AN APPROACH TO AUTOMATED BUILDING OF SOFTWARE SYSTEM CONFIGURATIONS

PAVOL NÁVRAT and MÁRIA BIELIKOVÁ

Slovak Technical University, Dept. of Computer Science 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, we concentrate on a method for building a software configuration. We build configurations by defining the system's model first. We have introduced a model of a software system as an AND/OR-type graph with two kinds of nodes: families and variants. Models serve as useful abstractions simplifying the process of configuration building. Being essentially a graph search, it is inevitable to have a method for selecting a proper version. Our approach offers to a software engineer a framework for specifying various heuristics describing which versions are to be preferred. In our method, the search is further reduced by introducing the concepts of generic and bound configurations and dividing the the fundamental steps of our method.

Keywords: Software configuration management, version control, selection controlled by heuristics.

1 Introduction

One of the important problems in software development and management 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 is built as a system consisting of many components which are naturally simpler than the whole, reducing the complexity in that way. Also the process of development is gradual, involving many more elementary, and therefore simpler steps. In those steps, very often only one particular component is transformed into a modified one, implementing nevertheless still essentially the same concept. Such transformations of components give rise to their versions. Consequently, we have a huge space of possible software system configurations viewed as sets of components satisfying requirements. The requirements can vary considerably, e.g. we can require a configuration intended as a product to be delivered to a customer, or a configuration intended as a document to resume development with.

The process of building a software system configuration is itself a complex one. Bookkeeping of attributes and relations of thousands of objects alone, not to speak of the frequency of their changes is a task which can best be handled by a computer. A support from a computer should further be sought in freeing the software engineer from the burden of a too detailed configuration specification. Instead, software engineer should have means to write higher level requirements which specify the configuration implicitly. Ultimately, this leads to employing relevant knowledge which would be represented explicitly and used by the computer. This can be considered an approach to automating the above mentioned part of the software engineering.

Any progress in automating the software engineering is hard to imagine without further formalisation in describing the objects and processes, and indeed there is much research endeavour oriented towards this aim, cf. e.g. [9]. More specifically, there have been various efforts to automate support to building software configuration. However, some of them do not take into account the existence of versions, e.g. [7, 27], or if they do, they neglect the difference between variants and revisions, e.g. [24, 18]. Some, on the other hand, make use of the difference between variants and revisions to increase efficiency of the building process, especially in version selection, e.g. [11, 13, 5, 12, 19, 26, 23]. An interesting view on the space of versions with revisions and variants evolving along orthogonal dimensions is presented in [21, 29].

An important aspect is reusability of the created configuration, as well as of its description and of the model of the software system, as has been pointed at by e.g., [1] in developing a system called ADELE, cf. [6]. The work influenced also our approach to the problem.

The outline of the rest of the paper is as follows. We present our approach to modelling software systems in section 2. We stress our interpretation of the notion of variant and give a formal description of the model. We briefly comment on differences between ours and related works. In section 3, we describe the way we use for specifying a configuration. Again, we briefly comment on the points which are new in our approach comparing to related works. The proposed method for building a configuration is presented in section 4. Next, we report on experimental evaluation of our method, including description of the tests performed, in section 5. Section 6 summarizes the points in our approach that are new when compared to related works. It makes also conclusions for the future work.

2 Modelling a Software System

Solving various problems related to building software system configurations in processes of software development and maintenance requires describing the actual software system in the simplest possible way, but still sufficiently rich to reflect the principal relations and properties which are decisive in the building process.

Generally, various kinds of graphs are being used to model software systems. A quite natural way to represent software systems is by means of a tree [22]. It does not, however, allow to describe the more complex aspects of such systems. An acyclic oriented graph is the next more suitable choice [27, 8]. Tichy [24] and later Estublier [6] have presented a model based on AND/OR graphs. An important aspect stressed also by the latter work is reusability of the created configuration.

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. We find the AND/OR graphs suitable in supporting the process of software system configuration building [3].

When attempting to identify basic parts of the system model, 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 concretization. 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 software component version. Creating a software 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, or as modifications caused by error correction, functionality enhancement, and/or adaptation to changes in environment. Such 'serial' versions, or revisions, are frequently result of concretizations of the same variant. A family of software components comprises all components which are versions of one another.

When defining a model of a software system, relations between software components should be considered. They 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_variant*, *has_revision*, which we shall commonly refer to as development-induced relations.

Let us formulate the notions more formally now.
Let $COMPONENT_S$ be a set of components of a software system S. Then 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. Relation $is\_version_S$ is reflexive, symmetric and transitive.

A set of all equivalence classes induced by the $is\_versions_S$ relation 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. We define which kinds of properties a component has. 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 (it represents relations expressing the system's architecture), $FunAttr$ is a set of functional attributes (the name together with the value), $CompAttr$ is a set of other attributes of that component (i.e. the attributes which are considered to revisions), $Constr$ is an expression for the constraint (in the sense of combining components to configurations), and *Realis* is the actual text of the component.

For example, let us consider a software component $c1$, for which there exists an architectural relation *contains* with a family of software components INIT, and a relation *has_document* with a family DOCUM:

$$\begin{array}{ll} c1 = (\{(contains, INIT), (has\_document, DOCUM)\}, & \text{\%architectural relations} \\ \quad \{(phase, implementation), (operating\_system, DOS), & \text{\%functional attributes} \\ \quad (prog\_language, C), (user\_interface, graphic), & \\ \quad (type\_of\_problem, diagnose)\}, & \\ \quad \{(author, peter), (date, 95\_01\_15), (status, tested)\}, & \text{\%other attributes} \\ \quad (parameters = ordered) \Rightarrow (system\_ver = DOS\_6\_2), & \text{\%constraint} \\ \quad \#define... \quad) & \text{\%program in C language} \end{array}$$

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.
The binary relation $is\_variant_S \subseteq COMPONENT_S \times COMPONENT_S$ is defined by:

$$x \; is\_variant_S \; y \Leftrightarrow x \; is\_version_S \; y \wedge x.ArchRel = y.ArchRel \wedge \\ x.FunAttr = y.FunAttr \wedge x.Constr = y.Constr$$

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.
As an example, let us present a part of software system which includes versions of (some of) its components. The example is taken from the software system KEX [17]. The system elements are shown in Figure 1 along with architectural relations between them. In Figure 1, the elements are organized 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 a variant and a family of software components. They are identified by names: EVAL, INIT, ACTIONS, INFER, DOCUM, INTERR, MANAGER. Each family includes several software components, e.g. the family EVAL includes three sets of components (i.e. variants) which include in turn nine revisions. Thus, the family EVAL includes nine software components.

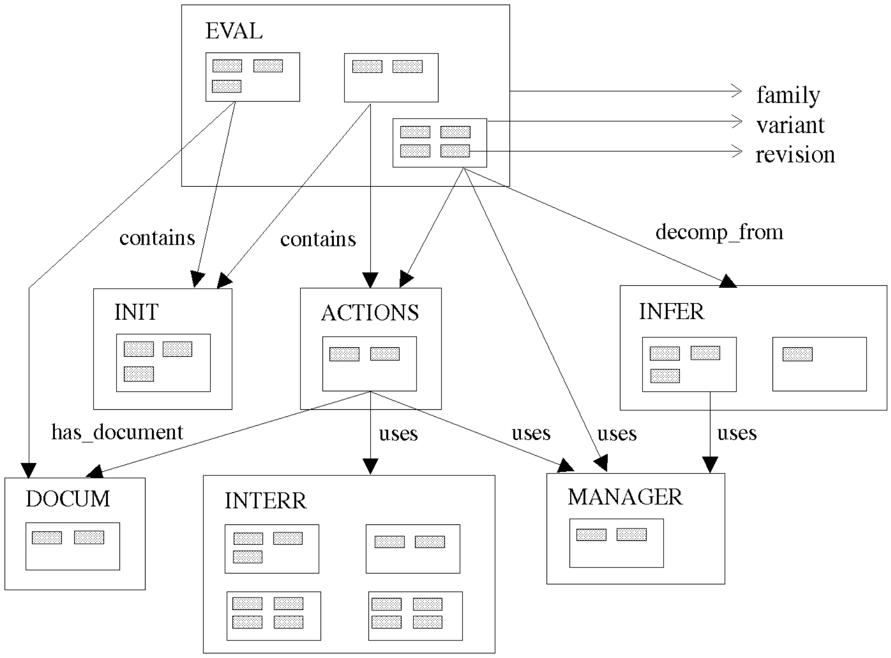

Figure 1: An example software system: partial hierarchy of elements.

The concepts introduced above will allow us to formulate a model of a software system which would support a process of configuration building. Our approach is based on an assumption that families of software components, variants and revisions are the basic entities involved in version management. All the relations between these entities can be grouped into architectural relations and development-induced relations. Development-induced relations determine membership of a variant in a family of components, and membership of a revision in a variant. Architectural relations express the system's architecture, especially the functionality of the components and the structure of the system. They must be defined explicitly at the level of variants and must be the same for all revisions included in a given variant. Particularly this assumption is very important, because it allows us to simplify the situation and to formulate a model of a software system which comprises only two kinds of elements: families and variants.

From the point of view of a family, a model should represent families and variants included in them. Links from a family to all ist variants are defined by relation $has\_variant_S \subseteq FAMILY_S \times VARIANT_S$: $x \; has\_variant_S \; y \Leftrightarrow y \subseteq x$. From the point of view of 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, 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. Taking into account that a software component is determined completetely only after a revision has been selected, a resulting configuration is built by selecting precisely one revision for each selected 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 and variants in such a way that these two kinds of nodes alternate on every path.

Any element of $E, (e_1, e_2) \in E$, called an edge, is of one from among the two mutually exclusive kinds. Either $e_1 \in VARIANT_S$ (a set of variants of a software system S), $e_2 \in F_S$ (a set of family names of a software system S). In this case, the node e_1 (variant) is called the A-node. Or $e_1 \in F_S, e_2 \in VARIANT_S$. In this case, the node e_1 (family) is called the O-node. Such graphs are denoted as A/O graphs. Revisions are covered in the model through A-nodes which represents variants, i.e. sets of revisions.

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 exactly one must be considered provided the O-node is under consideration (logical OR).

The example software system depicted in Figure 1 can be expressed by an A/O graph in Figure 2. For simplicity variants are given names which are derived from the name of the corresponding family of software components by suffixing it with a natural number.

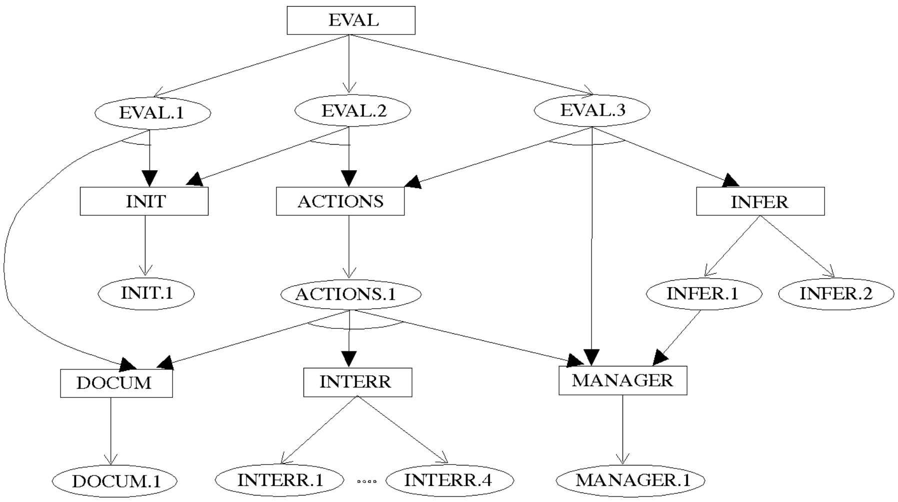

Figure 2: Model of the software system from Figure 1 represented as an A/O graph.

Let $FAMILY_S$ be a set of families of software components, $VARIANT_S$ be a set of variants of a software system S. Let F be a set of names and $f\_name_S : FAMILY_S \rightarrow F$ an injective function which assigns a unique name to each family of a software system S. Let $A \subseteq VARIANT_S \times F$ be a binary relation defined as

$e_1 \; A \; e_2 \Leftrightarrow \exists x \exists r(x \in e_1 \wedge r \in x.ArchRel \wedge r.FamilyId = e_2)$

Let $O \subseteq F \times VARIANT_S$ be a binary relation defined as:

$e_1 \; O \; e_2 \Leftrightarrow e_2 \subseteq f\_name_S^{-1}(e_1)$.

We define a model of a software system S to be an oriented graph $M_S = (N, E)$, where $N = F_S \cup VARIANT_S$ is a set of nodes with $F_S = \{x \mid x \in F \wedge f\_name_S^{-1}(x) \in FAMILY_S\}$, and $E = A \cup O$ is a set of edges such that every maximal connected subgraph has at least one root.

We remark that binary relation A represents architectural relations and relation O mirrors *has_variant* relation.

The requirement that a model of a software system should have at least one root is motivated by the fact that the model should serve the purpose of building of a software system configuration. When there is no root in a model, it is not possible to determine which components are to be selected for a configuration. Actually, the requirement is not a restriction in case of software systems. This follows from the very nature of the development of a software system and its description by transformations of solution states. Let us mention that the known approaches to modelling a software system by a graph all assume there is at least one root, cf. [15, 20, 14, 6].

3 Specifying a configuration

Taking into account the fact that nodes in our model are component families and variants, but not revisions (i.e., the actual software 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 [5]. To build a concrete (bound) configuration from a generic one, one revision for each variant in the generic configuration must be selected.

Let $M = (N, E)$ be a model of a software system. We take a subgraph $G_M = (U, H)$ of M, where $U \subseteq N, H \subseteq E$, such that each O-node (i.e. family) has exactly one successor in it, U includes at least one O-node (i.e. family) and at least one A-node (i.e. variant) and call it *a generic configuration* for M.

We take a set of software components such that for each variant included in the generic configuration G_M there is at most one revision (i.e., a software component) in it and call it *a bound configuration* B_{G_M}.

Requirements specifying a configuration determine which components are admissible for the configuration. Requirements specification is an important phase of the process of configuration building. The quality of the configuration largely depends on the requirements and on how they are actually used in the building process. Our approach to building the configuration is based on a model of a software system represented as an A/O graph. The requirements influence the subgraph derived from it, i.e. the generic

configuration, as well as selection of a revision for each variant, i.e. consequently forming the bound configuration.
The configuration requirements can be classified as:

- requirements related to properties of components viewed from the point of view of the whole system, i.e. which components (families of them) are to be considered when building the configuration,
- requirements on version selection.

The first group is specified by:

- a set of names of architectural relations (represented by edges originating in A-nodes), which will serve to integrate A-nodes of the system,
- a condition for selection of exported components (represented for instance by a logic expression referring to component attributes).

The requirement for version selection is expressed as a sequence of heuristic functions which reduce the set of suitable versions. The heuristic functions represent knowledge about the degree of suitability of the respective versions. We adopted a widespread approach to identify versions of components by the use of attributes [13, 5, 28]. The heuristic functions refer to properties of versions as defined by their attributes. 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:

- *a 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.
- *a 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 $h_1, h_2, \ldots, h_n$.

In order to build a configuration which would meet the requirements, our method builds a generic configuration first and then proceeds to building a bound one. We distributed configuration requirements to two parts: the generic configuration requirement and the bound configuration requirement.
We shall write the generic configuration requirement as a triple:

$gcr_M = (Rel, VariantCond, ConfConstr)$, where Rel is a set of names of architectural relations, $VariantCond$ is a set each element of which consists of three parts: family specification, necessary condition for variant selection and suitability condition for variant selection. $ConfConstr$ is an expression (built up from references to heuristic functions) specifying constraint for all components to be included in the configuration. A possible example of the generic configuration requirement is:

$$\begin{aligned}gcr_M = (&\{contains, uses, decomp\_from\},\\ &\{(*, operating\_system = DOS \wedge communication\_language = Slovak,\\ &\quad [progr\_language = Prolog,\\ &\quad \textit{prefer version with a greater number of definite attributes,}\\ &\quad \textit{prefer version with a smaller number of defined architectural}\\ &\quad \textit{relations to other components }])\},\\ &not(progr\_language(x) = Prolog \wedge progr\_language(y) = Pascal)).\end{aligned}$$

Let us note that in the example, requirements for variant selection are the same for all families in the software system (denoted as '*').

We shall write the bound configuration requirement as a pair:

$bcr_M = (ExpCond, RevisionCond)$,

where $ExpCond$ is an expression specifying a condition for selection of exported components, and $RevisionCond$ is a set each element of which consists of two parts: family specification and suitability condition for revision selection.

A possible example of the bound configuration requirement is:

$bcr_M = (true, [author = maria, date \leq 17\_6\_94, state = tested])$.

4 Method for building a configuration

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 approach is based on an assumption that all software components which can be possibly needed in the process of configuration building are available. However, having in mind the fact that the activities of configuration building and using [2, 6] can be separated, the above assumption is not a real restriction.

Our method of building a software system configuration makes use of some ideas from previous works, especially those of [11, 25, 18, 2, 6, 1]. 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.

Now we preview the main points of our method for 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 shall be considered in building the configuration. Selection of component families is based on edges originating in A-nodes and on component selection condition (exported components condition).
In selection of considered component families, there is selected a subgraph from the system model such that only edges representing relations specified in configuration requirement form it. The exported components selection is performed after completing the second task. Removing of nodes which do not satisfy the exported components condition can cause removing their successors which have to be considered.

Second subtask is 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 (i.e., a family) is its variant. A problem arises here in those cases when either there are more than one variant satisfying the requirements, or there is no such variant at all. The problem resembles kinds of 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 versions according to their suitability, but rather to delete step by step those least suitable from the set of all possible versions [16]. 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 suitability selection condition.

Finally, third subtask is to select a set of revisions, i.e. a set of components which form the bound configuration. 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.

An overall scheme of our method is in Figure 3. Detail specification of the method is put in the Appendix.

Figure 3: Method for building a configuration.

The above description of the method specifies in fact only *what* is to be achieved in the respective steps. *How* this can be done shall now be presented for each of the principal steps of our method.

4.1 Forming the subgraph

In the step 1.1, a subgraph of the system's model M is to be formed in such a way that all its A-edges shall represent only relations indicated in $gcr_M.Rel$. The task can be described as searching the graph M from its roots and selecting exclusively such edges and nodes which satisfy the condition stated in 1.1.

The algorithm uses two data structures which we denote as $CLOSED$ (listing all such nodes all the successors of which have been processed already) and $OPEN$ (listing all such nodes that only their predecessors, but not successors have been processed yet). Both $CLOSED$ and $OPEN$ list the nodes along with pointers to their respective predecessors which are useful in forming the resulting graph F. Therefore $CLOSED$ and $OPEN$ are both sets of elements which are pairs $(Node, Pred)$, where $Node \in N, Pred \subseteq N$.

Input: model $M = (N, E)$ of the software system,
a generic configuration requirement gcr_M for the model M.
Output: graph $F = (FN, FE)$

1. Push the roots of the graph M into the set $OPEN$. Initialize $CLOSED$ to an empty set:
   $OPEN \leftarrow \{(x, \{\}) \mid x \in N \wedge \neg(\exists y((y, x) \in E))\}$
   $CLOSED \leftarrow \{\}$

2. **if** $OPEN$ is empty **then halt**, with the sets FN and FE formed as follows:
   $FN = \{x \mid \exists u(u \in CLOSED \wedge u.Node = x)\}$
   $FE = \{(x, y) \mid \exists u(u \in CLOSED \wedge x \in u.Pred \wedge y = u.Node)\}$

3. $e \leftarrow$ the element selected from $OPEN$ and
   delete the value of e from $OPEN$

4. **if** $\exists p\ ((e.Node, p) \in CLOSED)$
   then $(e.Node, p) \leftarrow (e.Node, p \cup e.Pred)$
   (i.e. extend p with a pointer back to $e.Pred$) and
   go to 2.

5. **if** $e.Node$ is a A-node in graph M
   then for each successor e_succ of $e.Node$ do:
   if $\exists k \exists r(k \in e.Node \wedge r \in k.ArchRel \wedge\ r.RelationId \in gcr_M.Rel \wedge$
   $r.FamilyId = e\_succ)$
   then push $(e\_succ, \{e.Node\})$ into $OPEN$
   push e into $CLOSED$ and
   go to 2.

6. **if** $e.Node$ is an O-node in graph M
   then for each successor e_succ of $e.Node$ do:
   push $(e\_succ, \{e.Node\})$ into $OPEN$
   push e into $CLOSED$ and
   go to 2.

The result after applying the step 1.1 of the method to the model of the software system from the Figure 2 using a generic configuration requirement described in the section 3 is depicted on Figure 4.

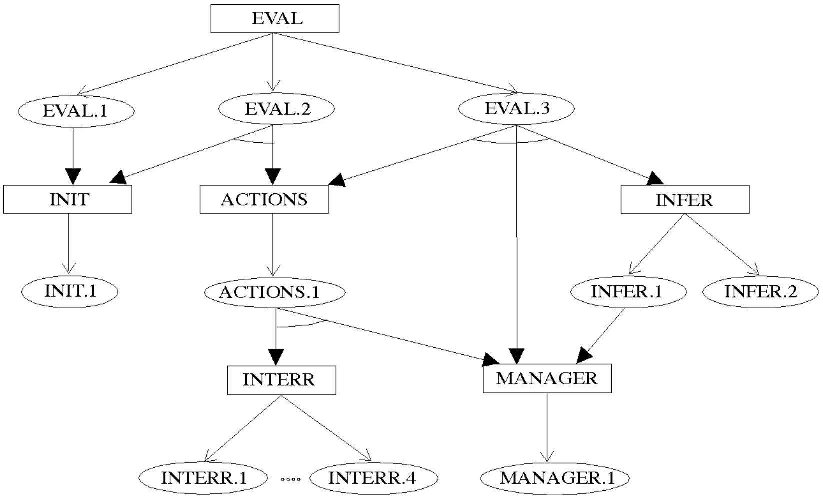

Figure 4: A/O graph after the step 1.1.

4.2 Forming the generic configuration

In the step 1.2 of the method, generic configuration G_M is to be formed. Input to this step is the graph F formed in the previous 1.1. Graph G_M includes all the roots of F. It includes just one successor of each O-node of F included in G_M. It includes all the successors of each A-node of F included in G_M. Moreover, the nodes in G_M must meet the constraints defined in the generic configuration requirement $gcr_M.ConfConstr$.

The task can be described as searching an A/O graph F starting from its roots and by selecting always nodes which satisfy the above conditions. However, the algorithm is not as simple as the similar one in 1.1. The reason is the additional condition that the graph, i.e. the set of nodes must be consistent with respect to given constraints. Further complication is due to the fact that the method of selecting the version used in finding the successor of an O-node can fail. As a consequence, there can occur the situation that the graph being formed is not consistent. It must be modified in that case. We have devised and implemented the method in a logic programming environment using techniques especially designed to cope with the challenge. More specifically, it is based on identifying the reason for a deadend. It is attempted to find a place in the graph where the search for an alternative solution should be resumed. The aim is to keep the number of visited nodes and number of performed consistency checks as small as possible. The technique of node marking is used as well [4].

The result after applying the step 1.2 of the method to the model of the software system from the Figure 2 is depicted on Figure 5. As was stated earlier, in the method for building a configuration step 1.2 exactly one successor to each O-node is selected by applying the method of version selection. Version selection is controlled by knowledge in form of heuristic functions which refer to properties of versions.

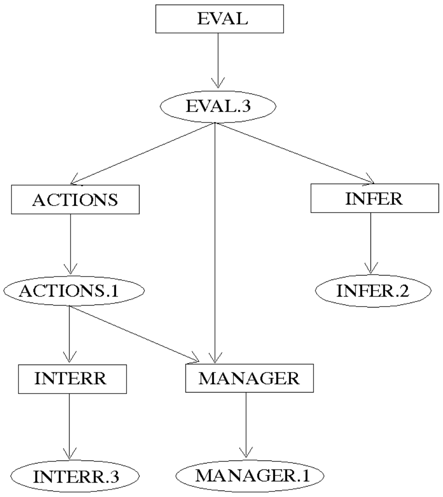

Figure 5: Generic configuration built from A/O graph in Figure 4.

4.3 Making provisions to include exported components

In the step 2.1 of the method, the condition of exported components selection $bcr_M.ExpCond$ is applied. The condition is represented by a function which assigns to each software component value from $\{satisfies, does\_not\_satisfy\}$. The condition of exported components selection refers only to properties which are the same for all components included in the given variant (architectural relations and functional attributes). Therefore, it suffices to apply it to any one component for each variant from the generic configuration G_M. The algorithm implementing the method is described in the following way:

Input: graph $G_M = (U, H)$
a bound configuration requirement bcr_M
Output: set VE

1. $VE \leftarrow \{\}$

2. for each element $v \in U$:
   if v is A-node
   then $k \leftarrow$ the element selected from v and
   if $bcr_M.ExpCond(k) = satisfies$
   then $VE \leftarrow VE \cup \{v\}$

Set of exported variants after applying the step 2.1 of the method to the generic configuration from the Figure 5 in the case the condition of exported components is set *true* is:

$VE = \{$ *EVAL.3, ACTIONS.1, INTERR.3, MANAGER.1, INFER.2* $\}$,
i.e. all the variants from the generic configuration.

4.4 Forming the bound configuration

In the last step of the method of building the configuration, the bound configuration is formed by selecting a revision for each variant included in the set resulting from the step 2.1. The selection takes place according to the revision selection condition $bcr_M.RevisionCond$. We apply our method of version selection accordingly.

5 Experimental evaluation

The above described method has been implemented in Prolog language. The primary purpose of the implementation was to create a prototype system supporting the process of building software system configurations which would be suitable for experimenting. 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 [10]. We have performed several kinds of experiments aiming at empirical evaluation of important properties of our method. For each kind, there were performed extensive tests, if fact thousands of them, to gather data allowing certain conclusions.

One kind of experiments concerns the version selection algorithm, which is part of our method. When analyzing this algorithm, we concentrated on elementary heuristic functions applications, which are the most expensive operations. The empirical analysis shows that for larger numbers of filters, number of applications of filters does not depend on it for a given number of versions. An explanation for it could be that if the filters are at least "reasonably", or "sufficiently" selective, then they succeed to arrive at one selected version "sufficiently" early before the later scheduled (i.e., those with higher index i) filters ever become applicable. No matter how many more filters we include, they would never be applied. On the other hand, number of applications of filters linearly depends on the number of versions, as could be expected.

Other kind of experiments concerns the algorithm for building the generic configuration. Essentially, it is a graph searching algorithm. It searches an A/O graph to find a solution which additionally satisfies given constraints. When comparing to the usual algorithm with chronological backtracking (CHB), our algorithm is based on two principal additional concepts. First, we have used a special version of the dependency-directed backtracking to resolve the deadend situation. It analyses the reasons of inconsistency at the deadend node and uses the results in deciding which is the most promising node to visit next. This algorithm is further enhanced by marking mechanism (RB) which allows recording and propagating results of the analysis of the current deadend node.

Our experiments were aimed at analysing the proposed algorithm how effective it is in coping with backtracks. We randomly generated problems, i.e. A/O graphs, and applied both algorithms to them. Number of backtracks necessary when solving the problem is compared to number of backtracks of the usual chronological backtracking algorithm. By definition, the ratio is always 1 for CHB algorithm. An alternative measure can be the number of consistency checks performed during searching a particular graph. In fact, we have experimented with both of these measures. We report on results using the number of bactracks in this paper. Due to the space limitations, we omit results using the latter. However, there has been a strong similarity between them.

In Figure 6 we can see how the number of backtracks (denoted as BACK) for the new proposed algorithm which allows recording and propagating results of the analysis of the current node (RB) depends on the number of backtracks for the basic algorithm with chronological backtracking (CHB).

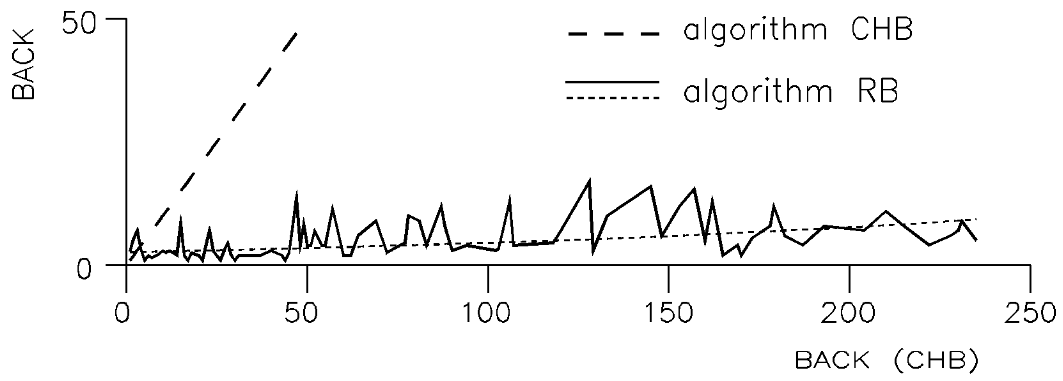

Figure 6: Number of backtracks for RB and CHB.

In absolute numbers, the results show that the number of backtracks by enhanced algorithm is less than the number of backtracks when performing the usual chronological backtracking for the same graph.

The presented graph is based on results of searching about 9000 generated graphs with number of nodes equal to 500. The reason for not including experiments with higher numbers of nodes was that they are obviously more time consuming and we could not have afforded them with our relatively modest computing environment. On the other hand, we have tried some experiments with other values as well. Based on them, we feel supported in the claim that our results are representative regardless this value.

6 Conclusions

The proposed techniques for configuration management have basis in the general principle behind our approach: to allow software engineers to express informations that can be interpreted by the tool that automates the support to software building. In the following, we summarize briefly the most important properties of the proposed method for building the software system configuration. The method integrates "good" properties of some of the known approaches and moreover, solves some of their weak points in an original way.

We consider families to be equivalence classes. Variants are sets of components, i.e. revisions. Moreover, the architectural relations are defined between variants and families. In such a way, we allow to define various properties of particular variants with respect to architectural relations to other components. This is motivated by the fact that indeed in practice we often have different architectural relations for alternative representations of some solutions when developing a software system.

Our approach to building a configuration is to allow for explicit specification of knowledge leading to constraining the model in such a way that only required components would be included in the configuration. This approach was used also in other systems, cf. [13, 6, 11]. What is different is the structure of knowledge (configuration requirements), its representation and interpretation.

Another important feature of our method is that relations between components, as well as their attributes and constraints can be defined at the level where they actually occur, similarly to e.g. [6]. This supports also the process of forming the system's model, defining the components' interfaces, and, last but not least, writing the configuration requirements.

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.

Our approach is similar to version selection in DSEE [11]. What is new is distribution the requirements for version selection into the necessary selection condition and the suitability selection condition and 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 necessary selection condition.

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 developent needs.

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 [5] allows consideration of a subset of defined architectural relations in a system model but no condition for selection of exported components. Other approaches simply consider all parts defined in a system model.

Our modelling a software system is limited by the fact that every change of functional properties in software component development results in a new variant regardless to the real nature of the change. Similar limitation has the system ADELE, cf. [6].

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.

Appendix

Specification of the 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 $gcr_M = (Rel, VariantCond, ConfConstr)$,
- a bound configuration requirement $bcr_M = (ExpCond, RevisionCond)$.

Output from the method is:

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

1. *Forming a generic configuration*

1.1. **Input:** software system model $M = (N, E)$, and
generic configuration requirement gcr_M

Output: a graph $F = (FN, FE)$, where $FN \subseteq N, FE \subseteq E$ and the following holds:

$$\begin{aligned}
&\forall i((i \geq 1 \wedge i \leq m) \Rightarrow s_i \in FN)\wedge\\
&\forall n_1((n_1 \in FN \wedge (n_1 \ is\ O\text{-}node))\Rightarrow\\
&\quad \forall n_2((n_2 \in N \wedge (n_1, n_2) \in E)\Rightarrow (n_2 \in FN \wedge (n_1, n_2) \in FE)))\wedge\\
&\forall n_1((n_1 \in FN \wedge (n_1 \ is\ A\text{-}node))\Rightarrow\\
&\quad \forall n_2((n_2 \in N \wedge (n_1, n_2) \in E\wedge\\
&\qquad \exists k \exists r(k \in n_1 \wedge r \in k.ArchRel\wedge\ r.RelationId \in gcr_M.Rel\wedge\\
&\qquad\quad r.FamilyId = n_2))\Rightarrow (n_2 \in FN \wedge (n_1, n_2) \in FE)))\wedge\\
&\forall n(n \in FN \Rightarrow \exists i(s_i\ FE^+ n).
\end{aligned}$$

1.2. **Input:** a graph $F = (FN, FE)$ formed in step 1.1, with roots $s_{F1}, s_{F2}, \ldots, s_{Fm}$

generic configuration requirement gcr_M

Output: generic configuration $G_M = (U, H)$, where
$U \subseteq FN, H \subseteq FE$ and the following holds:

$$\begin{aligned}
&\forall i((i \geq 1 \wedge i \leq m) \Rightarrow s_{Fi} \in U)\wedge\\
&\forall n_1((n_1 \in U \wedge (n_1 \ is\ O\text{-}node))\Rightarrow \exists! n_2((n_2 \in U \wedge (n_1, n_2) \in H))\wedge\\
&\forall n_1((n_1 \in U \wedge (n_1 \ is\ A\text{-}node))\Rightarrow\\
&\qquad \forall n_2((n_2 \in FN \wedge (n_1, n_2) \in FE)\Rightarrow (n_2 \in U \wedge (n_1, n_2) \in H)))\wedge\\
&\forall n(n \in U \Rightarrow \exists i(s_{Fi}\ H^+\ n)\wedge\\
&gcr_M.ConfConstr(U) = U\wedge\\
&\forall n((n \in U \wedge (n \ is\ A\text{-}node) \wedge \exists k(k \in n))\Rightarrow k.Constr(U) = U),
\end{aligned}$$

Exactly one successor to each O-node (i.e., family) included in the graph G_M is selected by applying our method of version selection. The method of version selection for O-node n is applied with following inputs:

- set of versions $MU = \{x | x \in FN \wedge (n, x) \in FE\}$,
  i.e. variants included in the family represented by the node n
- version selection requirement,
  i.e. necessary condition for variant selection, suitability condition for variant selection taken from $gcr_M.VariantCond$.

Output is the selected element from the set MU, i.e. a successor of the node n.

2. *Forming a bound configuration*

2.1. **Input:** generic configuration $G_M = (U, H)$ formed in step 1.2,
bound configuration requirement bcr_M

Output: set of exported variants VE such that the following holds:

$$VE = \{v | v \in U \wedge (v \ is \ A\text{-}node) \wedge \\ \forall k (k \in v \Rightarrow bcr_M.ExpCond(k) = satisfies\}$$

2.2. **Input:** set of exported variants VE formed in step 2.1,
bound configuration requirement bcr_M

Output: bound configuration, i.e. a set of software components V such that the following holds:

$$V = \{k | \forall v (v \in VE \Rightarrow \exists! k (k \in v)\},$$

where for each variant included in the set VE exactly one component has been selected by applying the method of version selection. In this case, the method of version selection is applied for the variant v with the following inputs:

- set of versions in consideration $MU = \{k | k \in v\}$, i.e. revisions included in the variant represented by the node v
- version selection requirement, i.e. suitability condition for revision selection taken from $bcr_M.RevisionCond$.

Output is the selected element from the set MU, i.e. the software component (revision) included in the variant v.

References

1. N. Belkhatir and M. Ahmed-Nacer, Major issues on PSEE: Process software engineering environments, *Computers and Artificial Intelligence*, **3**, (1993), pp.279–298.
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*, 1991, pp.43–52.
3. M. Bieliková and P. Návrat, Modelling software systems in configuration management, *Applied Mathematics and Computer Science*, **4** (1995), pp.751–764.
4. M. Bieliková and P. Návrat, A Prolog technique of implementing search of A/O graphs with constraints, Tech. report, Slovak Technical University, Bratislava, 1995.
5. J. Estublier, Configuration management: the notion and the tools, In *Proc. Int. Workshop on Software Version and Config. Control*, Stuttgart, 1988, pp.38–61.
6. J. Estublier, The Adele configuration manager, Tech. report, L.G.I., Grenoble, 1992.
7. S.I. Feldman, MAKE - a program for maintaining computer programs, *Software - Practice and Experience*, **3** (1979), pp.255–265.
8. D. Heimbigner, A graph transform model for configuration management environments, In P. Hederson, editor, *Proc. ACM SIGSOFT'88*, Boston, 1988, ACM Press, pp.216–225.
9. M. Holcombe, An integrated methodology for the specification, verification and testing of systems, *Software Testing, Verification and Reliability*, **4** (1993), pp.149–163.
10. A. Jazaa, Toward better software automation, *Software Engineering Notes*, **1** (1995), pp.79–84.
11. D.B. Leblang and R.P. Chase, Parallel software configuration management in a network environment. *IEEE Software*, **6** (1987), pp.28–35.
12. Luqi, A graph model for software evolution, *Trans. on Software Engineering*, **8** (1990), pp.917–927.
13. A. Mahler and A. Lampen, An integrated toolset for engineering software configurations, In P. Hederson, editor, *Proc. ACM SIGSOFT'88*, Boston, 1988, ACM Press, pp.191–200.
14. D.B. Miller and R.G. Stockton, An inverted approach to configuration management, ACM Sigsoft, 1989, pp.1–4.
15. K. Narayanaswamy and W. Scacchi, Maintaining configurations of evolving softwa-

re systems, *IEEE Trans. on Software Engineering*, SE-13(3), 1987, pp.325–334.

16. P. Návrat and M. Bieliková, Knowledge controlled version selection in software configuration management, *Software—Concepts and Tools*, **17** (1996), pp.40–48.
17. P. Návrat, P. Frič, M. Adámy, and I. Mladá, KEX: Computer aided knowledge engineering system, In *Proc. Computers'89 Conf.*, Blahová, 1989, pp.156–162.
18. 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 Conf. ESEC'89*, Springer-Verlag, 1989, pp.222-242.
19. J. Plaice and W.W. Wadge, A new approach to version control, *IEEE Trans. on Software Engineering*, **3** (1993), pp.268–275.
20. E. Ploedereder and A. Fergany, The data model of the configuration management assistant, ACM Sigsoft, 1989, pp.5–14.
21. C. Reichenberger, Concepts and techniques for software version control, *Software - Concepts and Tools*, **3** (1994), pp.97–104.
22. M.J. Rochkind, The source code control system, *IEEE Trans. on Software Engineering*, SE-1(4), 1975, pp.364–370.
23. I. Sommerville and G. Dean, PCL: a language for modelling evolving system architectures, *Software Engineering Journal*, March, 19996, pp.111–121.
24. 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*, North Holland, 1986, pp.219–236.
25. W.F. Tichy, Tools for software configuration management, In *Proc. Int. Workshop on Software Version and Configuration Control*, Stuttgart, 1988, pp.1–20.
26. M. Wein, S.A. MacKay, C.A. Gauthier, and W.M. Gentleman, Evolution is essential for software tool development, In *Proc. of the 1995 Int. Workshop on Computer Aided Software Engineering (CASE-95)*, Toronto, Ontario, 1995.
27. S.S. Yau and J.J. Tsai, Knowledge representation of software component interconnection information for large-scale software modifications, *IEEE Transactions on Software Engineering*, SE-13(3), 1987, pp.355–361.
28. A. Zeller, A unified configuration management model, In G. Kaiser, editor, *Proc. 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM Press, 1995, pp.151–160.
29. A. Zeller and G. Snelting, Handling versions sets through feature logic, In W. Schäfer and P. Botella, editors, *Proc. 5th European Software Engineering Conference*, Springer-Verlag, 1995, pp.191–204.