results, comparison to related works has been discussed throughout the paper at the relevant places of presentation.

Our method offers an improved comfort to the software engineer in this important software development and maintenance activity.

The area of software configuration building requires further research. Our method assumes the components are available in the moment the configuration is built. We did not tackle the problem of effective forming of the derived components in response to changes.

Another 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.

The proposed method could be incorporated into a CASE tool. The CASE tool, however, would have to support preserving and maintaining versions of software components. Here, the proposed model is to be used.

References

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

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.

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

M. Bieliková and P. Návrat. A Prolog technique of implementing dependency-directed backtracking. Tech. report, Slovak Technical University, Bratislava, 1995.

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.

R. Dechter and I. Meiri. Experimental evaluation of preprocessing algorithms for constraint satisfaction problems. *Artificial Intelligence*, 68:211–241, 1994.

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

K. Frühauf. Hygiene in software works: Software configuration management. In *Proc. of Second European Conference on Software Quality Assurance*, Norway, 1990.

E. Freuder and R.J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58:21–70, 1992.

R.C. Holt and S. Mancoridis. Using tube graphs to model architectural designs of software systems. Technical report, Toronto, 1994. Research report CSRI-304.

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

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.

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

Luqi. A graph model for software evolution. *IEEE Transactions on Software Engineering*, 16(8):917–927, 1990.

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

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.

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

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

H.D. Rombach and M. Verlage. Directions in software process research. *Advances in Computers*, 41:2–63, 1995.

I. Sommerville. *Software Engineering.* Addison-Wesley, 5th edition, 1996.

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