Modelling Software Systems in Configuration Management

Mária Bieliková and Pavol Návrat *

Abstract. In the paper, a model of a software system is presented which is sufficiently rich to represent the architectural and development-induced relations within the system which are decisive in building the configuration, yet simple enough to allow efficient processing. We distinguish variants and revisions as two kinds of components. Systems are represented by two kinds of nodes in an AND/OR type graph which models the system. The model forms a basis for a method to build a configuration.

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 (Tichy, 1988; Estublier, 1988; Reichenberger, 1994) two different kinds of modifications are reflected in two kinds of versions: revisions and variants.

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 (Mahler, Lampen, 1988). New variant represents an alternative solution. Variants usually exist concurrently. Variants result from experimental development or modifications towards amelioring 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.

A software system consists of many interrelated software components. A model is used to express its structure, respecting in our case the point of view of building the software system configuration. A frequent and quite natural way of representation is by means of a tree, e.g. (Rochkind, 1975; Tichy, 1985) or more generally by means of an oriented graph (usually constrained to be an acyclic one), e.g. (Yau, Tsai, 1987; Heimbigner, 1988; Reichenberger, 1989; Oquendo *et al.*, 1989; Luqi, 1990; Mahler, Lampen, 1990; Vescoukis, Psaromigos, Skordalakis, 1992; Plaice, Wadge, 1993). Sometimes, the graphs have a special structure. In (Holt, Mancoridis, 1994) the structure of a software system is expressed by so called tube graphs. They support control of changes of relations between components during the system development with respect to given constraints to its architecture.

A general approach in models based on graphs is to represent components by nodes and relations between them by edges. They differ mainly in ways of expressing the relations between components and in structuring the components.

* Slovak Technical University, Dept. of Computer Science and Engineering, Ilkovičova 3, 812 19 Bratislava, Slovakia, fax: (++ 42 7) 720 415, e-mail: {bielikova,navrat}@elf.stuba.sk

Relations between components are very important from the point of view of configuration management. In fact, if the components were independent, most of the problems with software configuration management during software development and maintenance would simply dissapear. Among the important relations influencing configuration building, we list e.g. *depends_on*, *contains*, *refers_to*, *uses*, *is_variant*, *is_revision* etc.

A model of a software system expresses first of all its structure. When referring to parts of the system, it uses names. Shape of the model is influenced by the intended application. In software configuration management, software system is frequently modelled by a graph in various variations. Besides an acyclic graph and a tree, some authors use also an AND/OR graph, advantages of which become apparent especially in connection with supporting the process of building a sofware system configuration.

In (Tichy, 1986), a model is presented which is essentially an AND/OR graph, and its advantages are stressed in comparison to a more "classical" acyclic graph. In this model, AND nodes represent configurations, OR nodes represent families of components (i.e., groups of versions) and leaves represent atomic components. AND and OR nodes can be freely intermixed in the graph. Configurations represent in fact composite components, because AND nodes take care for integration during configuration building. In the model, AND/OR graph has been generalised to an attributed graph, i.e. each node can have associated a set of attributes.

A model based on AND/OR graphs has been presented also in (Estublier, 1992). A component of a software system is represented by a three level scheme: family of components - interface - realisation. It is allowed to form versions of both the interface and the realisation of a component. Taking into account that the interface of a component changes significantly less frequently than its realisation, we can make use of this concept in software system development. On the other hand, the model becomes more complicated and less transparent. In the AND/OR graph, AND nodes and leaves represent realisations and OR nodes represent interface of software components.

In this paper, our goal is to devise a way of modelling software systems that would suit the process of building a software system configuration. We have also developed a method to support software engineer in actually building the configuration. The method uses the model presented here.

2. An outline of our approach

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

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_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 concretization 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_variant, has_revision*, which we shall commonly refer to as development-induced relations.

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_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.

3. Families of software components

Let us attempt to identify the class of those transformations of states in the software development process which result in forming a new version of some component. According to our analysis, two features are crucial: language (specification formalism) of the component being transformed, and '1-to-1' property, i.e. that one component is transformed into just one component. We assume that all the specification formalisms used in the software development process are categorised into groups. Each group represents certain type of formalism, e.g. structure diagram, programming language etc. The assumption allows us to consider as versions two components which are written in two different, but closely related languages. For example, it is often the case that we have two implementations of some algorithm, one in language L_1 (say, C), and the other in language L_2 (e.g., Pascal). Naturally, we wish to consider them as versions.

Therefore, we distinguish a special class of transformations. We call a transformation to be t_version, if it preserves both the specification formalism and the '1-to-1' property.

A transformation is applied to a state, which consists of software components. Generally, it may involve several components, and it may result in a different number of components. For example, any refinement step which refines a component (according to divide and impera principle) into two components, transforms one component into two components. Of course, all the remaining components are assumed to stay unchanged during this particular transformation, as they are not involved. Such transformation can never create a version.

A transformation which changes radically the language will not result in a version. For example, if a process specified by a data flow diagram is implemented by a module written in C, the module will not be considered a version of the DFD specification.

Only transformations preserving both the described properties lead to versions. In such a way, all the versions are formed solely by t_version transformations. Transformations of any other kind lead to forming new sets of components. Sets of software components which have been formed by applying t_version transformations within them are understood to be in *is_version* relation. Such understanding of the notion version conforms to most of the related works in SCM area. The sets formed in the described way are called families of software components.

Summing up, family of software components is such a set of software components, that the components included have been formed gradually only by t_version transformations.These transformations define on the set of software components of a software system S a binary relation. Let us call the relation $t\_version_S$. It is asymmetric and irreflexive.

We get the binary relation $is\_version_S$, if we take the reflexive and transitive closure of the relation $(t\_version_S \cup t\_version_S^{-1})$. This reflects the above informal description of the relation $is\_version_S$.

Definition. Let $COMPONENT_S$ be a set of components of a software system S. Let a binary relation $t\_version_S \subseteq COMPONENT_S \times COMPONENT_S$ be given as:

$x\ t\_version_S\ y \Leftrightarrow y$ is formed by applying transformation t_version to x in the software system S.

Then a binary relation $is\_version_S \subseteq COMPONENT_S \times COMPONENT_S$ is given as:

$is\_version_S = (t\_version_S \cup t\_version_S^{-1})^*$.

In order to simplify the notation, the index S will be omitted whenever it is sufficiently clear from the context which software system is being referred to. Thus, we write occasionally $COMPONENT$ instead of $COMPONENT_S$, $is\_version$ instead of $is\_version_S$, etc.

Conjecture. The relation $is\_version$ is an equivalence.

Demonstration. The relation is defined as the reflexive and transitive closure, and therefore it is trivially reflexive and transitive. It is also symmetric, because the relation $(t\_version \cup t\_version^{-1})$ is symmetric.

The relation *is_version* defines classes of equivalence on the set of components of a software system. The classes represent families of software components.

Definition. Let $COMPONENT_S$ be a set of components of the software system S, and $is\_version_S$ be a binary relation $is\_version_S \subseteq COMPONENT_S \times COMPONENT_S$. A set of all equivalence classes induced by the $is\_version_S$ relation is denoted $FAMILY_S$:

$FAMILY_S = \{ F \mid F = \{ x \mid x \in COMPONENT_S \wedge \ x\ is\_version\ y\ for\ some\ y \in COMPONENT_S \} \}$

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.

4. Variants

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.

Definition. Let $COMPONENT_S$ be a set of software components of a software system S, and $FAMILY_S$ a set of families of S. Let $F, NAME\_ATTR, NAME\_REL$ be mutually exclusive sets of names. Let DOM be a set of domains of admissible values of attributes such that for each name $a \in NAME\_ATTR$ there is only one domain $D_a \in DOM$ of admissible values of an attribute named a. Let $f\_name_S : FAMILY_S \rightarrow F$ be an injective function which assigns a unique name to each family of a software system S. Further let e be a well formed constraint expression from a suitable language, and r be an expression from a specification language. We call a software component a quintuple c_S:

$c_S = \{ArchRel : REL, FunAttr : ATTR1, CompAttr : ATTR2, Constr : e, Realis : r\}$,

where

REL is a set of pairs $(RelationId : n, FamilyId : t)$,

$n \in NAME\_REL, t \in \{ x \mid x \in F \wedge f\_name_S^{-1}(x) \ \in \ FAMILY_S\}, and\ \neg(c_S \in f\_name_S^{-1}(t))$,

$ATTR1$ and $ATTR2$ are finite sets of pairs $(a, d), a \in NAME\_ATTR, d \in D_a$.

In the definition, we have given names to elements of tuples to be able to refer to them throughout the text. For example, if e is a software component, i.e. a quintuple, we may refer to its first element as $e.ArchRel$, to its second element as $e.FunAttr$, etc.

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:

$c1 = (\{(contains, INIT), (has\_document, DOCUM)\},$ $\quad\%architectural\ relations$
$\{(phase, implementation), (operating\_system, UNIX),$ $\quad\%functional\ attributes$
$(prog\_language, C), (algorithm, simple), (type\_of\_problem, diagnose)\},$
$\{(author, peter), (date, 95\_01\_15), (status, working)\},$ $\quad\%other\ attributes$
$(parameters = ordered) \Rightarrow (system\_ver = UNIX\_4\_3),$ $\quad\%constraint$
$\#define...$ $\quad\%program\ in\ C\ language$
$)$

We included in the structure of a software component besides other elements also a constraint expression and a specification expression. We shall make use of them in the process of building a configuration.

In order to describe variants, we define a binary relation *is_variant* which determines a set of software components with equally defined architectural relations, functional attributes and constraints within a given family.

Definition. Let $COMPONENT_S$ be a set of software components of a software system S with a binary relation $is\_version_S$ defined. We define a binary relation $is\_variant_S \subseteq COMPONENT_S \times COMPONENT_S$:
$x\ is\_variant_S\ y \Leftrightarrow x\ is\_version_S\ y \wedge$
$x.ArchRel = y.ArchRel \wedge x.FunAttr = y.FunAttr \wedge x.Constr = y.Constr$

Conjecture. Relation $is\_variant$ is an equivalence.

Demonstration. We use properties of binary relations $is\_version$ and '=' which are both equivalences, i.e. they are reflexive, symmetric and transitive. We show that these properties hold for $is\_variant$ as well.

Relation $is\_variant$ is reflexive because the following holds:
$\forall x(x\ is\_version\ x \wedge$
$x.ArchRel = x.ArchRel \wedge x.FunAttr = x.FunAttr \wedge x.Constr = x.Constr) \Rightarrow$
$\forall x(x\ is\_variant\ x)$.

Relation $is\_variant$ is symmetric because the following holds:
$\forall x, y(x\ is\_variant\ y \Leftrightarrow$
$(x\ is\_version\ y \wedge$
$x.ArchRel = y.ArchRel \wedge x.FunAttr = y.FunAttr \wedge x.Constr = y.Constr) \Leftrightarrow$
$(y\ is\_version\ x \wedge$
$y.ArchRel = x.ArchRel \wedge y.FunAttr = x.FunAttr \wedge y.Constr = x.Constr) \Leftrightarrow$
$y\ is\_variant\ x)$.

Relation $is\_variant$ is transitive because the following holds:
$\forall x, y, z(x\ is\_variant\ y \wedge$
$y\ is\_variant\ z \Leftrightarrow$
$(x\ is\_version\ y \wedge$
$x.ArchRel = y.ArchRel \wedge x.FunAttr = y.FunAttr \wedge x.Constr = y.Constr \wedge$
$y\ is\_version\ z \wedge$
$y.ArchRel = z.ArchRel \wedge y.FunAttr = z.FunAttr \wedge y.Constr = z.Constr) \Leftrightarrow$
$(x\ is\_version\ y \wedge y\ is\_version\ z \wedge$
$x.ArchRel = y.ArchRel \wedge y.ArchRel = z.ArchRel \wedge$
$x.FunAttr = y.FunAttr \wedge y.FunAttr = z.FunAttr \wedge$
$x.Constr = y.Constr \wedge y.Constr = z.Constr) \Leftrightarrow$
$x\ is\_variant\ z)$.

Definition. Let $COMPONENT_S$ be a set of software components of a software system S and $is\_variant_S$ be its variant relation. A set of all equivalence classes in the relation $is\_variant_S$ is
$VARIANT_S = \{ V \mid V = \{ x \mid x \in COMPONENT_S\ \wedge\ x\ is\_variant_S\ y$
$for\ some\ y \in COMPONENT_S \}$
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.

Let us present an example of a software system which includes versions of (some of) its components. The example is taken from the software system KEX, (Návrat *et al.*, 1989; Bieliková

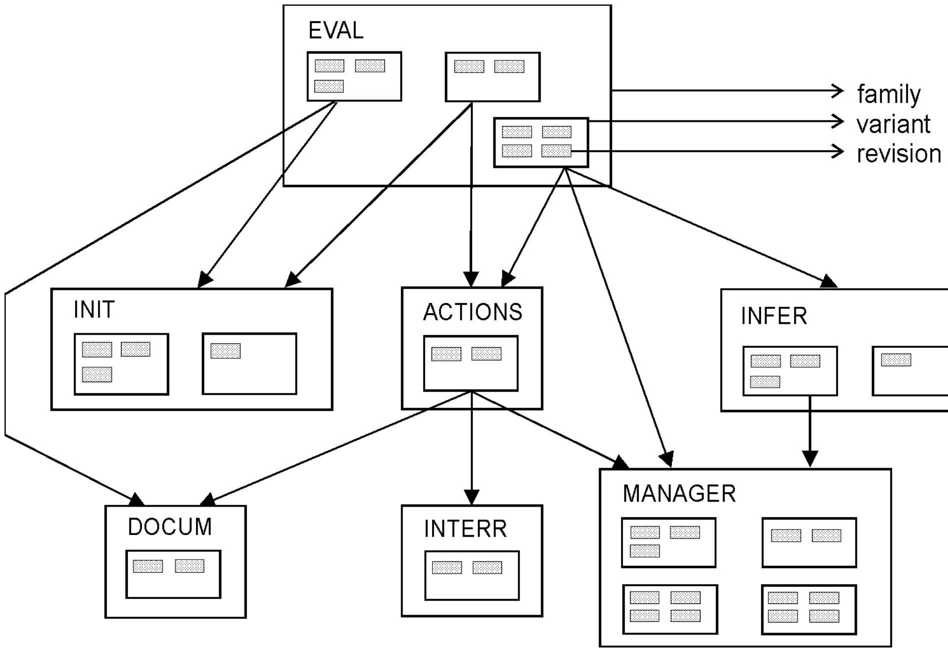

Fig. 1. An example of a software system: partial hierarchy of elements.

et al., 1992). The system elements are shown in Fig. 1 along with architectural relations between them.

In Fig. 1, elements are organised in a hierarchy. Families of software components comprise variants and variants comprise revisions. Architectural relations are defined on the level of variants (they are the same for all revisions within a variant) between a variant and a family of software components. Software system in Fig. 1 consists from seven families 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, a family EVAL includes nine software components.

5. A model of a software system

The concepts introduced above will allow us to formulate a model of a software system. When proposing the model, we attempted to find a model which would support a 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.

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 mebership of a revision in a variant. Architectural relations must be defined explicitly on 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 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. We shall call this relation *has_variant*. From the point of view of a variant, a 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. Taking into account that a software component is determined completely only after a revision has been selected, a resulting configuration is built by selecting precisely one revision for each selected variant.

We propose to represent elements and relations incorporated into a model of a software system by an oriented graph.

Definition. 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

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

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.

Any element of $E, (v_1, v_2) \in E$, called an edge, is of one from among the two mutually exclusive kinds. Either $e_1 \in VARIANT_S, e_2 \in F_S$, i.e. the edge is from A. In this case, the node e_1 is called the A-node. Or $e_1 \in F_S, e_2 \in VARIANT_S$, 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.

A/O graph which models a software system has several properties following directly from definitions of a family of software components, a variant, and a model itself:

- (P1) Every O-node has at least one successor.
- (P2) Every A-node has exactly one predecessor.
- (P3) On every path, A-nodes and O-nodes alternate.

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. (Narayanaswamy, Sacchi, 1987; Ploedereder, Fergany, 1989; Miller, Stockton, 1989; Estublier, 1992).

The model of a software system depicted in Fig. 1 can be expressed by an A/O graph in Fig. 2. To simplify referencing the variants, we invented names for them, Names of variants are derived from a name of the corresponding family of software components combined with a natural number.

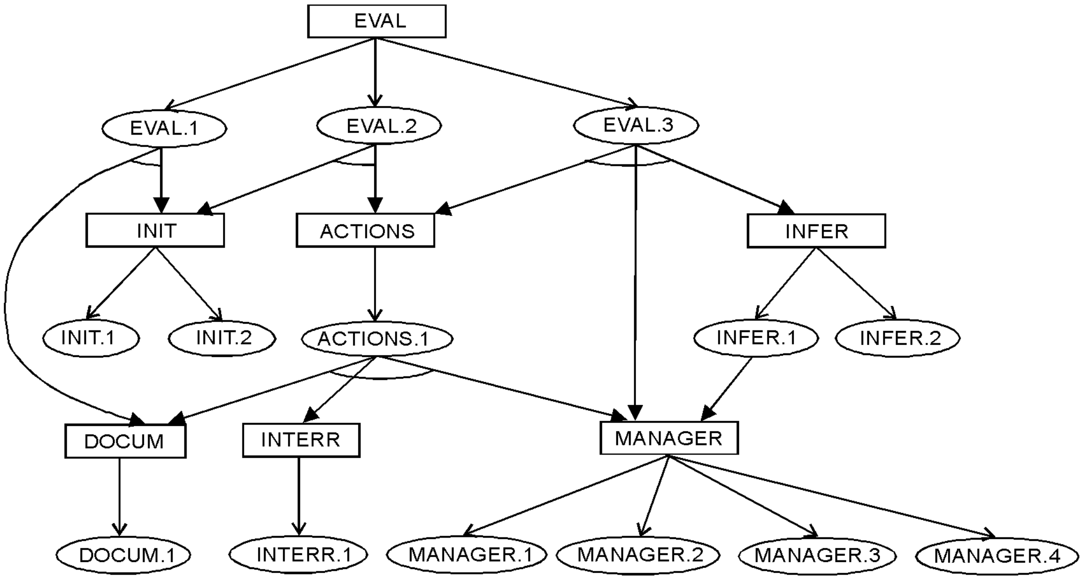

Fig. 2. Software system from above modelled by an A/O graph.

Let us stress once again, that in our model there are represented explicitly only families of software components and their variants. We were led to this choice by the fact that revisions do not represent different concepts or solutions nor they introduce them. Whenever a revision attempted to do so, according to our model it would become a variant. On the other hand, revisions are mere improvements, repairs or enhancements. Creating a revision does not assume any change in architectural relations between components in a software system. That is why they need not be represented in a system's model.

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).

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.

Now we can preview the main points of our method for building a configuration. First, a model of the software system must be available. The model is a suitable way of repesenting all the possible configurations. There can be built several different configurations from it, usually based on different required purposes of the desired configuration. There can be desired a configuration for the end user, a configuration for further development, etc. Such configurations can be specified by different configuration requirements. This second phase results in a generic configuration. In the third phase, a revision for each variant must be selected, resulting in a bound configuration. A detailed description of the method shall be presented elsewhere.

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 bound 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.

6. Experiments

The proposed method for modelling software configurations has been implemented as a part of a larger project aiming to develop a method of configuration building. We performed experiments aiming to analyze its properties. We have developed an experimental 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 (Bieliková, 1995; Bieliková, Návrat, 1995).

7. 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. It is based on a model of software system represented as A/O graph. In our model, we assume there are 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 those of e.g., (Tichy, 1985; Bernard *et al.*, 1987; Leblang, Chase, 1987; Mahler, Lampen, 1988). At the same time, our model is more informed than those cited above in the sense that several models are usually needed to describe the information as one our model. In particular, they allow architectural relations to be defined only between component families.

References

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

Bieliková M., Frič P., Galbavý M., and Vojtek, V. (1992): *KEX - an environment for development of expert systems.* In Proc. of the 14th International Conference on Information Technology interfaces ITI'92, pp. 153–159, Pula. Univ. Computing Centre, Zagreb.

Bieliková M. (1995): *Contribution to Knowledge Based Building of Software System Configuration.* Phd. thesis, Slovak Technical University, Bratislava.

Bieliková M. and Návrat P. (1995): *An approach to building software configuration using heuristic knowledge.* In D. Karpić and V.H. Dobrić, editors, Proc. of the 17th International Conference on Information Technology Interfaces ITI'95, pp. 575–580, Pula.

Estublier J. (1988): *Configuration management: the notion and the tools.* In Proc. Int. Workshop on Software Version and Configuration Control, pp. 38–61, Stuttgart.

Estublier J. (1992): *The Adele configuration manager.* Technical report, L.G.I., Grenoble.

Heimbigner D. (1988): *A graph transform model for configuration management environments.* In P. Hederson, editor, Proc. ACM SIGSOFT'88, pp. 216–225, Boston. ACM Press.

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

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

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

Mahler A. and Lampen A. (1988): *An integrated toolset for engineering software configurations.* In P. Hederson, editor, Proc. ACM SIGSOFT'88, pp. 191–200, Boston. ACM Press.

Mahler A. and Lampen A. (1990): *Integrating configuration management into a generic environment.* ACM Sigsoft Notes, 15(6):229–237.

Miller D.B. and Stockton R.G. (1989): *An inverted approach to configuration management.* pp. 1–4. ACM Sigsoft Notes.

Návrat P., Frič P., Adámy M. and Mladá I. (1989): *KEX: Computer aided knowledge engineering system.* In Proc. Computers'89 Conference, pp. 156–162, Blahová.

Narayanaswamy K. and Scacchi W. (1987): *Maintaining configurations of evolving software systems.* IEEE Transactions on Software Engineering, SE-13(3):325–334.

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

Ploedereder E. and Fergany A. (1989): *The data model of the configuration management assistant.* pp. 5–14. ACM Sigsoft Notes.

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

Reichenberger C. (1989): *Orthogonal version management.* In 2nd International Workshop on Software Configuration Management, pp. 137–140. ACM Sigsoft Notes.

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

Rochkind M.J. (1975): *The source code control system.* IEEE Transactions on Software Engineering, SE-1(4):364–370.

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

Tichy W.F. (1986): *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, pp. 219–236. North Holland.

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

Vescoukis V.C., Psaromiligos J. and Skordalakis E. (1992): *PB-VSS: a software version selection system based on logical programming.* In S. Tzafestas, P. Borne, and L. Grandinetti, editors, Parallel and Distributed Computing in Engineering Systems, pp. 141–146. North-Holland.

Yau S.S. and Tsai J.J. (1987): *Knowledge representation of software component interconnection information for large-scale software modifications.* IEEE Transactions on Software Engineering, SE-13(3):355–361.