Using Tube Graphs to Model Architectural Designs of Software Systems

Richard C. Holt and Spiros Mancoridis
Department of Computer Science
University of Toronto
Canada

Abstract

A tube graph is a mathematical entity that can be used for modelling architectural designs of software systems. It consists of a tree (representing containment) with a set of edges called tubes (representing dependencies) between the tree's vertices (representing components). Tubes, for example, can model the *import* relation at the module level and the *use* dependency relation at the subsystem level. Well-formedness constraints are imposed on tube edges to define restricted tube graphs that model particular styles of architectural designs. In this paper, the well-formedness constraints define tube graphs that model SIL language architectural designs. Different mathematical formulations of these well-formed tube graphs are given and proven to be equivalent. Algorithms for manipulating tube graphs, for possible use in a software development environment, are also given and their complexities are shown to be modest in most cases.

1 Introduction

An *architectural design* specifies a software system's structure in terms of its components and their interconnections. Software architecture, as described by Perry and Wolf [14] and Garlan and Shaw [7], is a broader term that encompasses architectural design as well as issues such as global control structures, protocols for communication, synchronization and data access, assignment of functionality to design elements, physical distribution, performance, and so on. Earlier textual notations for specifying architectural designs, called Module Interconnection Languages (MILs), include those by DeRemer and Kron [5], Cooprider [3], and Tichy [16]. More recent visual ones include those by Schwanke et al. [15], Penny et al. [13, 10], and the authors [9]. Although these notations differ in their details, they all feature mechanisms for aggregating related software components (procedures, modules, classes, files) into composite components called subsystems. These subsystems can contain other subsystems to form a containment hierarchy.[1] Programming languages, such as Modula-2, Object-Oriented Turing [8], C, and C++, provide mechanisms for specifying connections between atomic components via relations such as *import*, *include* and *inherit*. Similarly, architectural design notations should provide mechanisms for specifying, either explicitly or implicitly, such connections involving subsystems.

Recently, Dean and Lamb used a set theory based framework to formally describe existing MILs [4]. Our approach is based a mathematical entity called a tube graph.[2] A tube graph consists of a tree and an extra set of edges, called tubes, between the vertices of the tree. These graphs become useful for modelling architectural designs when well-formedness restrictions are imposed on their tubes to express the connectivity constraints between the software components. The tree represents subsystem containment. For example, the root of the tree could represent a large software system, the internal vertices could represent subsystems and the leaf vertices could represent source code (modules). The tubes represent dependencies. For example, a tube between two leaf vertices may represent a procedure call by one module to another.

Figure 1 shows the structure of system S containing two subsystems A and C. Subsystem A contains module B and subsystem C contains modules D and E. The edges from B to D and E represent inter-module dependencies. For example, a procedure in B may invoke procedures in D and E. The person in charge of the architectural design of this system might wish to permit modules in A to depend on modules in C, but not vice versa. This permission could be represented by a large arrow from A to C, which is drawn in the middle of the diagram of Figure 1 as a hollow tube, through which are routed the edges from B to D and E. The bottom diagram shows the tube graph that corresponds to the design in the middle of the diagram, where containment is shown by solid edges, and two dependencies B to D and B to E, as well as the tube A to C are shown as dotted edges. All of these dotted edges are tubes.

As the example illustrates, we use the term "tube" for the connecting edges to indicate that when such edges occur higher in the tree, they represent permissions or routes that must be used by lower-level edges. When all of the tube edges obey these permissions, we say that the tube graph is well-formed. During a system's development, its designers and programmers may add or delete dependencies, which implies changes to the corresponding tube graph. As a result, the graph may become ill-formed, and these people can use the algorithms, presented in Section 6, to efficiently manipulate tube graphs and check for well-formedness.

The rest of this paper is organized as follows: In Section 2, we describe how tube graphs fit into a software development environment framework that supports the specification and visualization of architectural designs. In Section 3, we formally define tube graphs. In Section 4, we define well-formed tube graphs that are useful for modelling architectural designs of a version of a language called SIL [9]. In Section 5, we give an alternative and equivalent definition of well-formed tube graphs using a graph grammar [1]. In Section 6,

[1]Some MILs might allow overlapping subsystems, where the containment hierarchy is a directed-acyclic graph instead of a tree. Such structures cannot be modelled using tube graphs.

[2]It is likely that tube graphs, being general mathematic entities, could have other applications as well.

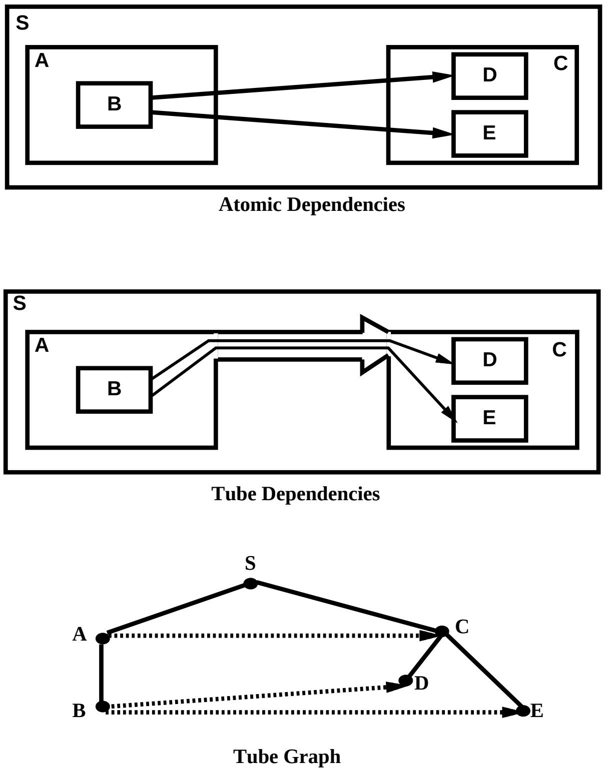

Figure 1: **Using Tube Graphs to Model the Structure of a System**

we give algorithms and their complexity for manipulating tube graphs. Finally, in Section 7, we summarize the paper and describe possible directions for further research.

2 Using Tube Graphs in a Software Development Environment

Tube graphs fit into a general framework[3] [9] (illustrated in Figure 2) for software development environments that support the specification and visualization of architectural designs. Within this framework, the designer can draw architectural designs using diagrams such as those in Figure 1. The way the diagrams can be drawn — what kinds of colours of boxes and edges are available, whether boxes can overlap and so on — is determined by the syntax (top right oval of Figure 2) and semantics (bottom right) of the drawing system. The drawing system may, in fact, be used for applications other than software design. Once a diagram has been drawn, it can be interpreted as a program in a language such as SIL [9]. One way to do this is by having the drawing system emit a SIL program that specifies the components of the system and their dependencies. This program can be interpreted

[3]This framework focuses on those components of a software development environment that are related to architectural design.

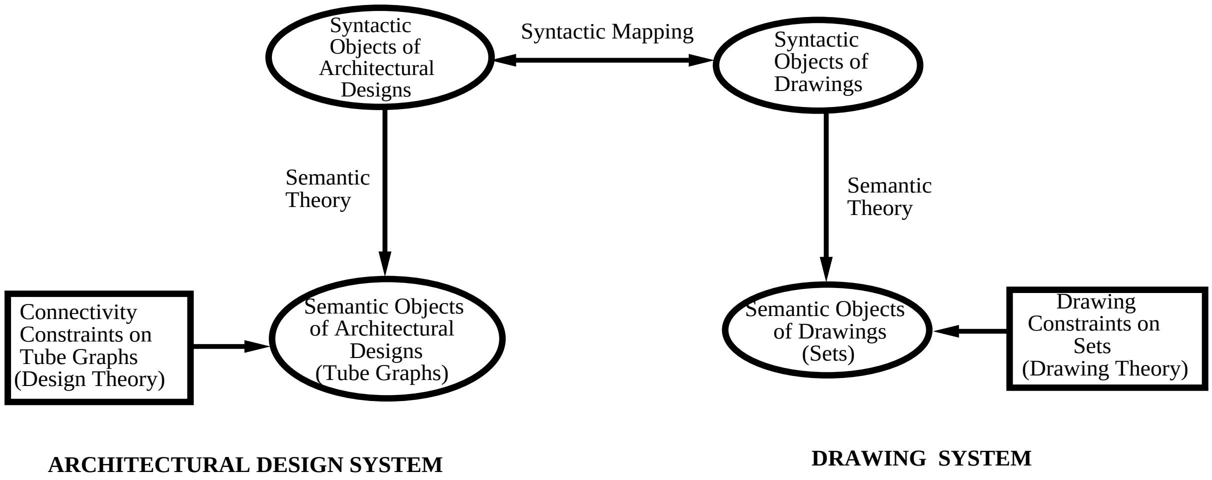

Figure 2: **Overview of the Framework**

by the SIL system as a tube graph. In other words, the syntax of a SIL specification (top left oval) can be parsed to produce its semantics[4] (bottom left oval). The resulting tube graph (the design semantics) can be checked for well-formedness, using the algorithms in this paper.

A possible alternative activity is that the designer could modify the tube graph, using our algorithms, request that a new SIL program be generated and translated to a diagram that can be displayed by the drawing system.

To summarize, we separate the various parts of a software development environment that supports the specification and visualization of architectural designs. We are primarily interested in models that represent the graph-theoretic structure of such designs (bottom left of Figure 2). This area of interest requires attention because we need methods of capturing the essence of software designs, distinct from the systems for drawing them and distinct from particular development projects.

We present tube graphs as a model for architectural designs. The model is simple — it has only one kind of tube — and its formalization is straightforward. We believe that the particular model we present here will prove to be of practical use and also that the general approach to this area will help clarify how approaches to architectural design can be modelled and integrated into software development environments.

We now proceed with the formal definition of tube graphs.

[4]At a separate level, not addressed in this paper, one must also give the semantics of the program in terms of the effect of its execution.

3 The Definition of a Tube Graph

A *tube graph* consists of a set of vertices (V) and two sets of edges called the *containment* (C) and the *tube* (T) edges[5], respectively. A tube graph can be thought of as any containment tree augmented with tube edges (tubes). Later, we impose well-formedness constraints to limit the allowed usage of tubes. Following is the formal definition of containment trees and tube graphs:

Definition 1: Containment Tree

Given a set of vertices V and the containment relation $C \subseteq V \times V$, we say (V, C) is a tree if and only if it has one root, each vertex has a single parent, and there are no cycles:

$$Root(\mathrm{x}) \equiv \neg \exists\ \mathrm{p} \in V \bullet \mathrm{p}C\mathrm{x}$$

$$\begin{array}{lll} Tree(V,C) \equiv & (\exists!\ \mathrm{r} \in V \bullet Root(\mathrm{r})) \wedge & \text{(unique root)} \\ & (\forall\ \mathrm{x} \in V \bullet \neg Root(\mathrm{x}) \Rightarrow (\exists!\ \mathrm{p} \in V \bullet \mathrm{p}C\mathrm{x})) \wedge & \text{(single parent)} \\ & (\forall\ \mathrm{p,x} \in V \bullet \mathrm{p}C^{+}\mathrm{x} \Rightarrow \mathrm{p} \neq \mathrm{x}) & \text{(no cycles)} \end{array}$$

where, C^{+} is the transitive closure of relation C.

□

Definition 2: Tube Graph

We say that TG=(V, C, T) is a tube graph if and only if (V, C) is a tree:

$$\begin{array}{ll} V, \text{ where } |V| \geq 1 & \text{(set of at least one vertices)} \\ C \subseteq V \times V & \text{(containment relation)} \\ T \subseteq V \times V & \text{(tube relation)} \end{array}$$

TG=(V, C, T) is a tube graph if and only if $Tree(V, C)$

□

4 Restricting Tube Edges

The preceding section defined a tube graph to be a containment tree with tubes between its vertices. By imposing well-formedness constraints on the tube relation we can obtain restricted tube graphs that can model particular styles of architectural designs. For example, designers of layered systems we may want to restrict the language to only allow connections between components on the same level in the containment tree. We will define a set of well-formedness rules that captures the connectivity constraints of a version of an architectural design language called SIL.

[5]Other applications may require various kinds of tube edges. To make the definition easier to understand, we make the simplifying assumption that there is only one kind of tube.

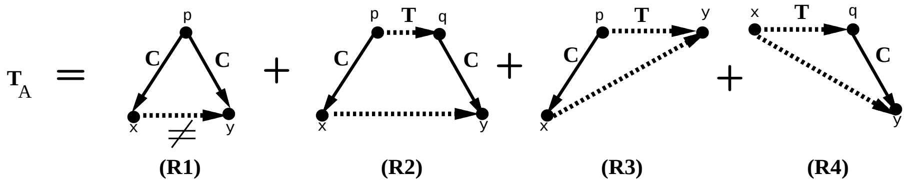

Figure 3: **A Visual Representation of Allowed Tubes,** T_A

4.1 Well-Formed Tube Graphs

We will consider that a tube xTy is well-formed if it satisfies one of these rules:

(R1) Sibling Rule: x and y are proper ($x{\neq}y$) siblings, or
(R2) Parent-Tube-Child Rule: x's parent has a tube to y's parent, or
(R3) Parent-Tube Rule: x's parent has a tube to y, or
(R4) Tube-Child Rule: x has a tube to y's parent.

Figure 3 gives a visual representation of the set of allowed tubes using a notation similar to GraphLog [2, 12]. The labelled edges (C and T) represent edges in a tube graph. The unlabelled dotted edges represent allowed tubes. Note that the sibling vertices for rule (R1) must be distinct. We denote the set of allowed tubes of a tube graph $TG{=}(V,C,T)$ as $T_A(TG)$. When the particular tube graph TG is obvious we write T_A instead of $T_A(TG)$. Following is the formal definition of the set of allowed tubes:

Definition 3: Allowed Tubes
Given a tube graph $TG{=}(V, C, T)$, its set of allowed tubes (T_A) is:

$T_A \equiv$ { (x,y) | WFTube(x,y) }

where

$$
\begin{array}{llr}
\text{WFTube(x,y)} \equiv \exists \text{ p,q} \in V \bullet & (\text{x}{\neq}\text{y} \wedge \text{p}C\text{x} \wedge \text{p}C\text{y}) \vee & \textbf{(R1)} \\
& (\text{p}C\text{x} \wedge \text{q}C\text{y} \wedge \text{p}T\text{q}) \vee & \textbf{(R2)} \\
& (\text{p}C\text{x} \wedge \text{p}T\text{y}) \vee & \textbf{(R3)} \\
& (\text{q}C\text{y} \wedge \text{x}T\text{q}) & \textbf{(R4)}
\end{array}
$$

□

Note that the right hand side of the definition of *WFTube* corresponds directly to the graphs in Figure 3.

Definition 4: Well-Formed Tube Graph

A tube graph $TG{=}(V,C,T)$ is well-formed if and only if its set of tubes T is a subset of its set of allowed tubes T_A:

$WF(TG) \equiv T \subseteq T_A$

which can be rewritten as:

$WF(TG) \equiv \forall$ (x,y) $\in T \bullet$ WFTube(x,y)

□

Definition 5: All Well-Formed Tube Graphs

We define τ as the set of all well-formed tube graphs:

$\tau \equiv \{ \ TG \mid WF(TG) \ \}$

□

In the following section, we define a graph grammar that generates exactly the set of all well-formed tube graphs.

5 A Graph Grammar for Well-Formed Tube Graphs

In this section, we give an alternative mathematical formulation of well-formed tube graphs based on graph grammars [1]. This formulation implies a construction algorithm for generating exactly the set of all well-formed tube graphs.[6] It differs from the logic predicate formulation in Section 4, which declaratively specifies the necessary conditions for a tube graph to be well-formed.

A *context-sensitive graph grammar* $CSG{=}(N, M, P, S)$ [11, 6] consists of a set N of non-terminal edges, a set M of terminal edges, a set P of context-sensitive productions, and the initial non-terminal edge $S{\in}N$. The left-hand side of a context-sensitive production is any graph that contains at least one non-terminal edge from N, and possibly terminal edges from M. The right-hand side of a context-sensitive production may be any graph of non-terminals from N and terminals from M, provided that the number of (terminal and non-terminal) edges on the right-hand side graph is not less than that of its respective left-hand side graph. A graph is terminal if all of its edges are terminal.

Given a context-sensitive grammar *CSG*, all terminal graphs Γ that are derivable from the initial edge S comprise the language *L(CSG)* generated by *CSG*:

$$L(CSG) \equiv \{\Gamma \mid S \stackrel{*}{\Rightarrow} \Gamma, \text{ and } \Gamma \text{ is terminal}\}$$

Where $\stackrel{*}{\Rightarrow}$ denotes the transitive closure of direct derivation.

The complete context-sensitive graph grammar G for well-formed tube graphs is shown in Figure 4. Vertices in the graph grammar are distinct. Note that production P0 generates the containment tree of the tube graph, while productions P1-P4 correspond to the four well-formedness rules. Production 5 simply makes non-terminal containment edges terminal. We need two kinds of non-terminals for containment edges because ill-formed tube graphs can be generated by a grammar that allows containment edges to be expanded after they have been connected to tube edges.

[6] Since we are using edge replacement graph grammars, grammar G cannot generate a single vertex tube graph.

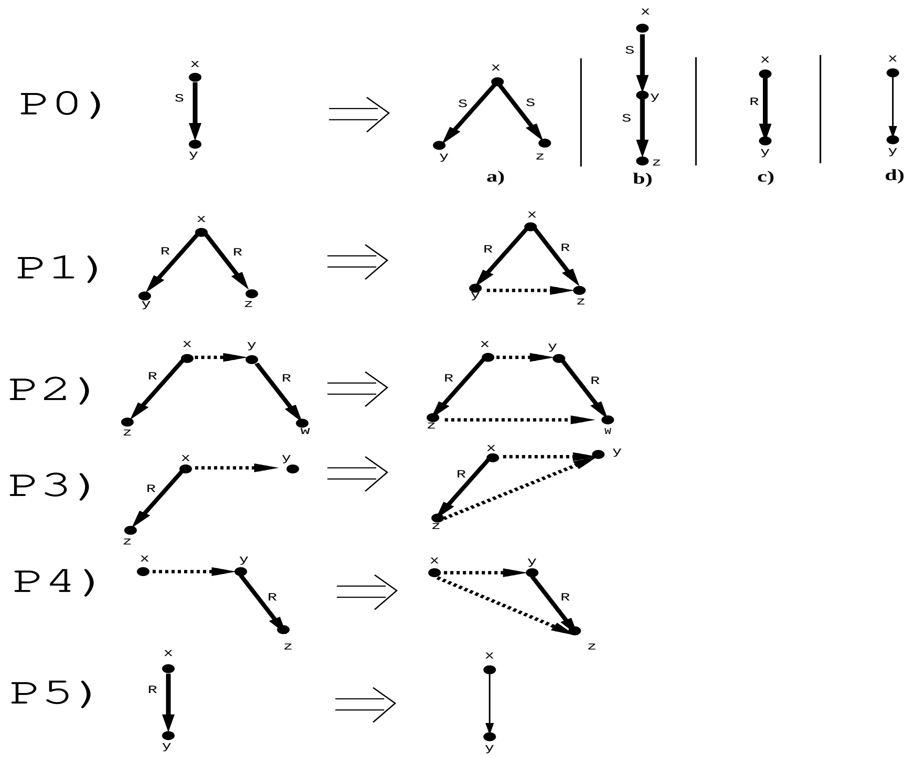

Figure 4: **Context-Sensitive Grammar G for Well-Formed Tube Graphs:** The grammar has six productions. The initial non-terminal edge (labelled S) is the left-hand side of production 0). There are two kinds of non-terminal edges, S and R, that both represent containment. Note that before any tube edges can be generated, their containment edges must be transformed into R edges using production 1c). All terminal edges are unlabelled. Terminal containment edges are thin and solid while tube edges are dotted.

Next, we present proofs that grammar G generates all well-formed tube graphs and nothing more, that is $WF(TG) \Leftrightarrow TG \in L(G)$. Underlying these proofs is the assumption of the canonical order of construction dictated by the graph grammar.

Theorem 1: Any Well-Formed Tube Graph is in L(G)

We will prove that any well-formed tube graph TG=(V,C,T) can be generated using the context-sensitive grammar G:

$$\tau \subseteq L(G)$$

Let $d(v,R)$ be the out-degree of vertex $v \in V$ with respect to relation R; hence, $d(v,C)$ is the out-degree of vertex v with respect to the containment relation (C) and $d(v,T)$ is the out-degree of vertex v with respect to the tube relation (T).

Definition 6: Out-Degree of a Vertex

$d(v,R) \equiv |\{(v,\mathrm{y}) \mid vR\mathrm{y}\}|$

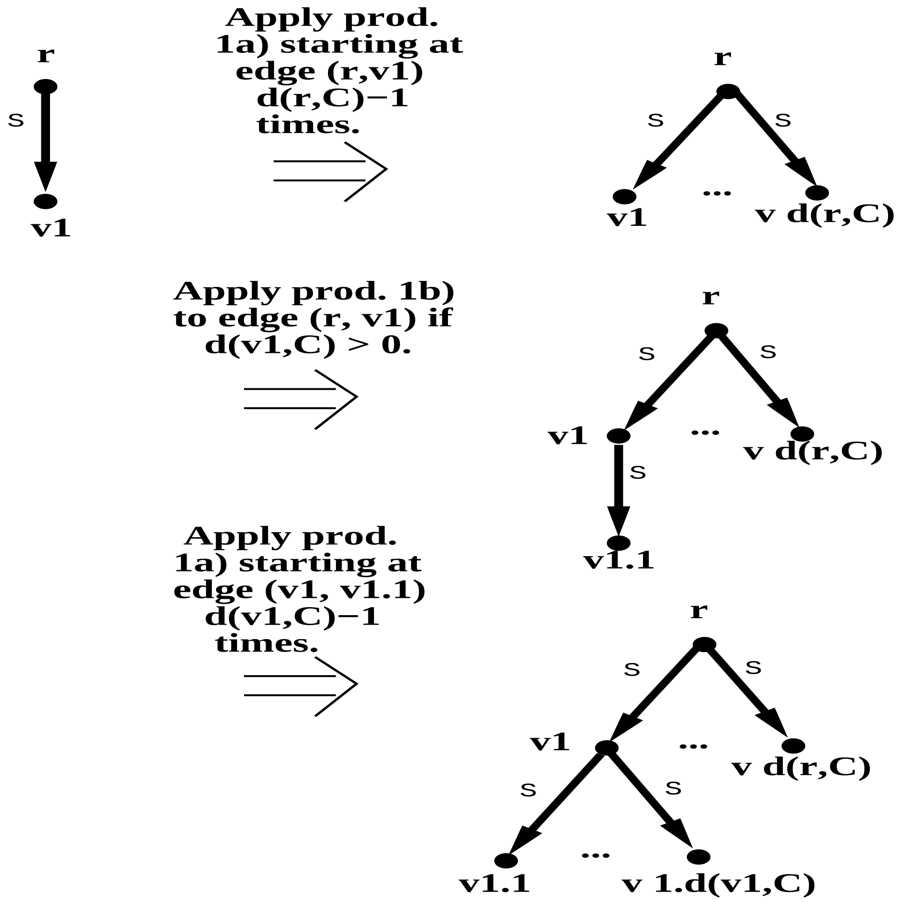

Figure 5: **Deriving the Containment Tree of a Tube Graph**

□

We first show how the containment tree can be constructed and then show how to augment this tree with well-formed tube edges.

Starting from the root vertex r, construct its children by applying the 1a) production $d(r,C)$–1 times. Then apply production 1b) to the edge from the root to each child vertex v_i if $d(v_i,C)>0$. Recursively repeat the steps performed on the root to v_i, until all containment edges are generated. Figure 5 illustrates these steps, which generate the containment tree.

To show how the tubes can be constructed, we first define the *level* of a vertex in a containment tree:

Definition 7: Level of a Vertex

A vertex $v \in V$, in a tube graph $TG=(V,C,T)$, is at level n if it has n proper ancestors in the containment tree.

$$level(v) \equiv |\{\mathrm{p} \mid \mathrm{p}C^{+}v\}|$$

□

We now augment the containment tree with tube edges. Before doing so we apply production 1c) so all S edges become R edges. Since the root vertex r of a well-formed tube graph cannot have any tubes (Theorem 3), we start with the $d(r,C)$ children of r, which are all the vertices on the first level of the containment tree. For each level k of the tree, in order 1 to 2 to 3, and so on, do the following: For each vertex v_i of level k, generate all of its $d(v_i,T)$ tubes. Start by selecting vertex v_j such that the tube v_iTv_j has not been generated and $level(v_j)$–$level(v_i)$ is minimum. Continue until all $d(v_i,T)$ tubes have been generated. Since the tube graph is well-formed, each tube v_iTv_j must fall into one of the following categories:

(R1): v_i and v_j have the same parent (use production 1), or
(R2): there exists a tube between the parents of v_i and v_j (use production 2), or
(R3): there exists a tube between the parent of v_i and v_j (use production 3), or
(R4): there exists a tube between v_i and the parent of v_j (use production 4).

By applying the appropriate production (indicated in parentheses), we can generate any well-formed tube v_iTv_j. After all the tubes in the graph have been generated, apply production 5) to all R edges to make them terminal. This proves that $\tau \subseteq L(G)$.

□

Theorem 2: Any Graph in L(G) is a Well-Formed Tube Graph

We will prove that any graph Γ=(V, C, T) constructed using grammar G is a well-formed tube graph, that is:

$$L(G) \subseteq \tau$$

First, we prove that $\Gamma \in L(G) \Rightarrow Tree$=$(V,C)$. We use the property of trees that they are connected and their number of edges is one less than their number of vertices.

The only productions of G involved in building the containment tree are 1a), 1b), 1c) and 1d). Since none of these productions generates unconnected vertices, the resulting graph is connected. Also, because we start with a single edge connecting two vertices, if we use productions 1a) or 1b), we introduce a single new vertex and a single new edge. Productions 1c) and 1d) preserve the number of vertices and edges. Hence, the number of edges in the graph is always one less than the number of vertices. Therefore,

$$\Gamma \in L(G) \Rightarrow Tree{=}(V,C)$$

Next, we prove that $\Gamma \in L(G) \Rightarrow WF(\Gamma)$. We know that:

$$\begin{aligned}
&\Gamma \in L(G) \Rightarrow WF(\Gamma)\\
= &\Gamma \in L(G) \Rightarrow \forall\ (\mathrm{x,y}) \in T \bullet \mathrm{WFTube(x,y)}\\
= &\Gamma \in L(G) \Rightarrow \forall\ (\mathrm{x,y}) \in T \bullet (\mathrm{x} \neq \mathrm{y} \wedge \mathrm{p}C\mathrm{x} \wedge \mathrm{p}C\mathrm{y}) \vee\\
&\qquad (\mathrm{p}C\mathrm{x} \wedge \mathrm{q}C\mathrm{y} \wedge \mathrm{p}T\mathrm{q}) \vee\\
&\qquad (\mathrm{p}C\mathrm{x} \wedge \mathrm{p}T\mathrm{y}) \vee\\
&\qquad (\mathrm{q}C\mathrm{y} \wedge \mathrm{x}T\mathrm{q})
\end{aligned}$$

Since productions 1-4) are the only productions with tubes in them, all tubes xTy generated by grammar G are derived from them. If production 1) is used, x and y have the same parent (Sibling Rule). If 2) is used, the parents of x and y have a tube between them (Parent-Tube-Child Rule). If 3) is used, there is a tube between the parent of x and y (Parent-Tube Rule). If 4) is used, there is a tube between x and the parent of y (Tube-Child Rule). In any case, the derived tubes satisfy the well-formedness rules. This proves that $L(G) \subseteq \tau$, and both Theorems 1 and 2 prove that $\tau \equiv L(G)$.

□

Theorem 3: In a well-formed tube graph there are no tubes between a vertex and its ancestors or descendants

The theorem states that in a well-formed tube graph TG=(V, C, T), if a tube xTy is well-formed x cannot be an ancestor of y, and vice versa. This statement is formulated in logic as follows:

$$WF(TG) \Rightarrow \forall\ (\mathrm{x,y}) \in T \bullet \neg\ \mathrm{x}C^{+}\mathrm{y} \wedge \neg\ \mathrm{y}C^{+}\mathrm{x}$$

This theorem may not be obvious from the definition of a well-formed tube graph, given in Section 4.1; namely, that there are no tubes between ancestors of descendants of a vertex, which also implies that are no tube to or from the root. Rules of this sort are common in MILs when specifying the dependencies between modules and subsystems, but would be relaxed if we were using tube graphs for specifying to dependencies between procedures, since a procedure may call (depend on) itself.

The proof of this theorem depends on the canonical order of construction dictated by the graph grammar. We use induction on the number of tubes of the well-formed tube graph TG.

Base Case

Let the number of tubes be one. Since TG is well-formed, the tube can only be between sibling vertices (otherwise the graph must contain additional tubes).

Inductive Step

Assume the theorem holds for any well-formed tube graph with n tubes. To add a tube xTy to any well-formed tube graph of n tubes, so that the resulting tube graph is also well-formed, the additional tube must satisfy one of the following conditions:

(R1): Vertices x and y are proper siblings.
In this case, trivially, x is not an ancestor or descendant of y.

(R2): There exists a tube between the parents of x and y.
By the induction hypothesis the parents are not descendants of one another. Therefore, x and y (their children) cannot be descendants of one another.

(R3): There exists a tube between the parent of x and y.
By the induction hypothesis, the parent of x and y are not descendants of one another. Therefore, x cannot be a descendant of y and vice versa.

(R4): There exists a tube between x and the parent of y.
By the induction hypothesis, x and the parent of y are not descendants of one another. Therefore, x cannot be a descendant of y and vice versa.

Therefore, the theorem that a well-formed tube graph does not have tubes between any vertex and its ancestors or descendants, holds for any well-formed tube graph.

□

Next, we present algorithms for manipulating tube graphs and checking for their well-formedness.

6 Operations on Tube Graphs

We now present algorithms for manipulating the tubes and vertices of well-formed tube graphs as well as for determining the well-formedness of general tube graphs. From a theoretical standpoint, these algorithms can be considered to be operations of a tube graph algebra. From a practical standpoint, they can be considered to be functions of a visual or textual editor that supports the manipulation of architectural designs based on tube graphs. When specifying the complexity of these operations (applied to a tube graph) we use the following definitions:

- |V| is the number of vertices,
- |T| is the number of tubes,
- *height* is the height of the containment tree, which, for the worst case tree, is |V|,
- *maxChildren* is the number of children of the vertex (or vertices) with the most children, which, for the worst case tree, is |V–1|,
- *maxTubeOut* is the number of outgoing tubes from the vertex (or vertices) with the most outgoing tubes, which, for the worst case tree, is |T|,
- *maxTubeIn* is the number of incoming tubes to the vertex (or vertices) with the most incoming tubes, which, for the worst case tree, is |T|,
- *maxChildTubes* is the total number of outgoing tubes from the children of the vertex (or vertices) whose children have the most outgoing tubes, which, for the worst case tree, is |T|.

The worst case complexity of each of the following algorithms depends on the complexity of the primitive operations, which are defined in Appendix A. The worst case complexity of all primitive operations is O(1), except for the *addChild* and *removeChild* operations which have worst case complexity $O(|V|\times height)$.

6.1 Operations on Tubes

We now present algorithms for manipulating tubes and checking the well-formedness of tube graphs. Note that the algorithms for adding *(addTube)* and deleting *(deleteTube)* tubes are primitive operations. All of the following algorithms assume that the graphs being manipulated are tube graphs.

Algorithm 1: Invalid Tube
Boolean function *invalidTube* checks if an ill-formed tube cannot become well-formed by adding other tubes to the graph. This situation occurs when the tube is a loop or the tube vertices are proper ancestors or descendents of each other (Theorem 3). Invalid tubes must eventually be deleted in order to achieve well-formedness. The complexity of *invalidTube* is O(1).

```
function invalidTube(x,y : V) : boolean
    result x=y ∨ isAncestor(x,y) ∨ isAncestor(y,x)
end invalidTube
```
□

Algorithm 2: Well-Formed Tube
Boolean function *WFTube* checks if a tube is well-formed based on the well-formedness definition of Section 4.1. Since this function does not rely on the existence of the tube, the function can also be used to check if adding a tube to a well-formed tube graph preserves its well-formedness. The complexity of *WFTube* is O(1).

```
function WFTube(x,y : V) : boolean
    result ¬ (isRoot(x) ∨ isRoot(y)) ∧
            ((x≠y ∧ par(x)=par(y)) ∨ tube(par(x),par(y)) ∨ tube(par(x),y) ∨ tube(x,par(y)))
end WFTube
```
□

Algorithm 3: Well-Formed Tube Graph
Boolean function *isWFTubeGraph* checks the well-formedness of a tube graph rooted at r. Starting from r, it visits each vertex in depth-first order. For each vertex, it checks that all of the outgoing tubes from that vertex are well-formed. The complexity of *isWFTubeGraph* is $O(|V|+|T|)$. The complexity is not simply $O(|T|)$ because if the tube graph has no tubes, *isWFTubeGraph* must still visit all of the vertices, which requires $O(|V|)$.

```
function isWFTubeGraph(r : V) : boolean
    for i : 1 .. numOfOutgoingTubes(r)
        if ¬ WFTube(r, vertexOfIthOutgoingTube(r,i)) then
            result false
        end if
    end for
    for i : 1 .. numOfChildren(r)
        if ¬ isWFTubeGraph(ithChild(r,i)) then
            result false
        end if
    end for
    result true
end isWFTubeGraph
```
□

Algorithm 4: Making a Tube Well Formed
Procedure *makeTubeWF* deletes a tube if it is invalid or adds well-formed tubes to the tube graph in order to make a valid tube well formed (there may be more than one set of tubes that can accomplish this). It is important to note that adding well-formed tubes to a well-formed tube graph preserves well-formedness. Let xTy be the tube under consideration. There are three cases: 1) the levels of x and y are the same, 2) the level of x is less than the level of y, or 3) the level of x is greater than that of y. In case 2) the algorithm adds tubes from x and to the ancestors of y until the added tube is at the same level as x. In case 3) the algorithm adds tubes between the ancestors of x and y until the added tube is

at the same level as y. Then, the algorithm adds tubes between the parents of the vertices of the last tube added (if any), and so on, until the common ancestor is reached. Note that this algorithm does not guarantee that the minimum number of tubes will be added to the tube graph to make it well formed. The complexity of *makeTubeWF* is O($height$).

```
procedure makeTubeWF(x,y : V)
    precondition tube(x,y)
    if invalidTube(x,y) then
        deleteTube(x,y)
    elsif ¬WFTube(x,y) then
        p ← par(x)
        q ← par(y)
        if level(x) < level(y) then
            loop
                addTube(x,q)
                q ← par(q)
                exit when level(x) = level(q)–1
            end loop
        elsif level(x) > level(y) then
            loop
                addTube(p,y)
                p ← par(p)
                exit when level(y) = level(p)–1
            end loop
        end if
        loop
            exit when p = q
            addTube(p,q)
            p ← par(p)
            q ← par(q)
        end loop
    end if
end makeTubeWF
```
□

Algorithm 5: Making a Tube Graph Well Formed
Procedure *makeAllTubesWF* calls procedure *makeTubeWF* for all of the tubes in a tube graph rooted at r. The complexity of *makeAllTubesWF* is O($|V|$ + $|T| \times height$), because all of the tubes and vertices must be visited and each ill-formed tube must be made well formed.

```
procedure makeAllTubesWF(r : V)
    for i : 1 .. numOfOutgoingTubes(r)
        makeTubeWF(r, vertexOfIthOutgoingTube(r,i))
    end for
    for i : 1 .. numOfChildren(r)
        makeAllTubesWF(ithChild(r,i))
    end for
end makeAllTubesWF
```
□

Algorithm 6: Tube Deletion and Well-Formedness
Function *okToDeleteTube* returns the conjunction of two boolean values. The first value, *okToDelete*, indicates if deleting a tube pTq from a well-formed tube graph preserves its well-formedness. The second one, *tubesBeneath*, indicates if there are any tubes "beneath"

tube pTq. By "beneath" we mean any tubes between p (or the children of p) and q (or the children of q), except the tube between p and q. The *tubesBeneath* value is useful when checking for gratuitous tubes in the algorithm presented next. The complexity of *okToDeleteTube* is $O(maxTubeOut + maxChildren + maxChildTubes)$.

```
function okToDeleteTube(p,q : V) : boolean
    precondition tube(p,q) ∧ ¬ invalidTube(p,q)
    tubesBeneath ← false
    okToDelete ← true
    deleteTube(p,q)
    for i : 1 .. numOfOutgoingTubes(p)
        y ← vertexOfIthOutgoingTube(p,i)
        if par(y)=q then
            if ¬ WFTube(p,y) then
                okToDelete ← false
            end if
            tubesBeneath ← true
        end if
    end for
    for i : 1 .. numOfChildren(p)
        x ← ithChild(p,i)
        for j : 1 .. numOfOutgoingTubes(x)
            y ← vertexOfIthOutgoingTube(x,j)
            if par(y)=q ∨ y = q then
                if ¬ WFTube(x,y) then
                    okToDelete ← false
                end if
                tubesBeneath ← true
            end if
        end for
    end for
    addTube(p,q)
    result okToDelete ∧ tubesBeneath
end okToDeleteTube
```

□

Algorithm 7: Delete Gratuitous Tubes

Procedure *deleteGratuitousTubes* deletes all of those tubes whose tubes "beneath" them do not depend on them. Such tubes can be deleted from a well-formed tube graph without affecting the well-formedness of the tube graph. If the algorithm ignores checking that there are no tubes beneath the potentially gratuitous tube, by deleting the tube, the semantics of the tube graph change (a dependency is lost). To find and delete all gratuitous tubes, the algorithm traverses the containment tree and for each of its vertices it checks if the removal of each of its tubes does not leave any unsupported tubes. Note that since this algorithm starts from the top of the tree and moves toward the leaves, it will never delete gratuitous tubes between vertices on the same level, but will prefer to delete skewed gratuitous tubes. The complexity of *gratuitousTubes* is $O(|V| + |T| \times (maxTubeOut + maxChildren + maxChildTubes))$.

```
procedure deleteGratuitousTubes(r : V)
    for i : 1 .. numOfOutgoingTubes(r)
        x ← vertexOfIthOutgoingTube(r,i))
        if okToDeleteTube(r,x) then
            deleteTube(r,x)
        end if
```

```
        end for
        for i : 1 .. numOfChildren(r)
            deleteGratuitousTubes(ithChild(r,i))
        end for
    end deleteGratuitousTubes
```
□

6.2 Operations on Vertices

We now present algorithms for operating on the vertices of a tube graph and consider how these operations affect its well-formedness.

Algorithm 8: Adding a Vertex
Procedure *addVertex* adds a new vertex x as the child of vertex p (if $p = nil$, x becomes the root). Vertex x can be the root of a subtree, or a single vertex. The *selectedChildrenOfP* set contains those children of p that become children of x. The algorithm adds tubes from x to all vertices the selected children have tubes to. Similarly, it adds tubes to x from any vertices with tubes to the selected children. Note that these tubes are not added if both vertices in the tube are selected children, as this would add a tube to or from their ancestor. This preserves the well-formedness of the tube graph. The complexity of *addVertex* is $O(|V|\times height + maxChildren\times(|V|\times height + maxTubeOut + maxTubeIn))$. If there are no selected children, the complexity is $O(|V|\times height)$.

```
    procedure addVertex(p,x : V, selectedChildrenOfP : set of V)
        precondition selectedChildrenOfP = ∅ ∨ ∀ c ∈ selectedChildrenOfP • par(c)=p
        addChild(p,x)
        foreach c ∈ selectedChildrenOfP
            removeChild(p,c)
            addChild(x,c)
            for i : 1 .. numOfOutgoingTubes(c)
                y ← vertexOfIthOutgoingTube(c,i)
                if y ∉ selectedChildrenOfP then
                    addTube(x,y)
                end if
            end for
            for i : 1 .. numOfIncomingTubes(c)
                y ← vertexOfIthIncomingTube(c,i)
                if y ∉ selectedChildrenOfP then
                    addTube(y,x)
                end if
            end for
        end foreach
    end addVertex
```
□

Algorithm 9: Deleting a Vertex
Procedure *deleteVertex* deletes a vertex from a well-formed tube graph without deleting its children (it links them to the parent of the deleted vertex). To preserve well-formedness, it then replaces any incoming tubes to the deleted vertex by incoming tubes to its parent — except if this causes loops or tubes between ancestors or descendants. Similarly, it replaces any outgoing tubes from the deleted vertex to outgoing tubes from the parent of the deleted vertex. Note that the root vertex cannot be deleted, since all tube graphs must have a root. The complexity of *deleteVertex* is $O(maxTubeOut + maxTubeIn + maxChildren\times|V|\times height)$.

```
procedure deleteVertex(x : V)
    precondition ¬ isRoot(x)
    for i : 1 .. numOfChildren(x)
        addChild(par(x),ithChild(x,i))
    end for
    for i : 1 .. numOfOutgoingTubes(x)
        y ← vertexOfIthOutgoingTube(x,i)
        if par(x) ≠ par(y) then
            addTube(par(x),y)
        end if
        deleteTube(x,y)
    end for
    for i : 1 .. numOfIncomingTubes(x)
        y ← vertexOfIthIncomingTube(x,i)
        if par(x) ≠ par(y) then
            addTube(y,par(x))
        end if
        deleteTube(y,x)
    end for
    removeChild(par(x),x)
end deleteVertex
```

□

Algorithm 10: Deleting a Subtree
Procedure *deleteSubtree* deletes the subtree rooted at x. To preserve well-formedness, the procedure ensures that any incoming and outgoing tubes to or from the vertices in the deleted subtree are deleted. The complexity of *deleteSubtree* is $\mathrm{O}(|V| \times height + |T|)$ because, in the worst case, x can be the root of the containment tree and the complexity of *removeChild*, when removing leaf vertices, is $\mathrm{O}(height)$.

```
procedure deleteSubtree(x : V)
    for i : 1 .. numOfChildren(x)
        deleteSubtree(ithChild(x,i))
    end for
    for i : 1 .. numOfOutgoingTubes(x)
        deleteTube(x, ithOutgoingTube(x,i))
    end for
    for i : 1 .. numOfIncomingTubes(x)
        deleteTube(vertexOfIthIncomingTube(x,i),x)
    end for
    removeChild(par(x),x)
end deleteSubtree
```

□

This concludes the description of our algorithms for manipulating tube graphs. Following, is a synopsis of the paper and directions for future research.

7 Summary and Future Research

We described tube graphs as mathematical entities and showed how they can be used — by imposing restrictions on tubes — to model particular styles of architectural designs. First, we described how tube graphs fit into our general software development environment framework for specifying and visualizing architectural designs. Then, we formally defined general tube graphs, as well as a restricted form, called well-formed tube graphs. Well-formed tube graphs model SIL architectural designs. These tube graphs were then specified using a graph grammar. We proved that the grammar generates exactly the set of all well-formed tube graphs. Finally, algorithms for manipulating well-formed tube graphs were specified and, assuming the data structures of appendix A, the complexity of each algorithm was given.

In the future, we plan to use tube graphs to model architectural designs based on the entire SIL language which includes relations such as *import*, *export*, and, possibly later on, *inheritance*, and *data flow*. We are also interested in using tube graphs to model designs and programs of other architectural design and programming languages. A technical problem that we are investigating is the automatic generation of tools for architectural design from graph grammar specifications.

A Data Structures and Primitive Operations

This appendix defines the primitive operations used by the algorithms described in Section 6. These primitive operations are used for manipulating the data structures described below.

A.0.1 Data Structures

The data structures are designed to optimize the speed of the primitive operations that deal with tubes. We assume that the available space is sufficient for any tube graph. In an implementation, however, we could trade-off some speed to reduce the space requirements by using hashing. The data structures are defined as follows:

```
type V : pointer to Vtype

const maxVertex : natural ← some very large number

type Vtype :
    record
        id : 1 .. maxVertex
        parent : V

        hasChild, hasTubeTo, hasTubeFrom, hasAncestor : array 1 .. maxVertex of 0 .. maxVertex
        child, tubeTo, tubeFrom, ancestor : array 1 .. maxVertex of V
        numOfChildren, numOfTubesTo, numOfTubesFrom, numOfAncestors : 0 .. maxVertex
    end record
```

For a vertex x, *x.id* [7] uniquely identifies it. A pointer to the parent of x is stored in *x.parent* (is nil if x is the root). The *x.hasChild* array is used for quickly determining the children of x. If a vertex v is not a child of x, *x.hasChild(v.id)* is 0, if it is, *x.child(x.hasChild(v.id))* is v. Variables *x.child* and *x.numOfChildren* are useful when iterating across the children of x. Similar data structures are employed for the outgoing tubes from x (*x.tubeTo, x.hasTubeTo, x.numOfTubesTo*), the incoming tubes to x (*x.tubesFrom,*

[7] We use "." for pointer dereferencing.

x.hasTubeFrom, x.numOfTubesFrom), and the ancestors of x (*x.ancestor, x.hasAncestor, x.numOfAncestors*).

We next present the operations for manipulating the data structures.

A.0.2 Primitive Operations

The primitive operations are used by the tube graph operations described in Section 6. These primitive operations depend on a few supporting operations — which keep track of ancestor information — defined in the next section.

Algorithm 11: Parent of a Vertex
Function *par* returns the parent of vertex x or the value *nil*, if x is the root. Its complexity is O(1).

```
function par(x : V) : V
      precondition x ≠ nil
      result x.parent
end par
```

□

Algorithm 12: Vertex is the Root
Boolean function *isRoot* checks if x is the root of the containment tree. Its complexity is O(1).

```
function isRoot(x : V) : boolean
      precondition x ≠ nil
      result par(x) = nil
end isRoot
```

□

Algorithm 13: Vertex is an Ancestor of Another Vertex
Boolean function *isAncestor* checks if x is a proper ancestor of y. Its complexity is O(1).

```
function isAncestor(x,y : V) : boolean
      precondition x ≠ nil ∧ y ≠ nil
      result y.hasAncestor(x.id) ≠ 0
end isAncestor
```

□

Algorithm 14: Level of a Vertex
Function *level* returns the level of vertex x in the containment tree (*level*=0 for the root). Its complexity is O(1).

```
function level(x : V) : 0 .. maxVertex
      precondition x ≠ nil
      result x.numOfAncestors
end level
```

□

Algorithm 15: Number of Children
Function *numOfChildren* returns the number of children of x. Its complexity is O(1).

```
function numOfChildren(x : V) : 0 .. maxVertex
      precondition x ≠ nil
      result x.numOfChildren
end numOfChildren
```
□

Algorithm 16: Number of Outgoing Tubes from a Vertex to Other Vertices
Function *numOfOutgoingTubes* returns the number of tubes from x to other vertices. Its complexity is O(1).

```
function numOfOutgoingTubes(x : V) : 0 .. maxVertex
      precondition x ≠ nil
      result x.numOfTubesTo
end numOfOutgoingTubes
```
□

Algorithm 17: Number of Incoming Tubes to a Vertex from Other Vertices
Function *numOfIncomingTubes* returns the number of tubes to x from other vertices. Its complexity is O(1).

```
function numOfIncomingTubes(x : V) : 0 .. maxVertex
      precondition x ≠ nil
      result x.numOfTubesFrom
end numOfIncomingTubes
```
□

Algorithm 18: Tube Between two Vertices
Boolean function *tube* checks if there is a tube between x and y. Its complexity is O(1).

```
function tube(x,y : V) : boolean
      precondition x ≠ nil ∧ y ≠ nil
      result x.hasTubeTo(y.id) ≠ 0
end tube
```
□

Algorithm 19: Adding a Tube
Procedure *addTube* updates the tube lists of x and y to indicate that there is a tube between them. Its complexity is O(1).

```
procedure addTube(x,y : V)
      precondition x ≠ nil ∧ y ≠ nil
      if ¬ tube(x,y) then
            x.numOfTubesTo ← x.numOfTubesTo + 1
            x.hasTubeTo(y.id) ← x.numOfTubesTo
            x.tubeTo(x.hasTubeTo(y.id)) ← y
            y.numOfTubesFrom ← y.numOfTubesFrom + 1
            y.hasTubeFrom(x.id) ← y.numOfTubesFrom
            y.tubeFrom(y.hasTubeFrom(x.id)) ← x
      end if
end addTube
```
□

Algorithm 20: Deleting a Tube
Procedure *deleteTube* updates the tube lists of x and y to indicate that there is no tube between them. Its complexity is O(1).

```
procedure deleteTube(x,y : V)
    precondition x ≠ nil ∧ y ≠ nil
    if tube(x,y) then
        x.tubeTo(x.hasTubeTo(y.id)) ← x.tubeTo(x.numOfTubesTo)
        x.hasTubeTo((x.tubeTo(x.numOfTubesTo)).id) ← x.hasTubeTo(y.id)
        x.hasTubeTo(y.id) ← 0
        x.tubeTo(x.numOfTubesTo) ← nil
        x.numOfTubesTo ← x.numOfTubesTo – 1
        y.tubeFrom(y.hasTubeFrom(x.id)) ← y.tubeFrom(y.numOfTubesFrom)
        y.hasTubeFrom((y.tubeFrom(y.numOfTubesFrom)).id) ← y.hasTubeFrom(x.id)
        y.hasTubeFrom(x.id) ← 0
        y.tubeFrom(y.numOfTubesFrom) ← nil
        y.numOfTubesFrom ← y.numOfTubesFrom – 1
    end if
end deleteTube
```

□

Algorithm 21: Adding a Child

Procedure *addChild* first assigns x to be the parent of y. It then updates the child lists of x to indicate that y is a child of x. Finally, it creates the ancestor lists of all vertices in the subtree rooted at y. The complexity of *addChild* is $O(|V| \times height)$.

```
procedure addChild(x,y : V)
    precondition y ≠ nil
    assignParent(y,x)
    if x ≠ nil then
        if x.hasChild(y.id) = 0 then
            x.numOfChildren ← x.numOfChildren + 1
            x.hasChild(y.id) ← x.numOfChildren
            x.child(x.hasChild(y.id)) ← y
            createAncestorsOfSubtree(y)
        end if
    end if
end addChild
```

□

Algorithm 22: Removing a Child

Procedure *removeChild* updates the child lists of x to indicate that y is not a child of x. It then removes x from being the parent of y. Finally, it creates the ancestor lists of all vertices in the subtree rooted at y. The complexity of *removeChild* is $O(|V| \times height)$.

```
procedure removeChild(x,y : V)
    precondition x ≠ nil ∧ y ≠ nil
    if x.hasChild(y.id) ≠ 0 then
        x.child(x.hasChild(y.id)) ← x.child(x.numOfChildren)
        x.hasChild((x.child(x.numOfChildren)).id) ← x.hasChild(y.id)
        x.hasChild(y.id) ← 0
        x.child(x.numOfChildren) ← nil
        x.numOfChildren ← x.numOfChildren – 1
    end if
    removeParent(y,x)
    createAncestorsOfSubtree(y)
end removeChild
```

□

Algorithm 23: Ith Child of a Vertex
Function *ithChild* returns the ith child of x. This function is useful when iterating across the children of x. Its complexity is O(1).

```
function ithChild(x : V, i : 1 .. maxVertex) : V
    precondition (x ≠ nil) ∧ (1 ≤ i ≤ numOfChildren(x))
    result x.child(i)
end ithChild
```

□

Algorithm 24: Ith Outgoing Tube From a Vertex
Function *vertexOfIthOutgoingTube* returns the ith tube from x to another vertex. This function is useful when iterating across all of the tubes from x to other vertices. Its complexity is O(1).

```
function vertexOfIthOutgoingTube(x : V, i : 1 .. maxVertex) : V
    precondition (x ≠ nil) ∧ (1 ≤ i ≤ numOfOutgoingTubes(x))
    result x.tubeTo(i)
end vertexOfIthOutgoingTube
```

□

Algorithm 25: Ith Incoming Tube to a Vertex
Function *vertexOfIthIncomingTube* returns the ith tube to x from another vertex. This function is useful when iterating across all of the tubes to x from other vertices. Its complexity is O(1).

```
function vertexOfIthIncomingTube(x : V, i : 1 .. maxVertex) : V
    precondition (x ≠ nil) ∧ (1 ≤ i ≤ numOfIncomingTubes(x))
    result x.tubeFrom(i)
end vertexOfIthIncomingTube
```

□

A.1 Supporting Primitive Operations

The following supporting operations are used by the primitive operations defined previously.

Algorithm 26: Assign a Parent to a Vertex
Procedure *assignParent*, used only by *addChild*, assigns vertex p to be the parent of vertex x. Its complexity is O(1).

```
procedure assignParent(x,p : V)
    precondition x ≠ nil
    result x.parent ← p
end assignParent
```

□

Algorithm 27: Initializing Ancestors
Procedure *initAncestors* resets the ancestor list of x. It is only used by *createAncestorsOf-Subtree* and has complexity O(*height*).

```
procedure initAncestors(x : V)
    for i : 1 .. x.numOfAncestors
        x.hasAncestor(x.ancestor(i)) ← 0
    end for
    x.numOfAncestors ← 0
ens initAncestors
```

□

Algorithm 28: Creating Ancestors
Procedure *createAncestors* creates a list of x's ancestors. It is only used by *createAncestorsOfSubtree* and has complexity O($height$).

```
procedure createAncestors(x : V)
    precondition x ≠ nil
    p ← x
    loop
        exit when isRoot(p)
        p ← par(p)
        x.numOfAncestors ← x.numOfAncestors + 1
        x.hasAncestor(p.id) ← x.numOfAncestors
        x.ancestor(x.hasAncestor(p.id)) ← p
    end loop
end createAncestors
```

□

Algorithm 29: Creating the Ancestors of a Subtree
Procedure *createAncestorsOfSubtree* creates the ancestor list of each vertex of the subtree rooted at x. Its complexity is O($|V| \times height$).

```
procedure createAncestorsOfSubtree(x : V)
    precondition x ≠ nil
    initAncestors(x)
    createAncestors(x)
    for i : 1 .. numOfChildren(x)
        createAncestorsOfSubtree(ithChild(x,i))
    end end
end createAncestorsOfSubtree
```

□

References

[1] Claus, V., Ehrig, H., and Rozenberg, G. In *Graph-Grammars and Their Application to Computer Science and Biology* (Berlin, Germany, 1978), Springer–Verlag.

[2] Consens, M., Eigler, F., Hasan, M., Mendelzon, A., Noik, M., Ryman, A., and Vista, D. Architecture and Applications of the Hy$^+$ Visualization System. *IBM Systems Journal 33*, 3 (1994).

[3] Cooprider, L. W. The Representation of Families of Software Systems. Tech. Rep. CMU–CS–79–116, Computer Science Department CMU, April 1979.

[4] Dean, T. R., and Lamb, D. A. A Theory Model Core for Module Interconnection Languages. In *Proceedings of the 1994 IBM CASCON Conference* (November 1994).

[5] DeRemer, F., and Kron, H. H. Programming–in–the–Large Versus Programming–in–the–Small. *IEEE Transactions on Software Engineering 2*, 2 (June 1976), 80–86.

[6] Fenton, N., and Hill, G. *Systems Construction and Analysis: A Mathematical and Logical Framework*. McGraw-Hill International, 1993.

[7] Garlan, D., and Shaw, M. Architectures for Software Systems. In *tutorial given at ACM SIGSOFT '93: Symposium on the Foundations of Software Engineering* (Los Angeles, California, December 1993).

[8] Holt, R. C., and Cordy, J. R. The Turing Programming Language. *Communications of the ACM 31*, 12 (December 1988), 1410–1423.

[9] Holt, R. C., and Mancoridis, S. A Framework For Specifying and Visualizing Architectural Designs. Tech. Rep. CSRI-300, Computer Science Research Institute, University of Toronto, June 1994.

[10] Holt, R. C., Penny, D. A., and Mancoridis, S. Multicolour Programming and Metamorphic Programming: Object Oriented Programming–in–the–Large. In *Proceedings of the 1992 IBM CASCON Conference* (November 1992), pp. 43–58.

[11] Kreowski, H. A Pumping Lemma for Context-Free Graph Languages. In *Graph-Grammars and Their Application to Computer Science and Biology* (Berlin, Germany, 1978), Springer–Verlag, pp. 270–283.

[12] Mendelzon, A. O. Declarative Database Visualization: Recent Papers from the Hy+/GraphLog Project. Tech. Rep. CSRI-285, Computer Science Research Institute, University of Toronto, 1993.

[13] Penny, D. A. *The Software Landscape: A Visual Formalism for Programming–in–the–Large*. PhD thesis, Department of Computer Science, University of Toronto, 1992.

[14] Perry, D. E., and Wolf, A. L. Foundations for the Study of Software Architectures. *Software Engineering Notes 17*, 4 (October 1992), 40–49.

[15] Schwanke, R. W., Altucher, R. Z., and Platoff, M. A. Discovering, Visualizing, and Controlling Software Structure. In *Proceedings of the Fifth International Workshop on Software Specification and Design* (Pittsburgh, Pennsylvania, May 1989), pp. 147–150.

[16] Tichy, W. F. Software Development Control Based on System Structure Description. Tech. Rep. CMU–CS–80–120, Computer Science Department CMU, January 1980.