An Approach to Building Software Configuration Using Heuristic Knowledge

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

Abstract

The paper describes original research in the area of software configuration management. We assume a software system consists from versions of two kinds: variants and revisions. Our approach is based on defining a suitable model of the software system, formulating requirements for a configuration to be built, and finally building a configuration. The requirements are in fact conditions which constrain the solution. To build a configuration requires search in the model represented as A/O graph. We work with generic configurations which represent solution on a level of variants, and then with bound configurations which represent solution on a level of revisions. We proposed a method to build a configuration which uses heuristic knowledge and implemented it in Prolog. Our experiments show such a method performs better than without supporting knowledge.

Keywords: software system configuration, heuristic knowledge, version control

1 Introduction

Software system changes more often than usually admitted. In fact, the changes are to be considered more a rule than an exception. Reasons for the unstability are not only possible errors in it which are to be corrected, but first of all the changing nature of the surrounding world to which the software system should be permanently adopted in the process of so called maintenance.

Software systems consist of many components which may undergo changes, as they indeed frequently do. These modifications are usually incremental, so it appears to be reasonable to consider results of such modifications to be versions of particular components rather than independent objects. In a traditional model of versions [9, 3] two different kinds of modifications are reflected in two kinds of versions: revisions and variants. Similar partitioning of versions is respected also in a change oriented model [4].

Revisions result from modifications which are caused by error correction, functionality enhancement, and/or adaptation to changes in environment. They develop in time sequence, with each next one usually intended to replace the previous one.

Variants of software components can be described as alternative implementations of a particular concept [7]. New variant represents an alternative solution. Variants usually exist concurrently. Variants result from experimental development or modifications towards ameliarating properties of the system.

A family of software components comprises a set of such versions of a software component that were formed by gradual modifications of the given component. A software system configuration is a set of software components which is consistent.

When specifying requirements for a configuration of a software system, we adopted the approach which is accepted universally, e.g. [9, 3], i.e. a configuration is built with respect to the relation *depends_on*, or its variants or refinements like *uses_type_definition*,

calls_procedure, *uses_variable*, etc.

Software components can be, as far as their structure is concerned, either simple or composed. If a composed component is included in a configuration, all the components which are included in it should be included as well.

Of course, the resulting set must be consistent. Moreover, it should include only components which are actually desired. For example, a configuration intended for an end-user should include executable program and a user documentation. On the other hand, a configuration intended for a further development must obviously include source program modules, but also design documentation etc. as well.

The process of configuration building should respect the existence of versions of software components. It should allow the highest degree of reusability. It should also support building at least an incomplete configuration in case some components do not satisfy the conditions.

Our goal is to devise a method that would support the process of building a software system configuration. The resulting configuration should be conformant to given requirements. Assuming software system components usually have several variants with several revisions each, selecting a complete and consistent set of components satisfying given constraints is essentially a process of searching.

2   The method

In order to devise a method for building a software system configuration, we have found at least the following questions to be important:

- how to model a software system,

- which steps should be followed when building a configuration.

We shall describe our approach to each one of these questions.

2.1   Model of a software system

We attempt to describe a software system with the specific purpose in mind, i.e. to be used during development and maintenance, and specifically in building the software system configuration. Therefore, our model encompasses those parts of the system and those relations among them which are important for building a configuration.

When attempting to identify 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. Because the initial specification of the system does not and should not include details of the solution, the overall orientation of the transformations is from abstract towards concrete. However, this does not mean that each particular transformation and especially when applied to a particular subsystem or component is a concretisation. In fact, there are involved abstracting, generalizing, and specializing transformations as well. Let us mention importance of such kinds of transformations in software reuse, reverse engineering, etc.

From among all the possible kinds of transformations, it is important to distinguish all those which correspond to the notion of component version. Creating a component version can be done in one of two possible ways. First, versions are created to represent alternative solutions of the same specification. They differ in some attributes. Such 'parallel' versions, or variants, are frequently result of different specializations. Second, versions are created to represent improvements of previous ones. Such 'serial' versions, or revisions, are frequently result of concretizations of the same variant.

Versions can by identified by relation *is_version*. 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, i.e. the family is a set of all components which are versions of one another. However, we can recognize within a family a binary relation *is_the_same_variant*. The relation is an equivalence. The equivalence classes are called variants.

We introduce a software component as a revision. In our meaning, even the very first con-

cretization of a variant is called a revision. A software component consists of two parts: an interface and a realization. In fact, from the point of view of the software configuration management is the realization part of only secondary importance. More important are the relations between components and component's properties as defined in the interface. Here, we identify two parts. One part of the component's interface is a description of the variant, which is common for all revisions of that variant. Any change to it results in forming a new variant. The other part is the revision's own interface. Any change to only revision part results in forming a new revision.

Relations between software components can be of two kinds:

- relations expressing the system's architecture, concerned especially with the funcionality of the components and structure of the system, such as *depends_on, specifies, uses,*

- relations expressing certain aspects of the system's development process, with important consequences especially for the version management, such as *is_version, has_revision*, which we shall commonly refer to as relations of the version kind.

The architectural relations are 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.

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 a software system model only two kinds of elements: families and variants.

For a family, the model should represent links to all its variants (links are implicitly defined by *is_the_same_variant* relation). When building a configuration, exactly one such variant is to be included for each family found to be included in the configuration.

For a variant, the model should represent links to all those families which are referred to in that variant (links are defined by architectural relations). When building a configuration, precisely all such families are to be included for each variant found to be included in the configuration. It should be noted that when a family or a variant is found to be included in the configuration during the process of building it, ultimately precisely one software component will be included. The selection of variant and software component within variant being part of the version control subproblem.

Let us start now to use a graph terminology. For any software system, we have a set F of software families, and a set V of variants. The sets F and V are disjoint. Let us further assume there are defined two binary relations $A = V \times F$ (architectural), and $O = F \times V$ (denoting option, version). We define the software system model to be an oriented graph $M = (N, E)$, where $N = F \cup V$, and $E = A \cup O$. Any element of E, $(v_1, v_2) \in E$, called an edge, is of one from among the two mutually exclusive kinds. Either $e_1 \in V, e_2 \in F$, i.e. the edge is from A. In this case, the node e_1 is called the A-node. Or $e_1 \in F, e_2 \in V$, i.e. the edge is from O. In this case, the node e_1 is called the O-node. 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 and variants A.1,A.2,B.1, etc.

The usual interpretation is that A-nodes are origins of edges leading to nodes, all of which must be considered provided the A-node is under consideration (logical AND). Similarly, O-nodes are origins of edges leading to nodes, from among which precisely one must be considered provided the O-node is under consideration (logical OR).

To sum up, our method of modelling a software system is to describe it by an A/O graph, with nodes representing families and variants in such a way, that these two kinds of nodes alternate on any path. A model does not
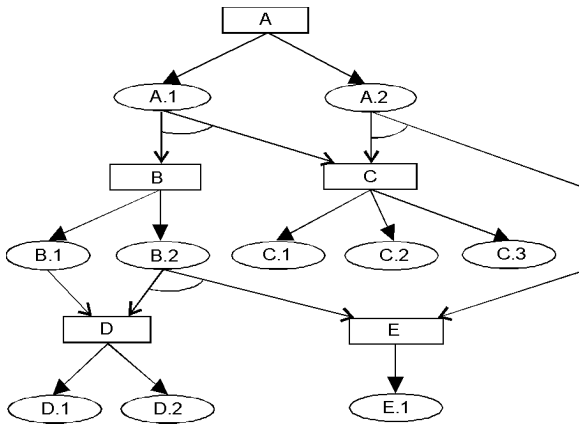
Figure 1: An example software system: partial hierarchy of elements.

take into account, however, revisions which are supposed to be mere implementations of variants. Similar concepts were employed in [8], who proposes an orthogonal organization of variants and revisions.

A model of a software system describes all the possible configurations. The space of configurations is extremely large even for a modest system. For example, assuming a rather small system with 100 families, and with 2 versions within each family, we have 2^{100} different versions of the system. The thing is, that practically only a very few of them are useful, either for maintenance or for further development. To solve the problem how to find a desired version of the software system (i.e., a configuration) without having to search the space of all possible versions is therefore a very practical question.

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. Besides simplifying the problem, the model also simplifies representation of the large set of possible configurations. Without such a model, large tables of configurations would have to be kept, which would be source of difficulties during maintenance.

Taking into account the fact that nodes in our model are component families and variants,

but not revisions (i.e., the actual components), it follows from it that any configuration we build by searching the model can only be a generic one. 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. A generic configuration consists from variants and it determines a set of bound configurations. To build a usable (bound) configuration from a generic one, one revision for each variant in the generic configuration must be selected.

The space of software system configurations is hierarchical, with two levels. One level comprises all possible generic configurations. For each generic configuration, the second level comprises all corresponding bounded configurations. 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 a change resulted in modifying the set of variants, also a new generic configuration must be built.

2.2 The process of building a configuration

In order to build a configuration which would meet the requirements, our method takes into account knowledge about the architectural relations between components and also about selecting components. The method must cope with three important tasks.

First subtask is to determine which component families and which variants shall be considered in building the configuration. Selection of component families is based on edges originating in A-nodes. A condition for selection of component families is a set of relations, $R \subset A$, which determine links from A-nodes in the system model.

Susequently, it is necessary to search this subgraph in such a way that for each A-node all successors are selected and for each O-node exactly one successor is selected. A successor to O-node (family) is its variant. A problem ari-

ses here in those special cases when either there are more than one variant satisfying the requirements, or there is no such variant at all. The problem resembles some other problems which are being tackled by artificial intelligence techniques. In evaluating suitability of possible alternatives of the solution, a heuristic information is used. It can be expressed e.g. in form of a heuristic function which assigns to each alternative a value from some well ordered set. The value estimates how suitable or promising it is to select the given alternative in the actual state.

In the case of software component selection, it is difficult to express a heuristic function which would define an ordering of versions based on their suitability. There must be taken into account various aspects, such as what kind of software system is being built, what are the requirements and properties of versions. The aspects should be assigned weights according to their relative importance.

We have found it more advantageous not to attempt to order the alternatives (i.e., component variants) according to their suitability, but rather to delete step by step those least suitable from the set of all possible versions. 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. 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.

We have distributed the requirements (i.e., the heuristic functions) for version selection into two groups:

- necessary selection condition, which must be satisfied by any potentially selected version. It is expressed by a heuristic function h_0 which builds from a set of all versions a set of all admissible versions. Necessary condition can be either condition which must be satisfied by all versions, or which must not be satisfied by any version,

- selection suitability condition, which is

used for a step by step reduction of the set of admissible versions with the aim of selecting one version. The condition is represented by a sequence of heuristic functions $(h_1, h_2, ..., h_n)$.

Heuristic functions represent knowledge about the degree of suitability of particular versions. They refer to properties of those versions as expressed by their attributes. We define heuristic function h to be a mapping:
$h : 2^V \rightarrow 2^V$ where V is set of all versions.

Heuristic functions can also express our knowledge from software engineering, e.g.

- prefer a version with the greatest number of defined attributes,

- prefer a version which is architecturally related to a least number of those components which have so far not been included in the configuration being built.

The particular ordering of the heuristic functions in our method is important. The earlier a heuristic function is applied, the more important it is in the process of version selection. The ordering is a control heuristics, and as such a meta-heuristics.

Next subtask is to select a set of exported components. Finally, set of revisions, i.e. a set of components which form the bound configuration is selected. Here, for each variant from a generic configuration, a suitable revision must be selected according to requirements for revision selection. Method for selecting the most suitable revision is in fact similar to the above one for selecting most suitable variants.

Any configuration eligible for selection must satisfy two important properties: consistency and completeness. Both the properties can be properly defined for configurations [6]. Our method builds a configuration for which the properties hold, if such a configuration exists at all.

3 Experiments

The proposed method has been implemented in order to perform experiments aiming to analyze its properties. We have developed an ex-

perimental implementation of the method in Prolog. The implementation endeavour became an interesting research theme by itself. While devising algorithm implementing our method, we were facing essentially the problem of searching an 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. A more detailed description of these results is reported in [2].

4 Conclusion

One principle behind our approach to software configuration management is to allow software engineers to write down informations which are effectively interpretable by a supporting tool. We have identified a possible portfolio of such informations. In our model, software system is represented as an A/O graph with two kinds of versions, viz. variants and revisions. In the model, only variants, along with component families are represented. A variety of architectural relations can be defined between variants and families.

Our approach makes not only process of configuration building easier, it provides for a high degree of reuse. One can reuse a software system model, built generic configuration and configuration requirements. The fact that architectural relations can be defined between variants and families allows our model to be "more generic" as for example [1, 5]. One model created for presented method usually has to be described by more than one model in above citied approaches, because in them architectural relations can be defined only between component families.

Besides defining relations, the software engineer can specify conditions which must be satisfied by components to be included in the configuration. These conditions are effectively restrictions influencing consistency.

Both relations and conditions can be defined on various levels. The software engineer can define e.g. the architectural relation "*imports_from*" between two component families,

or between a variant and a family.

As a special kind of heuristic information, our method opens for the software engineer a room for incorporating specific knowledge on how to select the suitable component version. Our experiments show that with such heuristics, the method performs better.

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. A Prolog technique of implementing dependency-directed backtracking. Technical report, Slovak Technical University, Bratislava, 1995.

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

[4] M.L. Jaccheri and R. Conradi. Techniques for process model evolution in EPOS. *IEEE Transactions on Software Engineering*, 19(12):1145–1156, 1993.

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

[6] D.E. Perry. Dimensions of consistency in source versions and system compositions. In J. Feiler, editor, *Proc. of the 3rd Int. Workshop on Software Configuration Management*, pages 29–32, 1991.

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

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

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