Engineering with Computers (1997) 13: 125–133
© 1997 Springer-Verlag London Limited

Engineering with Computers

Object Configurations in Software Engineering Databases

J. Ebert[1] and G. Vossen[2]

[1]Fachbereich Informatik, University of Koblenz, Koblenz, Germany; [2]Institut für Wirtschaftsinformatik, University of Münster, Münster, Germany

Abstract. *We consider the logical organization of database support for software engineering applications, which has to cope with such requirements as the adequate support of object composition and versioning, the derivation of (consistent) configurations, and the provision of application-specific transactions. A new approach to the organization, manipulation and management of multiversion objects in CASE databases is described and investigated, which clearly distinguishes four relevant concepts: documents, versions, configurations and databases. The approach is formally made precise using the concept of* AND/OR *graphs, which renders it possible to cast operations on any of the concepts in terms of graph operations. The levels of abstraction distinguished and maintained throughout the exposition give rise to an appropriate transaction concept: transactions involve graph operations on specific types of objects only; conflicts between transactions can thus be easily identified, and consistency of objects is easily maintained.*

Keywords. AND/OR graph; Configuration management; Software engineering database; Transaction

1. Introduction

The integration of database technology into non-standard applications, among them CASE environments, has been discussed and investigated for many years now, and various solutions have been proposed for a number of problems with which database systems are confronted here [1–3]. One of the key issues is the appropriate *logical* organization of database support for these applications, since this has to account for specific requirements including object composition and versioning, configuration derivation, consistency, and domain-specific transactions. In this paper we offer a novel approach to configuration management in CASE environments based on the formal concept of AND/OR graphs.

In software engineering environments, the central goal is the development of large software products. Typically, these products consist of a variety of interconnected documents, which come in multiple versions and variants, and which are developed in a team effort. Also, these products typically represent an entire family of related products, targeted, for example, towards various computer architectures or machine sizes. Given this situation, it is clear that appropriate version and configuration management are crucial; the versioning mechanism provided must moreover be embedded in an environment that allows for various operations on a product under design: viewing its structure at a conceptual level, viewing a particular configuration at an instance level, retrieving all modules referenced by a particular subroutine, updating some document version to fix a bug, etc.

In this paper, we describe such an environment. In particular, we develop an organizational framework which can serve as a mediator between a software development environment and a database management system. The central notion on which this framework is based is the formal model of AND/OR graphs, a well-known data structure for describing, for example, decomposable production systems in artificial intelligence [4], which has previously been used by other authors [5] as well. In brief, an AND/OR graph consists of two kinds of nodes, AND nodes and OR nodes, where an OR node is here interpreted as a document whose direct successors are its currently available versions; each version in turn is represented by an AND node whose direct successors are the representatives of those documents that it needs, depends on, or refers to.

We show that, in particular, configurations (of software products) can concisely be defined in these graph-theoretic terms. In addition, operations on

Correspondence and offprint requests to: Professor Dr. G. Vossen, Westf. Wielhelms-Universität Münster, Wirteschaftsinformatik, Grevener Strasse 91, D-48159 Münster, Germany.

multiversion objects and on configurations can be rephrased in the form of graph operations. This, in turn, allows for a novel transaction concept for CASE databases, where a transaction is a collection of graph operations to be executed as an atomic and consistency-preserving unit. As a result, a formal foundation of configuration management is given, which even allows for an incorporation of a concrete notion of consistency.

The approach presented in this paper is in part based on the constellation approach originally introduced in [6]: a *configuration*, being a 'snapshot' of some part of an artifact under design, is comprised of at most one version of each object of the underlying constellation; thus, configurations state which versions of distinct objects make sense together. We emphasize, however, that our graph-based model is a conceptual model, i.e. it is not to be understood as an implementation vehicle, but as a logical organization which can be implemented atop any type of database system. However, for reasons that will become clear in the remainder of this paper, we imagine that object-oriented database systems will provide the most suitable underlying databases for this model.

The organization of this paper is as follows: in Section 2, we briefly summarize some relevant background on software engineering; in Section 3, we introduce our model of software engineering databases; to this end, we define AND/OR graphs, their usage in modeling software documents together with their versions, and show how configurations can be introduced as special subgraphs of these graphs; we also discuss how to incorporate a notion of consistency into this model. In Section 4, we demonstrate that a number of important operations on software document databases can uniformly be described in terms of operations on AND/OR graphs. In Section 5 we show that these graph operations can be used as basic building blocks of transactions on software document databases, where collections of such operations are to be executed atomically; we also reconsider consistency in this context, since its preservation is an additional requirement to transactions. Finally, in Section 6 we summarize and discuss our approach, compare it to related work, and indicate questions that deserve future study.

2. The Software Engineering Domain

In this section we briefly summarize central concepts from the domain of software engineering as relevant to the remainder of this paper; we generally assume the reader to have a basic understanding of software engineering [7] and of databases [3].

In software engineering, the central goal is the development of large software products. Typically, these products consist of a variety of interconnected documents (e.g. modules, text fragments, object code, test data), which come in multiple versions and variants, and which are developed in a team effort. Also, these products typically represent an entire family of related products, targeted, for example, towards various computer architectures or machine sizes.

For the purposes of this paper, a *document* is an identifiable object during a software development and maintenance process. A document has a name, a state, and may be handled by appropriate tools. Furthermore, the current state of a document is a particular *set of versions*. A *software product* consists of several distinct documents which are interconnected. During the development of a product, several individuals interact and cooperate towards a common goal; usually various versions of documents are being worked on by different people at the same time.

Since the set of versions of a document d contains different alternative instances v, the combination of exactly one instance of each document leads to versioning of the product into different *configurations*. Thus, a configuration is a set of versions of documents. One reason for keeping different document versions lies in the necessity to produce the software product for different target environments at the same time (e.g. the development might simultaneously be done for small desktop computers, workstations and mainframes or different window management systems should be supported). In this case, versioning of the product stems from different architectural decisions having been taken.

Configuration management is the issue of keeping track of documents during the process of development and maintenance. According to [7], 'configuration management is the discipline of coordinating software development and controlling the change and evolution of software products and components'. Due to the huge number of documents which are to be kept in software development projects, and due to the problems resulting from the multiple access by a large number of developers, a common database is indispensable. Indeed, access to all documents must be synchronized appropriately to allow sharing of components without any loss of (update) information. Thus, a *software engineering database* has to handle a vast number of documents, their instances, possibly annotations (e.g. access rights, modification dates,

authorship) and their relationships, and must additionally act as a common repository of components for a team of software engineers, who access this repository in a shared fashion.

A particular problem in configuration management is that of keeping track of what is a commonly agreed stable version of each document, what is a preliminary sketch of ideas, and what are alternative approaches to a common document. If the goal of software production is not a single product, but a collection of products, these have to be *consistent* (e.g. the delivered source-code must match the module interface specifications). Furthermore, all these documents undergo many changes during the development and maintenance stages of the product.

It follows from the above that a database system that truly wants to support a software development process needs an appropriate version management. A mechanism in that direction is reported upon in the remainder of this paper. This mechanism is based on a formal organizational model, AND/OR graphs, which is intended to be independent of the actual data model of the underlying database. Hence, our model can act as a mediator between a database system and a software development environment; it should be considered as a conceptual model which can help to manage the configuration aspect of software development (and can be implemented atop standard, but preferably object-oriented database technology).

3. Modeling Software Document Databases Using AND/OR Graphs

In this section we present our formal model for reasoning about configuration management. The contents of a database is modeled as an AND/OR graph [4] containing documents and their instances as vertices. In general, an AND/OR graph is a directed, bipartite graph whose nodes are labeled with 'AND' or 'OR' usually in an alternating fashion. Direct successors of an OR-node represent alternatives, while direct successors of an AND-node represent mandatory components of this node. We now define the relevant graph-theoretic notions formally.

A *directed graph* $G = (V, E)$ consists of a non-empty finite set V of *vertices* and a finite set E of *edges* (or arcs), which are ordered pairs of vertices. An edge $e = (v, w) \in E$ is denoted as $v \to w$ with $v = \alpha(e)$ being its start vertex and $w = \omega(e)$ its target vertex. $\Gamma^+(v) := \{w \in V | v \to w\}$ denotes the set of all successors of v.

Let $\to^+$ denote the transitive closure of relation $\to$. A graph G is called *acyclic*, if $v \to^+ v$ does not hold for any $v \in V$. G is a *tree* if it is acyclic, there is a vertex $v_0 \in V$ (called the *root* of the tree), such that $v_0 \to^+ v$ holds for each $v \in V$, and for every $w \in V - \{v_0\}$ there is exactly one vertex v (called the *parent* of w) such that $v \to w$.

3.1. AND/OR Graphs

And AND/OR *graph* is a directed graph G such that:

(i) its vertex set V is the disjoint union of two sets V_{AND} and V_{OR} and
(ii) all edges connect only vertices of different sets.

We here use AND/OR graphs to model documents together with all their currently existing versions. For example, Fig. 1 shows an AND/OR graph whose root, an OR node, denotes a document d_1 which currently has two versions v_a and v_b. Version v_a depends on (e.g. uses, knows or includes) documents d_2–d_5, while version v_b depends on d_2–d_4; both version nodes are of type 'AND'. The nodes for d_2–d_5 are again OR nodes, and these objects have one or two versions as shown. Their versions are represented by AND nodes, three of which share document d_6, which in turn exists in two versions.

It follows from the above description that formally we consider a *CASE database* to be a not necessarily connected AND/OR graph $G = (V, E)$, whose vertex set V is the disjoint union of the set V_{OR} of conceptual *documents* and the set V_{AND} of *instances* (of documents). Hence, OR vertices represent the documents stored in

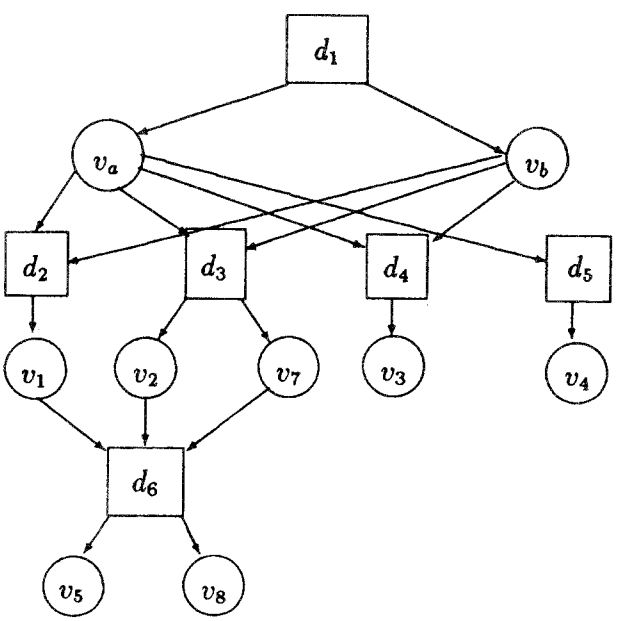

Fig. 1. An AND/OR graph.

the repository. Documents are the *conceptual objects* that a product might *consist of.* Since for each document there might be zero or more versions, the set $\Gamma^+(d)$ of a document vertex d is the set of alternative instances corresponding to d. The edges connecting a document to its instances will be called `hasAsVersion` edges. Correspondingly, `AND` vertices represent the data which make up *instances* of documents. A version may *depend on* zero or more other documents. Thus, the set $\Gamma^+(v)$ of an instance vertex v is a set of documents, which must be present in the database if v is present. These edges will be called `dependsOn` edges. Note that `dependsOn` edges may be further refined by more elaborate relations between documents, e.g. `knows`, `uses Syntactically`, `usesSemantically`, etc., [8], depending on which architectural concept is used by the developers.

A version v of a document d is a particular instance of d. To fully describe the notion of version, the respective document as well as its respective representation have to be provided. At our abstract level, a *version* is a (two-vertex) subgraph H of a given CASE database G consisting of one vertex $d \in V_{\text{OR}}$, one vertex $v \in V_{\text{AND}}$, and one edge $e \in E$ with $\alpha(e) = d$ and $\omega(e) = v$.

3.2. Configurations

A (complete) configuration should contain exactly one version for each of its documents, and for a given instance v it should contain (versions of) all the documents that v depends upon. Thus, a configuration is a non-empty subgraph H of a given CASE database G satisfying the following conditions:

(i) for each `AND` node v in H, all direct successors of v are in H,
(ii) for each `OR` node d in H, at most one direct successor of d is contained in H.

As an example, Figs 2 and 3 show two configurations which are subgraphs of the `AND/OR` graph shown in Fig. 1. Formally, a *configuration* is a subgraph H of database G satisfying

(i) for each $v \in V^H_{\text{AND}}$, $\Gamma^+(v) \subseteq V^H$ and
(ii) for each $d \in V^H_{\text{OR}}$, $|\Gamma^+(d) \cap V^H| \leq 1$.

Note that neither CASE databases nor configurations need to be acyclic or even trees.

Notice also that the previous definition captures the essential requirements from [6] as well as from [9–11] that a configuration is closed under composition (here called `dependsOn`), and that at most one version of each relevant object is included.

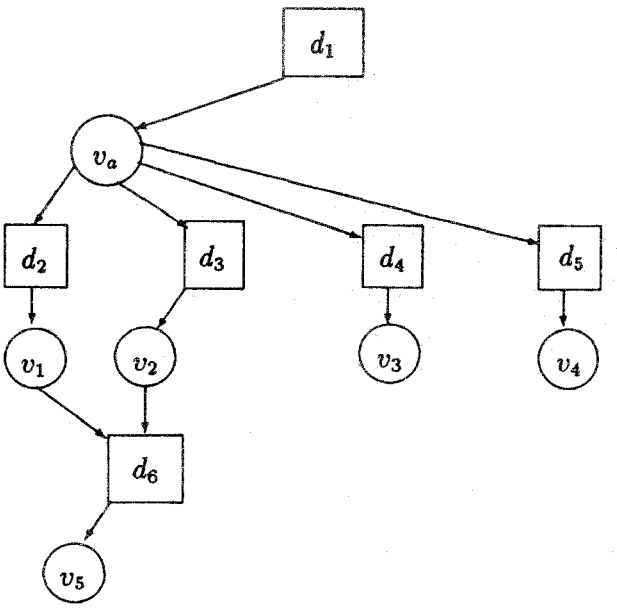

Fig. 2. A configuration.

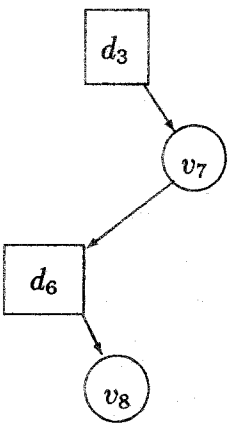

Fig. 3. Another configuration.

3.3. Consistency of Configurations

We want configurations to be units of consistency, as in [6]. In order to be able to make this precise, we assume that for each object at least some structure is given in the form of *typed attributes.* For example, a document might have as attributes the name of the *group* responsible for it, or the *language* in which it has to be written. An instance may bear the name of its *author*, and may also have a *language* attribute.

Here it is reasonable to assume that a document and its implementation have the same *language* attribute, and that the *author* of an instance is a member of the responsible *group.* This leads to a consistency condition on *versions.* Furthermore, one might want that all parts of a *configuration* have the

same *language* attribute, thus leading to conditions on configurations as well. More generally, arbitrary consistency conditions may be formulated on any of the subgraph classes (i.e. versions or configurations), using some graph-class specification language (like GRAL [12]).

If the documents to be modeled are assumed to have attributes describing some structure, a structuring of such conditions can be achieved by first defining 'elementary' subgraphs as follows: an *elementary* subgraph is

- either an AND node together with all its direct successors, i.e. an elementary configuration,
- or an OR node together with one of its direct successors, i.e. a version.

According to the above, we assume elementary subgraphs to have predicates over the values of all nodes involved. As elementary subgraphs are composed into larger ones, and finally into configurations, the obvious thing to do with their associated integrity constraints is to take their Boolean conjunction. Thus, a configuration is *consistent* if all elementary configurations and versions contained in it satisfy their associated predicates.

We will return to the issue of consistency in Section 5 when we consider transactions on CASE databases.

4. Operations on Software Document Databases

In this section, we consider the operational aspect of software engineering databases and introduce a number of corresponding operations; these will be used in the next section as the basic building blocks of transactions. Since *documents*, *versions*, and *configurations* are properly contained one in the other, e.g. versions get grouped into configurations, four corresponding levels of consistency preserving operations may be defined, namely

1. document operations,
2. version operations,
3. configuration operations, and
4. database operations.

In this section, we introduce such operations. Since the databases we consider are graphs, we can state these operations in terms of graph operations. Therefore, we define the graph operations needed first. The operations are given operationally as manipulations of the AND/OR graph defined above. As will be seen later, this approach even allows for an appropriately layered notion of conflict.

Notice that the level of abstraction chosen here is at the creation/deletion level only. Independent from those operations there is also an update/edit level for any given database, which allows a user to edit a given instance of a document or to make changes to it. In the remainder of this section, let G be an arbitrary, but fixed typed and directed graph representing the software engineering database under consideration.

4.1. Graph and Subgraph Operations

To define operations on graphs, we use a pseudo-code notation based on the set of basic operations and control structures given in [13]. We assume the basic operations on graphs in general as shown in Table 1.

In addition to these basic graph operations, we find it convenient to additionally assume the availability of a set of operations for handling subgraphs. To this end, recall from the previous section that in our model versions and configurations are special subgraphs which need to be manipulated. In the sequel, data type 'subgraph' is assumed to be a data type which is only defined relative to a given graph. Table 2 shows the basic operations on subgraphs, where as before G is a global graph.

In the following subsections, we will use the basic operations introduced here to define more complex

Table 1. Basic operations on graphs

(1) `createVertex (t: vertexType): vertex`
creates and returns a new vertex in graph G with type t

(2) `deleteVertex (v: vertex)`
precondition: v is a vertex in G
deletes vertex v in graph G (including all incident edges)

(3) `createEdge (v, w: vertex; t: edgeType): edge`
precondition: v and w are vertices in G
creates and returns a new edge from vertex v to vertex w in graph G with type t

(4) `deleteEdge (e : edge)`
precondition: e is an edge in G
deletes edge e in graph G

(5) `alpha (e: edge): vertex`
precondition: e is an edge in G
returns the start vertex of edge e in graph G

(6) `omega (e: edge): vertex`
precondition: e is an edge in G
returns the target vertex of edge e in graph G

(7) `for all edges e typed t out of v do S`
precondition: v is a vertex in G
performs statement list S for all edges e with edge type t whose start vertex is v

Table 2. Basic operations on subgraphs

(1) `createSubgraph ( ): subgraph`
creates and returns a new and empty subgraph of G

(2) `deleteSubgraph (H: subgraph)`
precondition: H is a subgraph of G
deletes the subgraph H of graph G

(3) `addVertexToSubgraph (v: vertex; H: subgraph)`
precondition: H is a subgraph of graph G and v is a vertex of graph G and not of subgraph H
adds vertex v to subgraph H

(4) `isVertexInSubgraph (v: vertex; H: subgraph): boolean`
precondition: v is a vertex of graph G
decides whether vertex v is a vertex of subgraph H

(5) `removeVertexFromSubgraph (v: vertex; H: subgraph)`
precondition: H is a subgraph of graph G and v is a vertex of graph G and subgraph H
removes vertex v from subgraph H

(6) `addEdgeToSubGraph (e: edge; H: subgraph)`
precondition: H is a subgraph of graph G, e is an edge of graph G and not of subgraph H but the start and target vertices of e are in H
adds edge e to subgraph H

(7) `isEdgeInSubgraph (e: edge; H: subgraph): boolean`
precondition: e is an edge of graph G
decides whether edge e is an edge of subgraph H

(8) `removeEdgeFromSubgraph (e: vertex; H: subgraph)`
precondition: H is a subgraph of graph G and e is an edge of graph G and subgraph H
removes edge e from subgraph H

operations on software engineering databases. To this end, the types `document`, `version` and `configuration` will be modeled by the types `vertex`, `edge` and `subgraph`, respectively. All new operations defined below will be implemented on the basis of the graph and subgraph operations given in this subsection.

4.2. Document Operations

Let G be an AND/OR graph model of a CASE database as introduced in Section 3. Then `AND` and `OR` are vertex types and `hasAsVersion` and `dependsOn` are edge types allowed in G.

Since documents are modeled as `OR` vertices, type `document` is used to model documents, and is implemented as a `vertex` of type `OR`. On the `OR` vertices of G, only create and delete operations are allowed. These are defined as shown in Table 3.

Table 3. Document operations

(1) `createDocument ( ): document`
creates a new document
implementation:
```
return createVertex(OR)
```

(2) `deleteDocument (d: document)`
deletes document d
implementation:
```
deleteVertex(d)
```

Table 4. Version operations

(1) `createVersionOfDocument (d: document): version`
creates a new version of document d
implementation:
```
v := createVertex(AND);
return createEdge(d, v, hasAsVersion)
```

(2) `deleteVersion (x: version)`
deletes version x
implementation:
```
v := omega(x)
deleteVertex(v)
```

4.3. Version Operations

A document may have any number of versions, which must be created and deleted as well. In our graph model versions are one-edge subgraphs. Thus, they can be modeled by edges in the graph. So type `version` is implemented as an `edge` of type `hasAsVersion`. Operations on versions are the ones shown in Table 4.

4.4. Configuration Operations

Finally, configurations may be added to and deleted from a CASE database. Configurations are subgraphs of G (i.e. type `configuration` is implemented as type `subgraph`), where the actual instance of the latter has to satisfy consistency constraints. The operations shown in Table 5 may be performed on configurations; it can be shown that these keep the configuration conditions invariant. To allow maintenance of configurations, it must be possible to insert and delete versions with respect to them; to this end, we have the operations shown in Table 6. Notice that the second operation shown in Table 6 just removes versions from configurations, but not documents; in order to accomplish the latter, one approach would be to define configurations as *rooted*

Table 5. Basic operations on configurations

(1) `createConfiguration ( ): configuration`
creates a new configuration
implementation:
```
return createSubgraph( )
```

(2) `deleteConfiguration (H: configuration)`
precondition: H is a configuration in G
deletes configuration H
implementation:
```
deleteSubgraph(H)
```

Table 6. Manipulation operations for configurations

(1) `insertVersionIntoConfiguration (x: version; H: configuration)`
precondition: H is a configuration in G and x is a version in G but not in H
inserts a document version x into a configuration H
implementation:
```
v := alpha (x)
if not isVertexInSubgraph(v, H) then
  addVertexToSubgraph(v, H)
v := omega(x)
if not isVertexInSubgraph(v, H) then
  addVertexToSubgraph(v, H)
addEdgeToSubgraph (x, H)
(* assert consistency *)
for all e typed dependsOn out of v do
  if not isEdgeInSubgraph(e,H) then
    if not isVertexInSubgraph
    (omega(e), H) then
      addVertexToSubgraph(omega(e), H)
    addEdgeToSubgraph (e, H)
```

(2) `deleteVersionFromConfiguration (x: version; H: configuration)`
precondition: H is a configuration in G and x is a version in H
deletes document version x from configuration H
implementation:
```
v := omega(x)
removeEdgeFromSubgraph (x, H)
removeVertexFromSubgraph (v, H)
```

subgraphs and then delete all documents which can no longer be reached from such a root.

5. Transactional Concepts

We are now ready to consider the operational aspect of CASE databases from a user's point of view. To this end, our perception is that a suitable notion of *transaction* must be available for working with a CASE database, in the sense that a user can group database operations together and be assured by the system that these operations are executed according to the ACID principle [14]. However, previous studies have shown that CASE databases have particular and novel requirements to a transactional support, and we try to reflect these in our proposal.

The transaction model we consider in this paper is similar in spirit to the proposal made in [6], in which each transaction can be of one out of three distinct *types*; since the types in [6] are orthogonal, conflicts between transactions of distinct types do not occur, and only transactions of the same type eventually need synchronization. In our approach, on the other hand, such an orthogonality is no longer present; nevertheless we are able to establish distinct transaction types and make precise statements about their interaction.

The graph model we employ in our approach differentiates between four kinds of concepts, each of which corresponds to classes of subgraphs of a given database graph:

1. *documents* and *instances* correspond to vertices of types `OR` and `AND`, respectively;
2. *versions* correspond to edges of type `has AsVersion` including their endpoints;
3. *configurations* correspond to `AND/OR` subgraphs;
4. *databases* correspond to an entire graph.

The important observation is that these concepts form a proper inclusion hierarchy as follows:

- documents and instances are included in versions,
- versions are in turn included in configurations, and
- configurations are included in the database graph.

Since everything is modeled by classes of (sub-) graphs, consistency conditions may be specified by appropriate predicates on the respective graph class. For example, if `language` is an attribute of every vertex, a predicate on versions might be as follows:

$$(\forall x\colon \texttt{version})\ \alpha(x).\,\texttt{language} = \omega(x).\,\texttt{language}$$

The inclusion hierarchy also leads to a layered approach of consistency. Indeed, consistency of a subgraph implies consistency of all other subgraphs contained in it, in the following sense:

- consistency of the database implies consistency of all configurations contained in it;
- consistency of a configuration implies consistency of all versions contained in it;
- consistency of a version implies consistency of its corresponding document and its instance.

Our transaction model is directly based on the above observations; indeed, for each of the concepts

listed above we have a corresponding kind of transaction:

1. A *database transaction* is a transaction which applies any database operation (or sequence thereof); however, it does not touch configurations. Hence, a database transaction keeps database consistency invariant.
2. A *configuration transaction* operates on a particular configuration, without going into specific versions. Thus, it keeps configuration consistency invariant.
3. A *version transaction* acts upon a specific version, but without changing the underlying documents, so that it preserves version consistency.
4. Finally, a *document/instance transaction* allows the manipulation of the attributes of a document or an instance. Clearly, it hence maintains the respective invariants.

In other words, the transactions we image for CASE databases each have a definite realm of operation, according to the inclusion hierarchy formed by the various concepts present in such a database as described above. At each particular level of this hierarchy, a well-defined set of graph operations is available for use in a transaction. Importantly, a transaction corresponding to any such level does not touch any 'deeper' level of the hierarchy. Thus, our transactions allow for some form of independence, and a notion of conflict between them can straightforwardly be stated as follows: a transaction t operating on object k at `level` i can only be in conflict with

- other transactions operating on the same object k, or
- transactions t' working on objects k' at lower levels, provided k' is contained in k, or
- transactions t'' working on objects k'' at higher levels, provided k is contained in k''.

Based on this notion of conflict, it appears straightforward to design scheduling mechanism for correctly processing concurrent transactions on CASE databases [14]. Notice that such mechanisms will in particular allow for cooperation between software engineers, due to the layering on which our approach is based.

6. Discussion

In this paper, we have introduced a formal model of software engineering databases which relies on well-understood graph-theoretic concepts, and adequately takes various central issues apart in terms of modeling. Indeed, the model of a database as an `AND/OR` graph allows one to view the database at various levels of abstraction and in an application-dependent way. For example, a developer may look at the structure of a project as a whole, or he or she might extract specific configurations that have to satisfy certain selection criteria. Operations on software engineering databases then amount to graph operations, which can involve documents, versions, configurations, or an entire database. Importantly, transactions can easily be specified using these graph operations as their basic building blocks; for such transactions, consistency preservation can be made precise, and a notion of conflict relevant to concurrent processing can be stated.

The approach we have described can be seen as a formal framework for an integration of configuration management capabilities into software-engineering databases. Indeed, if such a database follows our model, it can reflect both an explicit software architecture and the formation of versions as well as variants for all kinds of software documents. This is due to the fact that all these aspects are captured within a single uniform graph-theoretic model, where in this paper `dependsOn` edges reflect the architectural aspect, while `hasAsVersion` edges reflect configuration management information. Furthermore, from a more practical perspective, our work provides an abstract conceptual model into which the functionality of well-known UNIX tools like `make`, `sccs`, or `rcs` can be mapped. These tools are widely used for configuration description and management.

We now compare our approach with others which have previously been made in the literature; since our goal was to establish a well-defined operational approach for working with CASE databases, we concentrate here on a comparison of transactional aspects. Proposals that have recently been made to extend the classical database notion of a transaction to the CASE domain are surveyed in [15]. All of these share the perception that the ordinary transaction concept guaranteeing the ACID properties is too restrictive in this area. As a result, researchers have relaxed the notion of a transaction in various ways. For example, [16, 17] consider ways of allowing more concurrency, which is necessary in to order to support cooperative work between programmers involved in a large software project. However, [16, 17] stick to an implementation-dependent level w.r.t. transaction operations, i.e. transactions are considered sequences of read and write operations, which are now executed under user control. Reference [18] introduces a nested transaction model for CASE databases, which also allows for user interaction and cooperation.

The major differences between our approach and the one described in [18] are that (1) we ignore technicalities such as read and write operations, based on the view that higher levels of a abstraction are available in this context and can be exploited by transactions, and (2) we stick to the view that transactions are flat, since this is most easy to comprehend by a user. On the other hand, our transaction types still cover all relevant aspects of user interaction with a CASE database, so that in particular the cooperation aspects as disucssed in [16–18] are present here as well.

Our approach to the design of transactions in some sense concurs with the one described in [6]. There, transactions can either operate on object versions, on configurations, or on so-called constellations, where the latter are vehicles to describe the compositional structure of a complex object independent of particular versions. Since consistency is confined to configurations, transactions operating on object versions within the context of a configuration are traditional transactions; they are even orthogonal to transactions manipulating configurations and constellations. An object-version transaction manipulates object versions in one or several configurations of a constellation. It does not need to access all the object versions comprising a configuration, but must take the entire configuration from one consistent state to another. A configuration transaction is used to manipulate configurations of a constellation; it may derive a new configuration, or create or destroy an existing one. Finally, a constellation transaction is used to manipulate a constellation as a whole. With the collection of graph operations we have defined in Section 4, these transaction types can vastly be recast in our context, with the difference that we do not support the notion of a constellation, and that our objects form an inclusion hierarchy as exhibited in Section 5.

References

1. Adams, E.W.; Honda, M.; Miller, T.C. (1989) Object management in a CASE environment; Proceedings 11th IEEE International Conference on Software Engineering, 154–163
2. Dittrich, K.; Gotthard, W.; Lockemann, P. (1987) DAMOKLES: the database system for the UNIBASE software engineering environment, IEEE Data Engineering Bulletin, 10, 1, 37–47
3. Vossen, G. (1991) Data Models, Database Languages and Database Management Systems, Addison-Wesley, Reading, MA
4. Nilsson, N.J. (1980) Principles of Artificial Intelligence, Springer-Verlag, Berlin
5. Tichy, W.F. (1982) A data model for programming support environments and its applications. Schneider, H.J.; Wasserman, A.I. (Editors), Automated Tools for Information Systems Design and Development, North-Holland, Amsterdam, 31–48
6. Cellary, W.; Vossen, G.; Jomier, G. (1994) Multiversion object constellations: a new approach to support a designer's database work, Engineering with Computers, 10, 230–244
7. Ghezzi, C.; Jazayeri, M.; Mandrioli, D. (1991) Fundamentals of Software Engineering, Prentice-Hall, Englewood Cliffs, NJ
8. Ebert, J.; Engels, G. (1989) Konzepte einer Softwarearchitektur-Beschreibungssprache, Software-Entwicklung: Konzepte, Erfahrungen, Perspektiven (Lippe, W.M., Editor), Springer-Verlag, Berlin, 238–250
9. Agrawal, R.; Jagadish, H.V. (1989) On correctly configuring versioned objects, Proceedings 15th International Conference on Very Large Data Bases, 367–374
10. Cellary, W.; Jomier, G. (1990) Consistency of versions in object-oriented databases; Proceedings 16th International Conference on Very Large Data Bases, 432–441
11. Vidyasankar, K.; Dampney, C.N.G. (1988) Version consistency and serializability in design databases, Proceedings 2nd International Conference on Database Theory, Springer LNCS, Berlin, 326, 368–382
12. Ebert, J.; Franzke, A. (1995) A declarative approach to graph based modeling, graphtheoretic concepts in computer science (Mayr, E. W.; Schmidt, G.; Tinhofer, G., Editors), Springer-Verlag, Berlin, 38–50
13. Ebert, J. (1987) A versatile data structure for edge-oriented graph algorithms, Communications of the ACM, 30 513–519
14. Vossen, G.; Groß-Hardt, M. (1993) Grundlagen der Transaktionsverarbeitung, Addison-Wesley (Deutschland), Bonn
15. Barghouti, N.S.; Kaiser, G.E. (1991) Concurrency control in advanced database applications, ACM Computing Surveys, 23, 269–317
16. Kaiser, G.E. (1989) A marvelous extended transaction processing model, Information Processing 89 (Ritter, G.X., Editor), Elsevier Science Publishers, Amsterdam, 707–712
17. Kaiser, G.E. (1990) A flexible transaction model for software engineering, Proceedings 6th IEEE International Conference on Data Engineering, 560–567
18. Korth, H.F.; Silberschatz, A. (1990) Long-duration transactions in software design projects, Proceedings 6th IEEE International Conference on Data Engineering, 568–574