Ancestor-Controlled Submodule Inclusion in Design Databases

Lin Yu
linsky@albanycs.albany.edu

Daniel J. Rosenkrantz
djr@albanycs.albany.edu

Department of Computer Science
State University of New York at Albany
Albany, New York 12222

ABSTRACT A paradigm is proposed for representing hierarchically specified design data in CAD database systems where there are to be alternate expansions of hierarchical modules. The paradigm is based on an ancestor-based expansion scheme to control which instances of submodules are to be placed within each instance of a given module, and is formalized via a model that we call a VDAG. The approach is aimed at reducing storage space in engineering design database systems, and providing a means for designers to specify alternate expansions of a module. The VDAG model is defined, and a mechanism whereby a VDAG generates an exploded forest of design trees is described. Algorithms are provided which given a VDAG or VDAGs, generate design forests, determine whether one module is contained by a larger module, extract a version from a VDAG, test whether two VDAGs are equivalent, and try to reduce the size of a VDAG. Problems such as module containment and VDAG inequivalence are shown to be NP-complete; and the problems of finding a minimum-sized VDAG equivalent to a given VDAG are NP-hard.

1. Introduction

We investigate a simple model, called a VDAG, for representing hierarchical specifications where there is to be alternate expansions of hierarchical modules. The VDAG model features an ancestor-based expansion scheme to control which instances of submodules are to be placed within each instance of a given module. The model is aimed at reducing storage space in engineering design database systems, and providing a capability for designers to specify alternate expansions of a module. The VDAG model is not restricted to any specific design applications, such as VLSI design or civil engineering design.

This work is motivated by the importance of handling alternatives and versions in database systems for engineering design databases [DL, Kat, KL]. One aspect of this problem is to efficiently maintain configuration information and to compactly represent multiple versions [FM, KB, NS, YR, YT]. Another aspect is to provide mechanisms whereby designers of a hierarchically specified engineering project can conveniently specify alternative expansions for a given module when it occurs as a submodule within larger modules [BK, VHDL].

In existing version control systems for software engineering and document generation systems, the differences between two versions are usually described on a line basis [Roc, Tic]. These differences can be computed by algorithms such as those in [Hec, HM] and the utility program diff in Unix system on two given files [U]. With this approach, version differences that are kept track of are line differences. Record differences are used in [SL]. In this paper, the differences of two design versions considered are module differences, in that the basic granularity of differences that are kept track of are instances of submodules within another module.

Many different mechanisms have been proposed to reduce the amount of storage space used for storing data with versions/alternatives, for instance, an AVL DAG model was proposed in [FM] for storing different versions of text files. The problem considered in [FM] is that there is a set of versions to be stored, each of which can be represented as a tree. The set of trees may share common subtrees and a data structure is proposed which keeps one copy of certain common subtrees. From an abstract perspective, the method used in [FM] is to store a forest of trees compactly, by storing only one copy of common subtrees. The problem can be envisioned as that of producing a compact representation of a given forest. For this problem, the kind of ancestor-based control of tree expansion considered here can lead to more compact representations.

In a CAD environment, design objects are usually designed in a hierarchical fashion, and the user supplied input information is a set of hierarchically specified objects, rather than an explicit forest of design trees. A major concern is providing mechanisms whereby users can conveniently specify alternatives and versions in the spirit of their customary hierarchical design methodology.

Different mechanisms have been proposed to support design alternatives in CAD database of hierarchically specified designs. For instance, [VHDL] uses a scheme whereby a module can have alternate bodies which share a common interface, and a configuration of a module can be created by specifying which alternative body to use for submodules within that module. A configuration can have a different expansion specification for each instance of the same
module type within it. Another mechanism provided in [VHDL] is conditional expansions, which can be based on the value of a generic parameter. [BK] proposes a model, whereby a module can have alternate implementations (corresponding to bodies in VHDL), which share a common interface. A body can have *instantiations* of submodules, and can be *parameterized* by a specification of which body to use for specified occurrences of submodules. However, the model in [BK] does not provide any explicit mechanism to control expansion. Furthermore, in the model of [BK] certain kinds of alternatives are not supported conveniently, in the sense that they require the creation of separate implementations, even though they may differ only slightly. For example, suppose we want module A to contain certain submodules when it is used as a submodule of B and to contain some other submodules when it is used as a submodule of C. In this case, the model in [BK] would require two distinct implementations for module A.

A hierarchically specified set of modules can be represented as a multi-graph, where each vertex represents a module. If a design module B is used within a larger module A, we say "B is a direct submodule of A", and there is an arc from the vertex representing A to the vertex representing B. If there is a directed path from A to B, we say "B is a submodule of A".

Hierarchically specified design modules can be envisioned as *trees*. For instance, suppose that within a design module A there are three submodules where two are instances of B and the other is C, within each module B there are submodules D and E, within each module C there are two submodules that are instances of F, and each F contains two submodules that are instances of G. Fig 1.1 shows module A at different levels of the abstraction, and the tree representation of module A is shown in Fig 1.2. Note that module A includes the vertex A and all of its descendants in the tree. In the tree, for each arc (u,v), there is a function st(u) (where "st" stands for *stamp*) that provides information about the occurrence of the module v as a submodule within the module u.

In this tree representation, there are two copies of B, D, E and F, respectively, and four copies of G. If the tree were to be stored directly, there would be a copy of the design data for each *instance* of a module within A; e.g., there would be four copies of G. However, in CAD systems, a more succinct representation is usually used for hierarchically specified designs; namely a directed acyclic multigraph (dag). The dag representation for the same module A is shown in Fig 1.3.

In this dag representation, only one copy of the design data for each module is kept, regardless of the number of the instances of the module involved in the design. The dag representation uses an appropriate stamp to keep track of information about instances of submodules within other modules, such as the two instances of G within F. Since the dag representation reduces duplication of design module descriptions, it is cheaper to store hierarchically specified design data this way. Also, designers typically use a hierarchical approach to design their modules, so the dag would typically capture the design in the form specified by the designer.

We consider the following problem: How to succinctly represent hierarchically specified design module data that supports design alternatives in version control. To illustrate this, we give an example shown in Fig 1.4, where the two multigraphs represent two different versions of a design module A.

Since we do not want to keep multiple copies of the same module for different versions, we can use the following scheme, as illustrated in Fig 1.5, to store *versioned* hierarchically specified design module data. Under this scheme, we add one source vertex for each design version of module A (note that the newly added source vertices are "dummies" which do not contain actual design data), and we place labels on each arc to indicate to which version(s) the arc belongs.

Since the amount of storage for the labels would often be relatively small compared with the amount of storage saved from eliminating duplicated copies of submodules, this representation is more succinct than a method that keeps all of the design data for each version in separate files. If the number of versions is large and the differences between versions are relatively small, the storage savings can be quite significant.

Of course, since we are concerned with succinctness of the design representation, we would like to represent the labels succinctly. For example, in Fig 1.5, we might replace the two labels for the two arcs from A to B by {A} instead of (V1, V2) since each instance of A for either version contains two instances of B. Thus it is appealing to consider a more general scheme in which the elements of a label for an arc (u,v) are either u or some ancestors of u in the dag. The example in Fig 1.6 illustrates how the generalized scheme can be used to represent the two designs of Fig 1.4.

The VDAG model is a formalization of the technique used in Figs. 1.5 and 1.6 ("VDAG" stands for "versioned dag"). In this formalization, each module is represented by a VDAG vertex having a unique tag. Each possible use of one module as a direct instance within a larger module is represented as a VDAG arc. The arc is labeled with a specification as to when the potential instance should indeed be a real instance within the larger module. This specification is ancestor-based. It can say that the instance should always be included within the larger module, or it can say that the instance should be included only if some member of a given list of modules is an ancestor of the larger module. Not every design forest can be generated by a VDAG, where expansion of a module is based on its ancestors. A forest which cannot be generated by a VDAG would have to be modified in order to be VDAG generatable. Such a modification would take the form of changing the identity of some of the vertices in the forest so that they can be
represented by distinct vertices in the VDAG. However, a standard model of hierarchical designs would require even more duplication.

Under a CAD database system that uses the VDAG model, a design engineer would typically build a VDAG by specifying the vertices with their tags and arcs with their stamps. The design engineer would put appropriate labels on the arcs for controlling expansions of that VDAG, according to the design need. Note that this VDAG model supports incremental designs in that vertices and arcs can be added to or deleted from an existing VDAG easily.

The remaining sections are organized as follows. In Section II, we present some basic definitions. We examine some properties related to the VDAG model in Section III. Equivalence problems are considered in Section IV. VDAG simplification problems are considered in Section V. Algorithms are provided for many computational problems associated with VDAGs, and certain problems are shown to be NP-hard. The reason for the NP-hardness of these problems is the high level of compression of design data in the VDAG representation. Some problems which are NP-hard in terms of the VDAG representation have efficient polynomial time algorithms in terms of the size of the represented design object. Due to limited space, we omit or sketch the proofs of some theorems. For more details, see [YR2].

II. Basic Definitions

Definition. In a dag G(V, A) where V is a set of vertices and A a set of arcs, a vertex u is an ancestor of a vertex v if u is in v or there is a directed path from u to v in G. For a given dag and a vertex s, asc(s) is the set of ancestors of s.

Definition. A design tree is a triple (T, t, st) where T is a tree, t is a function that assigns each vertex v of T a string t(v) called the tag of v, and st is a function that assigns each arc a of T a string st(a) called the stamp of a.

Note that a design tree is an unordered tree. If it is desired that the ordering of children of a vertex should have some significance, this ordering information should be incorporated in the stamps on the arcs going to the children. In that case, the arcs exiting from the same vertex will have distinct stamps.

Definition. A design forest is a set of design trees such that each tag of a root vertex occurs nowhere else in the forest.

The tag on the root of each tree serves to uniquely identify the tree as a design module, or perhaps as a particular version or alternative of a design module.

In the future we use “tree” and “forest” to mean design tree and design forest, respectively.

Definition. A VDAG is a four tuple (G, t, st, l) where G is a directed, acyclic multigraph with vertex set V and arc set A; t is a function mapping each vertex v to a unique string denoted by t(v), (t for tag); st is a function mapping each arc a to a string denoted by st(a), (st for stamp); and l is a function mapping each arc from u to v in A to a nonempty subset of ancestors of u (l for label).

Although a VDAG is a multi-graph, we will frequently denote an arc from vertex u to vertex v as (u, v); when the presence of multiple arcs between u and v is an issue, we will be more careful in identifying each arc. Note that the label of an arc (u, v) might include u. In fact, if the label includes both u and some other elements, then the label can be simplified by deleting these other elements.

In a design tree, each vertex v represents a module. The tag t(v) on a vertex v is the actual design data for the module represented by that VDAG vertex. The information in the tag may also include an interface description describing how it is connectable when used as a direct submodule within a larger module. For example, the tag might contain a formal parameter list, comparable to a list of input and output ports in VLSI design application. An arc a from vertex u to vertex v represents an instance of module v as a submodule occurring within module u. For arc a, st(a) is the information necessary for specifying how the instance of v occurs inside u. For example, the stamp may specify the location and/or orientation of the instance within u. The stamp might also contain an actual parameter list; for instance, in the VLSI application st(a) might specify which signal of u is connected to each port of the instance.

The interpretation of w ∈ l(a), where a goes from u to v, is that if module w has an instance of u as a submodule, or a submodule of a submodule, etc., then each such instance of u within w contains an instance of v within the instance of u.

Each vertex of the VDAG might look like the one shown in Fig. 2.1 where vertex v has k exiting arcs pointing to (not necessarily distinct) submodules v1, ... , vk. The connection data field contains stamps, labels and pointers to submodules. The design data field contains the tag of v. Fig. 2.1 is only intended to be suggestive of how the design information might be stored, and many variations are possible. For instance, the information for each vertex might be kept in separate file.

Example. Given a forest of two trees representing two versions of a design shown in Fig. 2.2(a), a possible VDAG representation is in Fig. 2.2(b).

From the examples given above, we observe that if there is a path in the forest that starts at a vertex with tag A and ends at a vertex with tag B, then the module A corresponding to the former vertex contains an instance of B as a submodule. Each such path in the forest corresponds to a VDAG path that starts at a vertex with tag A and ends at a vertex of tag B, and has appropriate labels and stamps on its arcs. Similarly, for each properly labeled path in the VDAG, there should be a corresponding path in the design forest. However, care is needed in formalizing the concept of properly labeled paths. For instance, in Fig. 2.2(a) A contains D but B does not, even though there seems to be a
path from B to D in Fig 2.2(b). To capture the concept of a properly labeled VDAG path that corresponds to the containment relationship, we have the following definition. The idea behind this definition is that a given path can be extended by a given arc leaving the path's endpoint only if the path contains at least one vertex which is an element of the given arc's label.

**Definition.** In a given VDAG $(G, t, e, l)$, a valid path from $v_0$ to $v_k$ is either a null sequence of arcs, in which case $v_k = v_0$, or is a sequence of arcs $a_1, a_2, ..., a_k$ in $A_1, k \geq 1$, such that there is a sequence of vertices $v_0, v_1, ..., v_k$ for which all $i, 0 < i \leq k$, arc $a_i$ connects $v_{i-1}$ to $v_i$, and the intersection of $l(a_i)$ and $\{v_0, v_1, ..., v_{i-1}\}$ is nonempty. A generating path is a valid path starting at a source vertex.

Note that, by the definition of a valid path, for each $v$ in $V$, there is a valid path from $v$ to $v$, and if a sequence of prefixes $a_1, a_2, ..., a_k$ is a valid path, then so are each of its prefixes $a_1, a_2, ..., a_j$, where $1 \leq j < k$. We say that an arc is on a valid path if it is part of the arc sequence for that path.

The significance of a valid path from $v_0$ to $v_k$ is that an instance of module $v_0$ contains an instance of $v_k$ as a submodule because of that valid path. In particular, $v_0$ contains an instance of $v_1$, which in turn contains an instance of $v_2$, etc. The valid path corresponds to this sequence of nested module instances.

**Definition.** Given a VDAG $\beta$, the exploded forest generated by $\beta$ is a forest $F$ such that there is a distinct vertex for each distinct generating path in $\beta$. The tag on the vertex corresponding to such a path is the same as the tag of the last vertex in the path. For each source vertex in $\beta$, there is a tree root in forest $F$. For each generating path $a_1, a_2, ..., a_k$ having at least one arc, the tree vertex corresponding to the generating path $a_1, a_2, ..., a_k$ is the parent of the tree vertex corresponding to $a_1, ..., a_k$. The stamp on the tree arc between this parent and child is the same as the stamp of $a_k$.

Given a VDAG $\beta$, we will denote the exploded forest generated by $\beta$ as $F_\beta$.

Given a VDAG, the exploding forest generated from the VDAG is a set of design trees. The relationship between a given VDAG and this set of design trees constitutes the meaning of the VDAG; the purpose of the VDAG is to represent the set of design trees that it generates.

Given a VDAG, the exploded forest generated by the VDAG can be constructed by vertex splitting on the VDAG, as described by the following algorithm. Each source in the given VDAG $\beta$ becomes the root of a tree in the exploded forest $F_\beta$.

**Algorithm.** Given a VDAG $\beta$, the exploded forest $F$ generated by $\beta$ can be obtained from $\beta$ by repeatedly splitting each vertex occurrence in the VDAG, in a top-down order, according to arcs entering that vertex and the labels on those arcs. Each source of $\beta$ is the root of a tree in the exploded forest. If vertex $v$ is currently under consideration and $a_1 = (u_1, v), a_2 = (u_2, v), ..., a_m = (u_m, v)$ are the current arcs entering $v$ then for each $i, 1 \leq i \leq m$, the current graph has a unique path from a source to $u_i$. For each $a_i$ if $l(a_i)$ contains some vertex on the unique path from a source to $u_i$, then a copy of $v$ is made a child of $u_i$. That copy also has a copy of each arc exiting from $v$. For purposes of matching labels further down the forest, a record is maintained that each copy of $v$ is a version of vertex $v$ from the initial VDAG.

As an example, given the VDAG shown in Fig 2.3(a), the results at some stages of the replacement (instantiation) process are shown in Fig 2.3(b), (c), (d) and (e). The exploded forest generated by the VDAG in Fig 2.3(a) is shown in Fig 2.3(e). Note that the dashed arrows shown in the figures are part of the exploded forest emerging from the computation, and thus have no labels on them.

As another example, for the VDAG in Fig 2.2(b), the exploded forest generated by this VDAG is shown in Fig 2.2(a).

Since sometimes in processing VDAGs, we do not need to know the sequence of arcs in all valid paths, but may only need to know whether there exists a valid path connecting a pair of specified endpoints, we define the functions reaching and passing. The purpose of introducing reaching and passing is to specify whether a module $B$ is used by (i.e., part of) some version of a larger module $A$, without specifying how it is materialized. Reaching and passing can be used, for instance, in simplification of VDAGs, i.e., making a given VDAG "smaller" by eliminating useless information.

**Definition.** In a given VDAG, a vertex $w$ reaches a vertex $v$ if there exists a valid path starting at $w$ and ending at $v$ in the VDAG. A vertex $w$ passes an arc $\alpha = (u, v)$ if $\alpha$ is on some valid path from $w$ to $v$. For each vertex $v$, let the set of vertices which $w$ reaches be $\text{reaching}(w)$, and the set of arcs which $w$ passes be $\text{passing}(w)$. Let $\text{reaching}$ be $\text{reaching}$ defined only on sources, that is, $\text{reaching}(v)$ is $\text{reaching}(v)$ if $v$ is a source and is undefined otherwise.

From the above definition of passes, a vertex $w$ passes an arc $\alpha = (u, v)$ if and only if there is a valid path from $w$ to $u$ and the intersection of $l(\alpha)$ and the set of vertices on this path is nonempty.

Consider the possibility that a vertex $x$ is not on any generating path ending at a vertex $x$, but it is in the label for an arc $(u, v)$ for some vertex $v$. In this case, we may say that $x$ is a useless element in the label. To capture this concept, we define validity as the follows.

**Definition.** An element $x$ of a label $l((u, v))$ is valid if there exists a generating path entering $u$ which passes through $x$. A label is valid if every element of the label is valid. A VDAG is valid if all of its labels are valid. A label containing no valid elements is a useless label. A non-source vertex that is not reached by any of the source vertices is an useless vertex.

Note that if a VDAG contains a useless vertex, the arcs entering it have useless labels, so the VDAG is invalid.
Also note that useless vertices and arcs with useless labels cannot be on any generating path.

We next show that for a given VDAG that is not valid, by the removal of invalid label elements, arcs with useless labels, and useless vertices, a valid VDAG can be obtained that generates the same forest.

**Lemma 3.1.** Given an invalid VDAG α, there exists a valid VDAG that generates the same forest as α generates.

**Proof.** Because α is invalid, α has some labels containing invalid label elements, and possibly some useless vertices. Since useless vertices and arcs with useless labels are not part of any generating path, they do not contribute to the forest obtained from the explosion of the VDAG. Similarly, in the explosion process, invalid label elements cannot be used to extend any generating path. Hence a valid VDAG is obtained from α by deletion of invalid label elements, arcs with useless labels, and useless vertices; this VDAG generates the same forest which α generates. □

In Section III, an algorithm SREACHING-VALIDITY is presented which computes the reaching relation from the source vertices, the reaching relation, and invalid label elements.

Since there is often the need to obtain one version out of a set of versions for a design module, the following definition is provided.

**Definition.** For a given VDAG \((G(V,A),t,s,t',s')\) and a source vertex \(s\), a single-sourced VDAG rooted by \(s\) is \((G'(V',A'),t',s',t')\) where \(V'\) is the set of vertices which \(s\) reaches and \(A'\) is the set of arcs which \(s\) passes; \(t',s',\) and \(t'\) are \(t, s,\) and \(t\), respectively, with elements involving vertices or arcs not in \(V'\) and \(A'\) removed.

For example, for the VDAG shown in Fig 1.6, the single-sourced VDAGs rooted by \(V1\) and \(V2\) are shown in Fig 2.4(a) and (b), respectively.

In the remainder of this paper, by “the VDAG induced by vertex \(s\)” we mean \((G'(V',A'),t',s',t')\). In Section III, an algorithm EXTRACT is described which computes the VDAG induced by \(s\) for a given VDAG and a source vertex \(s\).

### III. Some Properties and Algorithms

**Lemma 3.1.** *Reaching* is a partial order, i.e., it is reflexive, transitive, and antisymmetric. □

**Lemma 3.2.** *(pseudo transitivity of passing)* If \(u\) reaches \(v\) and \(v\) passes an arc \(a\), then \(u\) passes \(a\). □

A problem germane to module containment can now be formulated as follows.

**Valid Path Problem (VPP).**

**Instance:** A VDAG and two vertices \(u\) and \(v\).

**Question:** Is there a valid path starting at \(u\) and ending at \(v\) in the given VDAG?

This problem can also be stated as: Does module \(u\) contain some instance of module \(v\) as a submodule? Unfortunately, this problem may not be efficiently solvable, as suggested by the following theorem.

**Theorem 3.3.** VPP is NP-complete in the strong sense.

**Proof.** (Sketch) Membership in NP is because a valid path from \(u\) to \(v\) can be guessed nondeterministically and verified in polynomial time. The NP-hardness proof is by a reduction from SAT (i.e., the problem of determining whether a given conjunctive normal form Boolean formula is satisfiable), which is well known to be NP-complete [GJ]. For a given formula to be tested for satisfiability, with variables \(x_1, x_2, \ldots, x_n\) and clauses \(c_1, c_2, \ldots, c_m\) we construct a VPP instance in the form as shown in Fig 3.1. In the construction, label \(l_i\) contains exactly the elements in clause \(c_i, 1 \leq i \leq m\). The tags on vertices and the stamps on arcs are assigned arbitrary unique values. Then, there is a valid path from \(a_0\) to \(b_m\) if and only if the given formula is satisfiable. □

**Corollary 3.4.** The following problems are NP-complete in the strong sense: 1) computing the binary relation \(\text{reaches}\); 2) computing the binary relation \(\text{passes}\); and 3) determining the validity of a given element in a label. □

We give an algorithm SREACHING-VALIDITY, as shown in Fig 3.2, to compute the reaching relation from the source vertices and a list of all invalid label elements. Note that for purposes of computing the reaching sets and invalid label elements, \(G\) can be converted from multigraph to a graph by consolidating each set of arcs that have the same endpoints. For more details of the algorithm SREACHING-VALIDITY, see [YR2].

Define two valid VDAG paths to be graph equivalent if they consist of the same sequence of vertices. The algorithm processes all equivalence classes of generating paths, one equivalence class at a time. The algorithm uses a stack to store the sequence of vertices in the equivalence class currently being processed. Since a generating path contains each VDAG vertex at most once, the size of the stack is linearly bounded by the number of vertices. The size of \(\text{reach}\) is bounded by \(O(|V|^2)\). The algorithm keeps track of which label elements are valid, but the space required for this is linear in the size of the VDAG description. Hence the space required is at most quadratic in the number of vertices, and linear in the other parameters of the VDAG description. The time cost can be, however, exponential in the number of vertices, since the number of generating paths in a given VDAG can be exponential in the numbers of vertices, but the time cost cannot exceed the time required to generate the VDAG's forest. Due to the NP-completeness of reaching, we do not attempt to find a polynomial-time algorithm.

**Lemma 3.5.** SREACHING-VALIDITY correctly computes the reaching relation and invalid labels on given VDAGs.
**Definition.** Two VDAGs $\alpha$ and $\beta$ are equivalent, denoted $\alpha \equiv \beta$, if $F_\alpha \equiv F_\beta$.

For instance, in Figure 4.2(a), VDAG A contains VDAG B but not vice versa. In Figure 4.2(b), VDAG A and VDAG B are equivalent.

**Theorem 4.1.** Given two VDAGs $\alpha$ and $\beta$, the problem of deciding whether $\alpha \neq \beta$ is $NP$-complete.

**Proof.** (Sketch) The proof of membership of the problem in $NP$ is omitted. The $NP$-hardness proof is by a reduction from SAT [GJ]. Given a formula to be tested for satisfiability, we build $\alpha$ in the same way as that in the proof of Theorem 3.3, and let $\beta$ be $\alpha$ with the vertex $b_m$ and the arc from $b_m-1$ to $b_m$ removed. It is easy to see that $\alpha \neq \beta$ if and only if $a_0$ reaches $b_m$ in $\alpha$.

**Corollary 4.2.** Given two VDAGs $\alpha$ and $\beta$, the problem of deciding whether $\alpha \equiv \beta$ is co-$NP$-hard.

We next consider the solution for the equivalence problems of VDAGs. A straightforward method, which comes directly from the definition of the VDAG equivalence, is to first build the exploded forests from the given VDAGs, and then employ EqVDAG to decide if the two VDAGs are equivalent. Since the size of the exploded forests may be exponentially larger than their VDAG counterparts, this method may cost exponentially both in time and in space.

Our next algorithm employs the concept of equivalent generating paths.

**Definition.** Two generating paths, $p_1 = a_1, \ldots, a_m,$ and $p_2 = b_1, \ldots, b_n$, are equivalent, denoted $p_1 \equiv p_2$, if the following three conditions are simultaneously satisfied:

1) $m = n$;
2) if $(u_{i-1}, u_i)$ and $(v_{i-1}, v_i)$, then $t(u_{i-1}) = t(v_{i-1})$ and $f(u_i) = f(v_i)$ for all $i, 1 \leq i \leq n$; and
3) the stamp on $a_i$ is the same as the stamp on $b_1$ for all $i, 1 \leq i \leq n$.

The algorithm EqVDAG, shown in Fig 4.3, determines whether two given VDAGs are equivalent. The running time of the algorithm may be exponential in terms of the input size, but it uses polynomial space. The algorithm uses two stacks, $S_\alpha$ and $S_\beta$, for the given two VDAGs respectively. For each VDAG, the corresponding stack is used to store a generating path, one at a time.

V. Automatic Simplification

In this section we consider the problem of simplifying a VDAG, i.e., given a VDAG, producing an equivalent VDAG that is, in some sense, smaller.

**Definition.** A VDAG $\alpha$ is smaller than VDAG $\beta$ if $\alpha$ has fewer arcs than $\beta$, or if $\alpha$ and $\beta$ have the same number of arcs and $\alpha$ has a smaller total label size than $\beta$. A VDAG $\alpha$ is minimum-sized if there is no equivalent VDAG $\beta$ such that $\beta$ is “smaller”.
Since the size of a VDAG is finite, for each VDAG there must exist some equivalent minimum-sized VDAG.

**Theorem 5.1.** Given a VDAG , it is NP-hard to determine whether is minimum-sized.

Proof. (Sketch) By a reduction similar to that in the proof of Theorem 3.3.

**Corollary 5.2.** Given a VDAG , finding a VDAG such that is minimum-sized is NP-hard.

**Theorem 5.3.** Given a VDAG , it is NP-hard to find a VDAG such that has a minimum number of arcs.

Proof. (Sketch) By a reduction from the graph coloring problem [GJ].

In order to reduce the label size while preserving equivalence, we define three operations on VDAGs. The operations involve a special kind of cut set that separates the set of sources from a non-source vertex, with respect to the generating paths entering that vertex, i.e., it "cuts" these generating paths.

**Definition.** Given a VDAG , a GP-cut-set separating vertex from a set of source vertices is a set of vertices such that removing the members of and their associated arcs from makes unreachable from any member of .

\[ \text{Op}_1: \text{If } x \in \{l(u,v)\}, y \in \{l(u,v)\}, \text{and every generating } \]

\[ \text{path that traverses both } x \text{ and } u \text{ also traverses } y, \text{ then delete } x \text{ from } \{l(u,v)\}. \]

\[ \text{Op}_2: \text{If } \{l(u,v)\} \text{ contains a GP-cut-set separating } x \text{ from the set of source vertices that reach } u, \text{ then let } \{l(u,v)\} = \{u\}. \]

\[ \text{Op}_3: \text{If } \{l(u,v)\} \text{ properly contains a GP-cut-set separating } u \text{ from the set of source vertices which pass } \]

\[ \text{arc } (u,v), \text{ then select such a subset of } \{l(u,v)\} \text{ of minimum cardinality, and delete the other } \]

\[ \text{elements of } \{l(u,v)\}. \]

**Lemma 5.4.** \( \text{Op}_1, \text{Op}_2, \text{ and } \text{Op}_3 \) are equivalence preserving operations on valid VDAGs.

**Theorem 5.5.** Given a VDAG, TRIMMER reduces the size of the VDAG.

**Proof.** Steps 2, 3, and 4 preserve equivalence from the proof of Lemma 2.1. By Lemma 5.4, Step 5 preserves equivalence. Since TRIMMER might delete some arcs or vertices and/or replace labels by smaller ones, but never add anything to the VDAG, the resulting VDAG is either unchanged or smaller.

**VI. Conclusion**

The VDAG model proposed in this work can be used to concisely represent hierarchically specified design data in a way that supports flexible design alternatives in engineering design database systems. A variety of computational problems involving VDAGs are shown to be NP-hard in terms of the size of VDAG. Algorithms are provided to determine whether one module is contained in a larger module, to extract a version from a given VDAG, to test the equivalence of two VDAGs, and to reduce the size of a VDAG.

**REFERENCES**


34


Algorithm SREACHING-VALIDITY
Input (G(V,A), t, st, l), where V = \{v_1, v_2, \ldots, v_k\}
Output searching and a list of all invalid label elements
(for each source v \in V, searching(v) is a set of vertices in V. When the algorithm terminates, searching(v) is the set of all vertices that v reaches).
INV is a set of pairs from A x V, each of which is an arc and an invalid element in the arc's label.)
begin (sketch)
1. Convert G from a multigraph to a graph by merging all arcs having the same end-points. The label on each arc resulting from a merger is the union of the labels of the arcs that participated in the merger.
2. For each source vertex u, extend all valid paths which start at u; add the vertices reached by u into searching(u) and mark the label elements used in extending the valid paths;
3. Let INV be the set of (arc, label-element) pairs where the label-element in the label of an arc is never marked.
end.

Fig 3.2.

Algorithm EXTRACT
Input (G(V,A), t, st, l), and a source s \in V
Output (G'(V,A'), t', st', l'), a single-source dag induced by s
begin
let V' be the set of all vertices reached by s;
let A' be the set of all arcs a such that s passes a;
for each \( v \in V' \) do
\( r'(v) \leftarrow r(v); \)
for each \( a \in A' \) do
begin
\( st'(a) \leftarrow st(a); \)
\( l'(a) \leftarrow l(a) \cap V'; \)
end.
end.

Fig 3.4.

Algorithm PASSING
Input (G(V,A), t, st, l), where V = \{v_1, v_2, \ldots, v_k\}
Output passing relation
(for each v \in V, passing(v) is a set of arcs. When the algorithm terminates, passing(v) is the set of all arcs which v passes.)
begin
for each vertex v, extend all valid paths which start at v, and include all of the arcs occurred in those valid paths into passing(v)
end.

Fig 3.3.

Algorithm EqFOREST
Input two exploded forests F = (G(V, A), t, st) and F' = (G(V', A'), t', st')
Task determine whether F and F' be H;
for h := 0 to H do
for each pair of vertices u \in V and v \in V' with height h, do
if \( r(u) = r'(v) \) and \( k = 0 \), or \( r(u) = r'(v) \) and there exist a one-to-one mapping \( f : \{1, \ldots, k\} \rightarrow \{1, \ldots, k\} \) where \( eq(u, v_{f(i)}) \)\ is true, 1 \leq i \leq k, then set eq(u, v) to true, and set eq(u, v) to false otherwise;
if for each source s in V, there is a source s' in V' of the same height such that eq(s,s') = true, and vice versa, then return("F and F' equivalent")
else return("F and F' not equivalent")
end.

Fig 4.1.

Algorithm EqVDAG
Input VDAGs \( (G_1(V_1,A_1), t_1, s_1, l_1) \) and \( (G_2(V_2,A_2), t_2, s_2, l_2) \)
Task determine if the two VDAGs are equivalent
begin
1. if there is a source u in one VDAG that cannot be paired with any source v in the other VDAG, such that u and v have the same tags, then return("inequivalent");
2. for each such source pairs u, v found in Step 1 do
   if for some equivalence class of generating paths starting at u in one VDAG, there is no corresponding equivalence class of generating paths starting at v in the other VDAG with the same cardinality, then return("inequivalent");
3. return("equivalent")
end.

Fig 4.2.

Algorithm TRIMMER
Input a VDAG \( \beta \)
Output a possibly smaller VDAG equivalent to \( \beta \)
begin
1. SREACHING-VALIDITY(\( \beta \));
2. delete all useless vertices in \( \beta \) and their associated arcs;
3. delete all arcs with useless labels;
4. for each remaining label, delete invalid elements from it;
5. repeatedly apply Op_1, Op_2, and Op_3, until there is no further change;
end.